

32nd European Conference on Object-Oriented Programming

ECOOP 2018, July 16–21, 2018, Amsterdam, The Netherlands

Edited by

Todd Millstein



Editor

Todd Millstein
Computer Science Department
University of California, Los Angeles
todd@cs.ucla.edu

ACM Classification 2012
Software and its engineering

ISBN 978-3-95977-079-8

Published online and open access by

Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at <http://www.dagstuhl.de/dagpub/978-3-95977-079-8>.

Publication date

July, 2018

Bibliographic information published by the Deutsche Nationalbibliothek

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

License

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



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

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

The copyright is retained by the corresponding authors.

Digital Object Identifier: 10.4230/LIPIcs.ECOOP.2018.0

ISBN 978-3-95977-079-8

ISSN 1868-8969

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

LIPICs – Leibniz International Proceedings in Informatics

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

Editorial Board

- Luca Aceto (*Chair*, Gran Sasso Science Institute and Reykjavik University)
- Susanne Albers (TU München)
- Chris Hankin (Imperial College London)
- Deepak Kapur (University of New Mexico)
- Michael Mitzenmacher (Harvard University)
- Madhavan Mukund (Chennai Mathematical Institute)
- Anca Muscholl (University Bordeaux)
- Catuscia Palamidessi (INRIA)
- Raimund Seidel (Saarland University and Schloss Dagstuhl – Leibniz-Zentrum für Informatik)
- Thomas Schwentick (TU Dortmund)
- Reinhard Wilhelm (Saarland University)

ISSN 1868-8969

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

■ Contents

Message from the Chairs <i>Frank Tip and Todd Millstein</i>	0:vii–0:viii
Message from the President of AITO <i>Eric Jul</i>	0:ix
ECOOP 2018 Conference Organization	0:xi–0:xiii
External Reviewers	0:xv
List of Authors	0:xvii–0:xx

Regular Papers

Fault-tolerant Distributed Reactive Programming <i>Ragnar Mogk, Lars Baumgärtner, Guido Salvaneschi, Bernd Freisleben, and Mira Mezini</i>	1:1–1:26
ContextWorkflow: A Monadic DSL for Compensable and Interruptible Executions <i>Hiroaki Inoue, Tomoyuki Aotani, and Atsushi Igarashi</i>	2:1–2:33
Theory and Practice of Coroutines with Snapshots <i>Aleksandar Prokopec and Fengyun Liu</i>	3:1–3:32
A Concurrent Specification of POSIX File Systems <i>Gian Ntzik, Pedro da Rocha Pinto, Julian Sutherland, and Philippa Gardner</i>	4:1–4:28
A Characteristic Study of Parameterized Unit Tests in .NET Open Source Projects <i>Wing Lam, Siwakorn Srisakaokul, Blake Bassett, Peyman Mahdian, Tao Xie, Pratap Lakshman, and Jonathan de Halleux</i>	5:1–5:27
Learning to Accelerate Symbolic Execution via Code Transformation <i>Junjie Chen, Wenxiang Hu, Lingming Zhang, Dan Hao, Sarfraz Khurshid, and Lu Zhang</i>	6:1–6:27
Type Regression Testing to Detect Breaking Changes in Node.js Libraries <i>Gianluca Mezzetti, Anders Møller, and Martin Toldam Torp</i>	7:1–7:24
Targeted Test Generation for Actor Systems <i>Sihan Li, Farah Hariri, and Gul Agha</i>	8:1–8:31
Typed First-Class Traits <i>Xuan Bi and Bruno C. d. S. Oliveira</i>	9:1–9:28
CrySL: An Extensible Approach to Validating the Correct Usage of Cryptographic APIs <i>Stefan Krüger, Johannes Späth, Karim Ali, Eric Bodden, and Mira Mezini</i>	10:1–10:27

32nd European Conference on Object-Oriented Programming (ECOOP 2018).
Editor: Todd Millstein



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

Safe Transferable Regions <i>Gowtham Kaki and G. Ramalingam</i>	11:1–11:31
KafKa: Gradual Typing for Objects <i>Benjamin Chung, Paley Li, Francesco Zappa Nardelli, and Jan Vitek</i>	12:1–12:25
Dependent Types for Class-based Mutable Objects <i>Joana Campos and Vasco T. Vasconcelos</i>	13:1–13:28
Static Typing of Complex Presence Constraints in Interfaces <i>Nathalie Oostvogels, Joeri De Koster, and Wolfgang De Meuter</i>	14:1–14:27
Mailbox Types for Unordered Interactions <i>Ugo de'Liguoro and Luca Padovani</i>	15:1–15:28
Accelerating Dynamically-Typed Languages on Heterogeneous Platforms Using Guards Optimization <i>Mohaned Qunaibit, Stefan Brunthaler, Yeoul Na, Stijn Volckaert, and Michael Franz</i>	16:1–16:29
CROCHET: Checkpoint and Rollback via Lightweight Heap Traversal on Stock JVMs <i>Jonathan Bell and Luís Pina</i>	17:1–17:31
ThingsMigrate: Platform-Independent Migration of Stateful JavaScript IoT Applications <i>Julien Gascon-Samson, Kumseok Jung, Shivanshu Goyal, Armin Rezaiean-Asel, and Karthik Pattabiraman</i>	18:1–18:33
Automating Object Transformations for Dynamic Software Updating via Online Execution Synthesis <i>Tianxiao Gu, Xiaoxing Ma, Chang Xu, Yanyan Jiang, Chun Cao, and Jian Lu</i> ..	19:1–19:28
FHJ: A Formal Model for Hierarchical Dispatching and Overriding <i>Yanlin Wang, Haoyuan Zhang, Bruno C. d. S. Oliveira, and Marco Servetto</i>	20:1–20:30
Modeling Infinite Behaviour by Corules <i>Davide Ancona, Francesco Dagnino, and Elena Zucca</i>	21:1–21:31
The Essence of Nested Composition <i>Xuan Bi, Bruno C. d. S. Oliveira, and Tom Schrijvers</i>	22:1–22:33
Defensive Points-To Analysis: Effective Soundness via Laziness <i>Yannis Smaragdakis and George Kastrinis</i>	23:1–23:28
LEGATO: An At-Most-Once Analysis with Applications to Dynamic Configuration Updates <i>John Toman and Dan Grossman</i>	24:1–24:32
Definite Reference Mutability <i>Ana Milanova</i>	25:1–25:30
Efficient Reflection String Analysis via Graph Coloring <i>Neville Grech, George Kastrinis, and Yannis Smaragdakis</i>	26:1–26:25

■ Message from the Chairs

It is our great pleasure to welcome you to Amsterdam for ECOOP 2018, the 32nd European Conference on Object-Oriented Programming, to be held on July 16-21. ECOOP is the European forum for bringing together researchers, practitioners, and students to share their ideas and experiences in all topics related to programming languages, software development, object-oriented technologies, systems and applications.

This year, ECOOP is co-located with the International Symposium on Software Testing and Analysis (ISSTA 2018), and with CurryOn, a conference that is focused on the intersection of emerging languages and emerging challenges in industry. The ECOOP 2018 program includes technical papers and keynotes. Furthermore, ECOOP 2018 features workshops, a doctoral symposium, a posters session, and a summer school that are jointly organized with ISSTA 2018.

ECOOP 2018 received 66 submissions of research papers. Each submission was evaluated by at least three members of the Program Committee (PC). Similar to ECOOP 2017, a light-weight double-blind reviewing process was adopted in which authors did not reveal their name, identity, or affiliation in their submissions. Author identities were revealed to a reviewer only once his/her review was submitted. Authors were also given an opportunity to provide a response to the reviews before decisions were made.

The Program Committee met in Amsterdam, The Netherlands on April 5 and 6 to discuss the submissions and accepted 26 papers (39.4% acceptance rate). One paper, “Defensive Points-To Analysis: Effective Soundness via Laziness,” by Yannis Smaragdakis and George Kastrinis, was selected to receive an AITO Distinguished Paper Award. We thank our 24 PC members for their thorough reviewing and for attending the two-day physical PC meeting.

Authors of accepted papers were also invited to submit artifacts, which were evaluated by a separate Artifact Evaluation Committee (AEC). The committee received 13 artifacts and accepted 10 of them. We thank our Artifact Evaluation Chairs Maria Christakis, Philipp Haller, and Marianna Rapoport.

The ECOOP 2018 program includes three keynote talks. Harry Xu (University of California, Los Angeles) will present “Object-Orientation Meets Big Data: Performance Impact, Restoration, and Thoughts on Language Design” as part of winning the AITO Dahl-Nygaard Junior Prize in 2018. (The AITO Dahl-Nygaard Senior Prize was awarded to Lars Bak, who unfortunately is unable to attend ECOOP 2018.) The other two keynote talks are “Program Analysis for Everyone” by Oege de Moore (Semmler) and “Parser-Directed Test Generation” by Andreas Zeller (Saarland University).

The following workshops are colocated with ECOOP 2018 and ISSTA 2018:

- COP: 10th International Workshop on Context-Oriented Programming
- BenchWork: First Workshop on Reproducible Experiments and Benchmarking
- DPA: First Workshop on Declarative Program Analysis
- FTfJP: 20th Workshop on Formal Techniques for Java-like Programs
- ICOOLPS: 13th Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems Workshop
- INTUITESTBEDS: 4th Workshop on UI Test Automation & 8th Workshop on TESTing Techniques for event BasED Software
- ISAGT: Introspective Systems for Automatically Generating Tests
- ML4PL: 2nd International Workshop on Machine Learning for Programming Languages
- Panathon: Program analysis hackathon

32nd European Conference on Object-Oriented Programming (ECOOP 2018).
Editor: Todd Millstein



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

- SALAD: First International Workshop on Software debloating And Delaying
- SOAP: 7th International Workshop on the State Of the Art in Program Analysis
- TAV-CPS/IoT: Testing, Analysis, and Verification of Cyber-Physical Systems and Internet of Things
- VORTEX: Runtime verification Workshop
- WoSSCA: First International Workshop on Speculative Side Channel Analysis

We thank our Workshop Chairs Julian Dolby and William G.J. Halfond for putting together this comprehensive workshops program.

Bringing ECOOP 2018 into existence involved a great effort from the members of our community, and we would like to extend our most heartfelt thanks to everyone who worked hard to make this possible. We thank the authors and the keynote speakers, who provided the content of our program and are presenting their contributions at the conference. We cannot thank enough the Program Committee members, who reviewed many papers, wrote detailed reviews, travelled at their own expense to the Program Committee meeting, and helped us select the great program for the conference.

Many other people contributed to various aspects of the program: the Doctoral Symposium was chaired by Julia Belyakova, Cristian Cadar, and Jasper Schulz; Sarah Mount served as Diversity Chair; Annabel Satin served as Finance Chair; Jan Vitek and Andreas Zeller co-organized the Summer School; Alisa Maas and Ming-Ho Yee served as Student Volunteer Co-Chairs; Karim Ali managed the ECOOP web site; Nicolás Rosner served as Publicity Chair; Laurence Tratt served as Sponsorship Chair; Lisa Nguyen served as Posters Chair; and last but not least, Tijs van der Storm and Jurgen Vinju served as Local Organizing Chairs for ECOOP/ISSTA 2018.

We gratefully acknowledge our sponsor AITO as well as our financial supporters — Amazon, Facebook, Google, IBM Research, JetBrains, Mozilla, the National Science Foundation, the Office of Naval Research, Oracle, Semmle, Uber, and VMware — for their generous contributions. Finally, we want to thank all of the conference attendees for contributing to making the conference a success. We hope that you will find the ECOOP 2018 program interesting, thought-provoking, and inspiring, and that the conference will give you valuable opportunities to share ideas with researchers and practitioners from our vibrant community.

Frank Tip
ECOOP 2018 General Chair
Northeastern University

Todd Millstein
ECOOP 2018 Program Chair
University of California, Los Angeles

■ Message from the President of AITO

This year ECOOP is held in Amsterdam, co-located with ISSTA and Curry On. Object-oriented programming has moved from an esoteric, academic endeavor to be so mainstream that most undergraduate introductory programming courses in the world use some kind of object-oriented language. The number of different and specialized conferences has meant that there are many other events that attract the traditional ECOOP audience. So AITO believes that co-location with other events can provide cross-feed between different areas and different environments; specifically, Curry-On's mix of industry and academia is great — even fantastic at times.

This year, the Dahl-Nygaard prizes go to practitioners of object-oriented languages having contributed to efficient OO programs. The Senior prize goes to Lars Bak, who has spent decades as a practitioner trying — successfully — to build efficient implementations of object-oriented languages. Unfortunately, he will not be able to accept the prize at ECOOP 2018 — we will, hopefully, be able to invite him to ECOOP 2019 for a talk. The Junior Prize goes to Guoqing Harry Xu who has made significant contributions to different aspects of object orientation due to a unique combination of technical strength and ambition to deliver effective object-oriented programming technologies for big data systems.

On behalf of AITO, I would like to thank the people, including you, who contribute to making ECOOP 2018 a successful conference; we hope that you will find it inspiring and, we hope, even fun.

Eric Jul
AITO President
University of Oslo



■ ECOOP 2018 Conference Organization

General Chair

Frank Tip (*Northeastern University, USA*)

Program Chair

Todd Millstein (*University of California, Los Angeles, USA*)

Artifact Evaluation Co-Chairs

Maria Christakis (*MPI-SWS, Germany*)

Philipp Haller (*KTH Royal Institute of Technology, Sweden*)

Marianna Rapoport (*University of Waterloo, Canada*)

Workshop Co-Chairs

Julian Dolby (*IBM T.J. Watson Research Center, USA*)

William G.J. Halfond (*University of Southern California, USA*)

Webmaster

Karim Ali (*University of Alberta, Canada*)

Publicity Chair

Nicolás Rosner (*University of California, Santa Barbara, USA*)

Diversity Chair

Sarah Mount (*King's College London, UK*)

Finance Chair

Annabel Satin (*Petit Canard Kitchen, UK*)

Posters Chair

Lisa Nguyen (*Paderborn University, Germany*)

Summer School Co-Chairs

Jan Vitek (*Northeastern University, USA*)

Andreas Zeller (*Saarland University, Germany*)

Doctoral Symposium Co-Chairs

Julia Belyakova (*Czech Technical University, Czech Republic*)

Cristian Cadar (*Imperial College London, UK*)

Jasper Schulz (*King's College London, UK*)

Student Volunteer Co-Chairs

Alisa Maas (*University of Wisconsin-Madison, USA*)

Ming-Ho Yee (*Northeastern University, USA*)



Local Organizing Co-Chairs

Jurgen Vinju (*CWI, The Netherlands*)
Tijs van der Storm (*CWI, The Netherlands*)

Program Committee

Karim Ali (*University of Alberta, Canada*)
Nada Amin (*University of Cambridge, UK*)
Earl Barr (*University College London, UK*)
Michael Carbin (*MIT, USA*)
Sophia Drossopoulou (*Imperial College London, UK*)
Christian Hammer (*University of Potsdam, Germany*)
Robert Hirschfeld (*HPI at the University of Potsdam, Germany*)
Atsushi Igarashi (*Kyoto University, Japan*)
Mohsen Lesani (*University of California, Riverside, USA*)
Yu David Liu (*State University of New York at Binghamton, USA*)
Magnus Madsen (*University of Waterloo, Canada*)
Nate Nystrom (*Università della Svizzera italiana, Switzerland*)
Michael Pradel (*TU Darmstadt, Germany*)
Murali Ramanathan (*Uber, USA*)
Jennifer Sartor (*Vrije Universiteit Brussel, Belgium*)
Ina Schaefer (*TU Braunschweig, Germany*)
Max Schaefer (*Semmler, UK*)
Xipeng Shen (*North Carolina State University, USA*)
Jeremy Siek (*Indiana University Bloomington, USA*)
Scott Smith (*The Johns Hopkins University, USA*)
Tijs van der Storm (*CWI / University of Groningen, Holland*)
Gang Tan (*The Pennsylvania State University, USA*)
Peter Thiemann (*Freiburg University, Germany*)
Mandana Vaziri Tardieu (*IBM T. J. Watson Research Center, USA*)

Artifact Evaluation Committee

Ambrose Bonnaire-Sergeant (*Indiana University, USA*)
Elias Castegren (*Uppsala University, Sweden*)
Ezgi Cicek (*MPI-SWS, Germany*)
Ankush Desai (*UC Berkeley, USA*)
Jonathan Eyolfson (*University of Waterloo, Canada*)
Yu Feng (*UT Austin, USA*)
Hugo Feree (*University of Kent, UK*)
Thomas Gilray (*University of Maryland, USA*)
Stefan Heule (*Stanford University, USA*)
Ravichandhran Madhavan (*EPFL, Switzerland*)
Guillaume Martres (*EPFL, Switzerland*)
Gianluca Mezzetti (*Aarhus University, Denmark*)
Fabian Muehlboeck (*Cornell University, USA*)
Filip Niksic (*MPI-SWS, Germany*)
Alceste Scalas (*Imperial College London, UK*)
Emma Tosch (*UMass Amherst, USA*)
Ming-Ho Yee (*Northeastern University, USA*)

Doctoral Symposium Committee

Abdulmajeed Alameer (*University of Southern California, USA*)
Mateus Borges (*Imperial College London, UK*)
Benjamin Chung (*Northeastern University, USA*)
Raimil Cruz (*University of Chile, Chile*)
Alex Gyori (*Facebook, USA*)
Darya Melicher (*Carnegie Mellon University, USA*)
Manuel Rigger (*Johannes Kepler University Linz, Austria*)
Christopher Schuster (*University of California, Santa Cruz, USA*)
Justin Smith (*North Carolina State University, USA*)
Tyler Sorensen (*Imperial College London, UK*)
Wei Sun (*University of Nebraska-Lincoln, USA*)
Vanya Yaneva (*University of Edinburgh, UK*)

Posters Committee

Michael Carbin (*MIT, USA*)
William G.J. Halfond (*University of Southern California, USA*)
Kathryn Stolee (*North Carolina State University, USA*)
Chao Wang (*University of Southern California, USA*)
Francesco Zappa Nardelli (*Inria, France*)

■ External Reviewers

Robert Bocchino
Cristian Cadar
Adam Chlipala
Wolfgang De Meuter
Matthew Hammer
Ben Hardekopf
Naoki Kobayashi
Doug Lea
Anders Moeller
Soo-Mook Moon
Ilya Sergey
Jerome Simeon
Tom Van Cutsem
Panagiotis Vekris
Christian Wimmer
Hongwei Xi



■ List of Authors

Gul Agha (8)

Department of Computer Science, University
of Illinois at Urbana-Champaign, Urbana,
USA
agha@illinois.edu

Karim Ali (10)

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

Davide Ancona (21)

DIBRIS, University of Genova, Italy
davide.ancona@unige.it
<https://orcid.org/0000-0002-6297-2011>

Tomoyuki Aotani (2)

School of Computing, Tokyo Institute of
Technology, Tokyo, Japan
aotani@c.titech.ac.jp

Blake Bassett (5)

University of Illinois at Urbana-Champaign,
USA
rbasset2@illinois.edu

Lars Baumgärtner (1)

Philipps-Universität Marburg, Germany

Jonathan Bell (17)

George Mason University, Fairfax, VA, USA
bellj@gmu.edu
<https://orcid.org/0000-0002-1187-9298>

Xuan Bi (9, 22)

The University of Hong Kong, Hong Kong,
China
xbi@cs.hku.hk

Eric Bodden (10)

Paderborn University & Fraunhofer IEM,
Germany
eric.bodden@uni-paderborn.de

Stefan Brunthaler (16)

National Cyber Defense Research Institute
CODE, Munich, and SBA Research
brunthaler@unibw.de

Joana Campos (13)

LASIGE, Faculdade de Ciências,
Universidade de Lisboa, Portugal
jcampos@lasige.di.fc.ul.pt
<https://orcid.org/0000-0002-2185-8175>

Chun Cao (19)

State Key Laboratory of Novel Software
Technology, Nanjing University, Nanjing,
China
caochun@nju.edu.cn

Junjie Chen (6)

Key Laboratory of High Confidence Software
Technologies (Peking University), MoE,
Institute of Software, EECS, Peking
University, Beijing, 100871, China
chenjunjie@pku.edu.cn

Benjamin Chung (12)

Northeastern University, Boston, MA, USA

Francesco Dagnino (21)

DIBRIS, University of Genova, Italy
francesco.dagnino@dibris.unige.it
<https://orcid.org/0000-0003-3599-3535>

Jonathan de Halleux (5)

Microsoft Research, USA
jhalleux@microsoft.com

Ugo de'Liguoro (15)

Università di Torino, Dipartimento di
Informatica, Torino, Italy
deligu@di.unito.it
<https://orcid.org/0000-0003-4609-2783>

Michael Franz (16)

University of California, Irvine
franz@uci.edu

Bernd Freisleben (1)

Philipps-Universität Marburg, Germany

Philippa Gardner (4)

Imperial College London, UK
pg@doc.ic.ac.uk

32nd European Conference on Object-Oriented Programming (ECOOP 2018).
Editor: Todd Millstein



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

- Julien Gascon-Samson (18)
Electrical and Computer Engineering
Department, University of British Columbia,
2332 Main Mall, Vancouver, BC, Canada,
V6T 1Z4
julien.gascon-samson@ece.ubc.ca
- Shivanshu Goyal (18)
Electrical and Computer Engineering
Department, University of British Columbia,
2332 Main Mall, Vancouver, BC, Canada,
V6T 1Z4
shivanshu3@gmail.com
- Neville Grech (26)
Dept. of Informatics and
Telecommunications, University of Athens,
Greece and Dept. of Computer Science,
University of Malta, Malta
me@nevillegrech.com
- Dan Grossman (24)
Paul G. Allen School of Computer Science &
Engineering, University of Washington, USA
djg@cs.washington.edu
- Tianxiao Gu (19)
State Key Laboratory of Novel Software
Technology, Nanjing University, Nanjing,
China
tianxiao.gu@gmail.com
- Dan Hao (6)
Key Laboratory of High Confidence Software
Technologies (Peking University), MoE,
Institute of Software, EECS, Peking
University, Beijing, 100871, China
haodan@pku.edu.cn
- Farah Hariri (8)
Department of Computer Science, University
of Illinois at Urbana-Champaign, Urbana,
USA
hariri2@illinois.edu
- Wenxiang Hu (6)
Key Laboratory of High Confidence Software
Technologies (Peking University), MoE,
Institute of Software, EECS, Peking
University, Beijing, 100871, China
huwx@pku.edu.cn
- Atsushi Igarashi (2)
Graduate School of Informatics, Kyoto
University, Kyoto, Japan
igarashi@kuis.kyoto-u.ac.jp
- Hiroaki Inoue (2)
Graduate School of Informatics, Kyoto
University, Kyoto, Japan
hinoue@fos.kuis.kyoto-u.ac.jp
- Yanyan Jiang (19)
State Key Laboratory of Novel Software
Technology, Nanjing University, Nanjing,
China
jyy@nju.edu.cn
- Kumseok Jung (18)
Electrical and Computer Engineering
Department, University of British Columbia,
2332 Main Mall, Vancouver, BC, Canada,
V6T 1Z4
kumseok@ece.ubc.ca
- Gowtham Kaki (11)
Purdue University, USA
gkaki@purdue.edu
- George Kastrinis (23, 26)
Dept. of Informatics and
Telecommunications, University of Athens,
Greece
gkastrinis@di.uoa.gr
- Sarfraz Khurshid (6)
Department of Electrical and Computer
Engineering, University of Texas at Austin,
78712, USA
khurshid@ece.utexas.edu
- Joeri De Koster (14)
Vrije Universiteit Brussel, Brussels, Belgium
jdekoste@vub.ac.be
- Stefan Krüger (10)
Paderborn University, Germany
stefan.krueger@uni-paderborn.de
- Pratap Lakshman (5)
Microsoft, India
pratapl@microsoft.com

- Wing Lam (5)
University of Illinois at Urbana-Champaign,
USA
winglam2@illinois.edu
- Paley Li (12)
Czech Technical University, Prague, Czech
Republic, and Northeastern University,
Boston, MA, USA
- Sihan Li (8)
Department of Computer Science, University
of Illinois at Urbana-Champaign, Urbana,
USA
sihanli2@illinois.edu
- Fengyun Liu (3)
École Polytechnique Fédérale de Lausanne,
Lausanne, Switzerland
fengyun.liu@epfl.ch
<https://orcid.org/0000-0001-7949-4303>
- Jian Lu (19)
State Key Laboratory of Novel Software
Technology, Nanjing University, Nanjing,
China
lj@nju.edu.cn
- Xiaoxing Ma (19)
State Key Laboratory of Novel Software
Technology, Nanjing University, Nanjing,
China
x xm@nju.edu.cn
- Peyman Mahdian (5)
University of Illinois at Urbana-Champaign,
USA
mahdian2@illinois.edu
- Wolfgang De Meuter (14)
Vrije Universiteit Brussel, Brussels, Belgium
wdmeuter@vub.ac.be
- Mira Mezini (1, 10)
Technische Universität Darmstadt, Germany
mezini@cs.tu-darmstadt.de
- Gianluca Mezzetti (7)
Aarhus University, Denmark
mezzetti@gmail.com
- Ana Milanova (25)
Dept. of Computer Science, Rensselaer
Polytechnic Institute, 110 8th Street, Troy
NY, USA
milanova@cs.rpi.edu
- Ragnar Mogk (1)
Technische Universität Darmstadt, Germany
- Anders Møller (7)
Aarhus University, Denmark
amoeller@cs.au.dk
- Yeoul Na (16)
University of California, Irvine
yeouln@uci.edu
- Francesco Zappa Nardelli (12)
Inria, Paris, France, and Northeastern
University, Boston, MA, USA
- Gian Ntzik (4)
Imperial College London & Amadeus, UK
gian.ntzik@amadeus.com
- Bruno C. d. S. Oliveira (9, 20, 22)
The University of Hong Kong, Hong Kong,
China
bruno@cs.hku.hk
- Nathalie Oostvogels (14)
Vrije Universiteit Brussel, Brussels, Belgium
noostvog@vub.ac.be
- Luca Padovani (15)
Università di Torino, Dipartimento di
Informatica, Torino, Italy
luca.padovani@unito.it
<https://orcid.org/0000-0001-9097-1297>
- Karthik Pattabiraman (18)
Electrical and Computer Engineering
Department, University of British Columbia,
2332 Main Mall, Vancouver, BC, Canada,
V6T 1Z4
karthikp@ece.ubc.ca
- Luís Pina (17)
George Mason University, Fairfax, VA, USA
lpina2@gmu.edu
<https://orcid.org/0000-0003-4585-5259>
- Pedro da Rocha Pinto (4)
Imperial College London, UK
pmd09@doc.ic.ac.uk
- Aleksandar Prokopec (3)
Oracle Labs, Zürich, Switzerland
aleksandar.prokopec@gmail.com
<https://orcid.org/0000-0003-0260-2729>

Mohaned Qunaibit (16)
University of California, Irvine
m.qunaibit@uci.edu
<https://orcid.org/0000-0001-6759-7890>

G. Ramalingam (11)
Microsoft Research, India
grama@microsoft.com

Armin Rezaiean-Asel (18)
Electrical and Computer Engineering
Department, University of British Columbia,
2332 Main Mall, Vancouver, BC, Canada,
V6T 1Z4
armin.rezaiean.asel@gmail.com

Guido Salvaneschi (1)
Technische Universität Darmstadt, Germany

Tom Schrijvers (22)
KU Leuven, Belgium
tom.schrijvers@cs.kuleuven.be

Marco Servetto (20)
Victoria University of Wellington, New
Zealand
marco.servetto@ecs.vuw.ac.nz

Yannis Smaragdakis (23, 26)
Dept. of Informatics and
Telecommunications, University of Athens,
Greece
yannis@smaragd.org

Johannes Späth (10)
Fraunhofer IEM
johannes.spaeth@iem.fraunhofer.de

Siwakorn Srisakaokul (5)
University of Illinois at Urbana-Champaign,
USA
srisaka2@illinois.edu

Julian Sutherland (4)
Imperial College London, UK
jhs110@doc.ic.ac.uk

John Toman (24)
Paul G. Allen School of Computer Science &
Engineering, University of Washington, USA
jtoman@cs.washington.edu

Martin Toldam Torp (7)
Aarhus University, Denmark
torp@cs.au.dk

Vasco T. Vasconcelos (13)
LASIGE, Faculdade de Ciências,
Universidade de Lisboa, Portugal
vv@di.fc.ul.pt
<https://orcid.org/0000-0002-9539-8861>

Jan Vitek (12)
Czech Technical University, Prague, Czech
Republic, and Northeastern University,
Boston, MA, USA

Stijn Volekaert (16)
University of California, Irvine
stijnv@uci.edu

Yanlin Wang (20)
The University of Hong Kong, China
ylwang@cs.hku.hk

Tao Xie (5)
University of Illinois at Urbana-Champaign,
USA
taoxie@illinois.edu

Chang Xu (19)
State Key Laboratory of Novel Software
Technology, Nanjing University, Nanjing,
China
changxu@nju.edu.cn

Haoyuan Zhang (20)
The University of Hong Kong, China
hyzhang@cs.hku.hk

Lingming Zhang (6)
Department of Computer Science, University
of Texas at Dallas, 75080, USA
lingming.zhang@utdallas.edu

Lu Zhang (6)
Key Laboratory of High Confidence Software
Technologies (Peking University), MoE,
Institute of Software, EECS, Peking
University, Beijing, 100871, China
zhanglu@pku.edu.cn

Elena Zucca (21)
DIBRIS, University of Genova, Italy
elena.zucca@unige.it
<https://orcid.org/0000-0002-6833-6470>

Fault-tolerant Distributed Reactive Programming

Ragnar Mogk

Technische Universität Darmstadt, Germany

Lars Baumgärtner

Philipps-Universität Marburg, Germany

Guido Salvaneschi

Technische Universität Darmstadt, Germany

Bernd Freisleben

Philipps-Universität Marburg, Germany

Mira Mezini

Technische Universität Darmstadt, Germany

Abstract

In this paper, we present a holistic approach to provide fault tolerance for distributed reactive programming. Our solution automatically stores and recovers program state to handle crashes, automatically updates and shares distributed parts of the state to provide eventual consistency, and handles errors in a fine-grained manner to allow precise manual control when necessary. By making use of the reactive programming paradigm, we provide these mechanisms without changing the behavior of existing programs and with reasonable performance, as indicated by our experimental evaluation.

2012 ACM Subject Classification Software and its engineering → Software fault tolerance, Software and its engineering → Data flow languages

Keywords and phrases reactive programming, distributed systems, CRDTs, snapshots, restoration, error handling, fault tolerance

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2018.1

Funding This work is supported by the European Research Council (ERC, Advanced Grant No. 321217), by the German Research Foundation (DFG, SFB 1053 and SA 2918/2-1), and by the LOEWE initiative in Hessen, Germany (HMWK, NICER).

Acknowledgements We thank all contributors of REScala and related projects, Julian Haas for his contributions on CRDTs, and all reviewers of this paper for their comments and suggestions.

1 Introduction

Ubiquitous connectivity together with web, mobile, and Internet of Things (IoT) computing platforms require software developers to consider distributed execution as an integral part of reactive applications. In a distributed reactive application, multiple connected devices update their state and behavior in response to the flow of events and data. Examples include notifications and messaging (instant messengers, chats), activity streams (social networks), data visualization applications (e.g., Jupyter), multi-user collaborative applications (e.g., Google Docs, Microsoft Office), and multi-player online games.



© Ragnar Mogk, Lars Baumgärtner, Guido Salvaneschi, Bernd Freisleben, and Mira Mezini; licensed under Creative Commons License CC-BY

32nd European Conference on Object-Oriented Programming (ECOOP 2018).

Editor: Todd Millstein; Article No. 1; pp. 1:1–1:26

Leibniz International Proceedings in Informatics



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

Developing such applications is challenging. Their inverted control flow is typically modeled in some form of continuation-passing style, resulting in the so-called callback hell [17]. Designs using continuation-passing style are fragile, hard to maintain, and hard to reason about. In particular, callback-based communication makes handling of exceptional conditions during the execution of an application challenging [39]. These issues become even more apparent in a distributed setting, where the control flow is spread across several networked nodes and faults occur due to lost connections or shutdowns of remote devices. For example, a mobile device may get disconnected during a network request (e.g., due to a weak cellular link or mobility) and may have to eventually shut down due to excessive battery consumption while trying to reconnect.

State of the art frameworks offering automated fault tolerance (e.g., Spark [49], Flink [6]) are designed for applications that process data without user interaction and that are deployed on cluster architectures, which are easier to control than mobile wireless systems with intermittent connectivity. Approaches for building reactive distributed applications (e.g., actors [24]) cannot provide restoration or synchronization automatically, because they do not have knowledge of the overall dataflow in the application. Finally, reactive programming languages [13, 14, 18, 8], for designing reactive applications in a declarative, modular, and composable manner [43, 42], do not support (automated) handling of networking or application faults.

In this paper, we extend REScala [44] to create a fault-tolerant reactive programming language for developing distributed reactive applications. Our extensions retain the syntax and functionality of REScala for local devices. REScala has first-class abstractions for *events* and *signals*, collectively called *reactives*. Events produce distinct occurrences of values, e.g., an event corresponding to an input field produces the text a user submits. Events can be derived from each other using operations such as filters or transformations, and they can be aggregated into signals. Signals represent time-changing values, such as the latest text a user submitted. Signals resemble spreadsheet cells where the value of a cell is derived from the values of other cells and a change causes updates of all derived values. These abstractions enable developers to program reactive applications without inversion of control. Reactives and their derivations form a dynamic *dataflow graph* with nodes corresponding to reactives and edges corresponding to the dataflow between reactives. The dataflow graph has been used to automate coordination of message propagation between multiple devices [15, 28, 47]. However, none of the existing approaches provide fault tolerance.

REScala enhances the traditional dataflow graph to support recovery after crashes and adds a distribution mechanism to cope with unreliable network connections, thus simplifying the development of fault-tolerant distributed reactive applications. By combining the declarative dataflow style of reactive programming with structured techniques for eventually consistent replication [10, 45, 22] and snapshots [6], REScala provides application-wide fault tolerance with little overhead in terms of both performance and syntactic clutter. Furthermore, REScala provides language abstractions for propagating and handling errors at the application level to enable developers to handle faults when the default behavior of REScala is undesirable and to seamlessly integrate application-level fault handling into the dataflow graph.

Contributions. We make two high-level contributions. First, we use features of reactive programming to generalize existing techniques for automated fault handling to work for distributed reactive applications. Second, we extend distributed reactive programming to enable declarative fault handling. In detail, we make the following contributions:

- We extend the update propagation mechanism of reactive programming to support recovery of managed application state by consistently restarting an application after a crash as if the crash never occurred (Section 3). The extension is transparent to the application and produces little overhead in the common case (without crashes).
- We integrate eventually consistent data types with reactive programming to cope with distributed state management across devices in the presence of faults (Section 4). Our approach maintains strict consistency on a single device, but uses eventual consistency for every distributed path in the dataflow graph to guarantee availability [21].
- We design language abstractions for error propagation and adapt the runtime semantics correspondingly (Section 5), thus enabling developers to programmatically handle faults when the default behavior of REScala is undesirable, and to seamlessly integrate application-level fault handling into the dataflow graph.
- We provide an implementation of the fault-tolerant runtime and its error propagation abstractions (Section 6).
- We provide empirical evidence that REScala guarantees eventual and crash consistency in an efficient and transparent way (Section 7). To this end, we evaluate REScala using case studies to analyze the programming interface, and using microbenchmarks to evaluate the performance behavior.

The core sections mentioned above are complemented by a high-level presentation of REScala in Section 2, including an overview of the addressed kinds of faults, and a discussion of related work in Section 8. Section 9 concludes the paper and outlines areas for future work.

2 REScala from the User Perspective

In this section, we introduce REScala from the point of view of a programmer developing a simplified shared calendar application. The application tolerates disconnects and crashes, and users can update their calendar even when they are disconnected. We first discuss the fault model in detail, and then we introduce REScala by implementing the calendar application.

2.1 Faults

We use the term *fault* to refer to the origin of a failure and *error* to refer to the representation of a fault in the language [25]. REScala tolerates crashes and disconnects. REScala does not address data corruption (malicious or accidental).

Crashes happen when a device hosting part of the application runs out of battery, reboots after a crash or update, or runs out of memory, resulting in the OS to terminate the application. In these cases, the state of the application must be restored – on the same device – after a crash. Permanent faults are not addressed, because there are no spare devices or connections in the scenario we consider, i.e., we cannot equip users with new mobile phones.

Disconnects between devices are due to crashes of remote devices or due to broken network links. Disconnects cause messages to get lost, resulting in an inconsistent state across devices. REScala addresses the case where faulty devices recover after a crash and broken links are eventually restored – otherwise, state on the disconnected device is lost.

Current reactive programming approaches [15, 28, 47, 27, 31, 14] do not provide mechanisms for any kind of fault tolerance and delegate the responsibility for handling errors to the language into which the reactive framework is embedded – sidestepping the issue. In contrast, REScala provides tolerance of the above faults in the reactive programming paradigm in an automated manner.

1:4 Fault-tolerant Distributed Reactive Programming

```
1 val newEntry = Evt[Entry]()
2 val automaticEntries: Event[Entry] = App.nationalHolidays()
3 val allEntries = newEntry || automaticEntries

5 val selectedDay: Var[Date] = Var(Date.today)
6 val selectedWeek = Signal { Week.of(selectedDay.value) }

8 val entrySet: Signal[Set[Entry]] =
9   if (distribute) ReplicatedSet("SharedEntries").collect(allEntries)
10  else allEntries.fold(Set.empty) { (entries, entry) => entries + entry}

12 case class Entry(title: Signal[String], date: Signal[Date])

14 val selectedEntries = Signal {
15   entrySet.value.filter { entry =>
16     try selectedWeek.value == Week.of(entry.date.value)
17     catch { case DisconnectedSignal => false }
18   }
19 }

21 allEntries.observe(Log.appendEntry)
22 selectedEntries.observe(
23   onValue = Ui.displayEntryList,
24   onError = Ui.displayError)
```

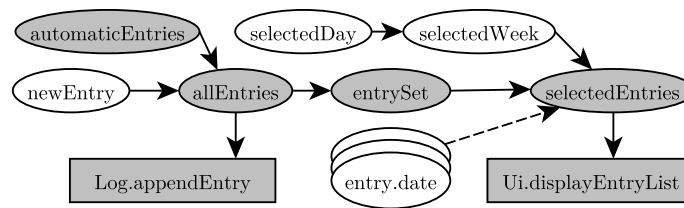
■ **Figure 1** Excerpt of REScala source code for the shared calendar application.

2.2 Shared Calendar Application in REScala

A user of the shared calendar application can create new calendar entries and select the displayed week. The calendar will be synchronized with other users when a connection is available.

Figure 1 shows our implementation. We refer to it as we introduce REScala’s events, signals, conversions between events and signals, and how they are relevant for fault tolerance. The dataflow graph of the application is depicted in Figure 2, where the node labels correspond to identifiers used for reactives in the code example and edges represent the dataflow between those. The highlighted part shows all reactives reachable from `automaticEntries` and to which changes are automatically propagated. The rest of this section describes how this graph is created and how it behaves.

Events. Distinct occurrences of values are produced by events in REScala. There are input events and derived events. Input events are denoted by the keyword `Evt` (cf. Line 1) and allow to emit values using `evt.fire(value)`. The imperative firing of events typically happens as part of an integration with some external event source, such as an imperative UI library where a button press triggers a callback that fires the event, and is thus not shown in the code example. Derived events are defined by filtering or aggregating other events on which they depend. For example, the `||` operator combines two events into a new event that emits all values emitted by either of its dependencies, e.g., `allEntries` (Line 3) emits entries whenever generated by the user or from some external data source, such as a stream of national holidays.



■ **Figure 2** Dataflow graph for the calendar application. Nodes reachable from `automaticEntries` are highlighted.

Signals. Values that change over time are represented as signals in REScala. There are input signals and derived signals. Input signals are denoted by `Var`. In Line 5, an input signal is used to represent the currently selected day. The value of a `Var` can be changed by imperative code, e.g., `selectedDay.set(Date.tomorrow)`. The `Signal` keyword uses a user-defined computation – the *signal expression* – to express a derived signal. The signal expression can access other signals, called its *dependencies*. Accessing dependencies is explicit – using the `value` method to syntactically mark accessed dependencies. In Line 6 of our example, the `selectedWeek` signal is derived from `selectedDay.value`. The current value of a derived signal is updated automatically by executing the given computation whenever its dependencies change, similar to a formula in spreadsheets. Changes of a signal are automatically propagated to *derived* signals that use the signal in their definitions.

Crash-tolerant signals. Signals hold state and are restored after a crash (c.f., Section 3), either by loading the value from a snapshot, or by recomputation. For example, a user may have selected a different day than today, thus the selected day has to be stored in persistent storage. On the other hand, the selected week is recomputed from the selected day. REScala uses the Scala type system to statically ensure that the values of signals that are included in snapshots are serializable. For example, the `Var[Date]` in Line 5 requires that `Date` is serializable. Explicit annotations by the programmer are not required due to type inference and implicit parameters.

Folds and replication. Signals are convertible to events by aggregating individual event occurrences into an updating signal value – similar to folding over (infinite) lists. We refer to such signals as *fold signals* or simply *folds*. A fold, such as in Line 10, creates a signal with an initial value and updates it according to the parameter function every time the event fires. Lines 8 to 10 define a list of all calendar entries as the signal `entrySet` by folding over emitted calendar entry events. When the flag `distribute` is false, in Line 10, the `fold` operator aggregates the calendar entries emitted by the `allEntries` event into a list of all calendar entries. Line 9 demonstrates a distributed aggregation that has the same behavior as the local `fold`, aggregating all entries into a set. However, a `ReplicatedSet` has a name, `SharedEntries` in this case, and elements of the replicated set are shared with other devices that also use a `ReplicatedSet` with the same name.

Fault-tolerant folds. Fold signals are particularly interesting for fault tolerance, since they aggregate state that (a) must be included in a snapshot for restoration after a crash, and (b) is reliably replicated to other devices. Additionally, a distributed aggregation such as a `ReplicatedSet` is also synchronized after a crash to ensure that all devices eventually see the same set of entries. REScala uses built-in data types based on CRDTs [45] to ensure that

1:6 Fault-tolerant Distributed Reactive Programming

changes to all replicas eventually become consistent in the presence of crashes and message losses. The order in which entries are added, however, may be different on every device, and the intermediate values are visible.

Dynamic dependencies. Until now, all dependencies have been static, i.e., the dataflow does not change during runtime. However, signals may have a *dynamic* set of dependencies to support *higher-order* signals, i.e., signals whose values contain *inner* signals. For example, the entries (Line 12) we have been using in the calendar have a title and a date, both of which are signals and may change their value. The `selectedEntries` (Line 14) is derived by filtering the `entrySet` (Line 15). The filter function dynamically accesses the inner `date` signal of each entry (Line 16).

Interactions with the environment. In addition to firing and setting inputs, events and signals can be *observed* to produce side effects. Observing an event executes the side-effecting handler function every time the event emits a value, e.g., each entry is appended to a log file in Line 21. Observing a signal bridges between time-changing values and ordinary imperative state. For example, in (Line 22) the current state of the UI is overwritten when it becomes inconsistent with the signal after a signal change. REScala guarantees that the handler function of a signal observer is always called with the most recent value of the signal after an update, which allows the application to extend its invariants from the dataflow graph to external imperative libraries (in this case, the invariant is that the UI always displays the values held by the `selectedEntries` signal). Both event and signal observers can take an additional parameter to observe errors (Line 24), as explained next.

Explicit error handling. Reactives in REScala propagate errors along the dataflow graph. Errors can be handled as exceptions in signal expressions. For example, in Line 16, if the network connection fails before an inner date signal is transferred for the first time, then the access to the unavailable entry date signal throws a `DisconnectedSignal` exception. The default error handler postpones further evaluation of the `selectedEntries` until the signal is available. Instead of the default behavior, Line 17 explicitly catches the exception and returns false, causing the filter to drop the entry. Explicit error handling enables the use of application specific knowledge for more precise control of application behavior.

3 Fault-tolerant Application State

As illustrated in Section 2, crashes of individual devices during the execution of a distributed reactive application may result in a loss of the state of the reactive subgraph hosted on these devices. Loss of local device data is problematic since such data often contain important private or unsynchronized information of the current user. To address this issue, REScala provides automatic snapshots and recovery.

Snapshot anatomy. Conceptually, a snapshot of an application is a function that maps unique keys, denoting reactives, to their current values. The REScala runtime performs an analysis of the dataflow graph to minimize the number of key-value pairs that need to be stored. Applications often store redundant derived state in memory for efficiency. For example, a histogram displayed to the user can be recomputed from database entries, but it would be expensive to repeat this process for every frame the application displays. Local REScala applications typically consist of many small derived parts of the state (i.e., single

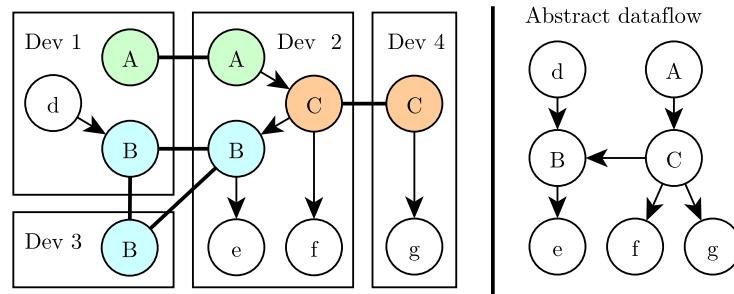
reactives) to take advantage of incremental updates. In such a setting, REScala detects derived state and excludes it from snapshots. Precisely, in REScala, the only reactives that cannot be derived are *vars* and *fold signals* (signals that aggregate event occurrences), since their state depends on past user interactions. All other reactives are either stateless events or derived signals that can re-execute their user-defined expression to recompute their state. We say that vars and fold signals constitute the *essential state*, and REScala recovers the state of the dataflow graph from the essential state.

Creating snapshots. Snapshots are created between semantically related changes, such that ongoing control flow does not interfere. REScala can detect related changes, because the dataflow graph makes semantic relations in applications explicit as (transitive) edges between reactives. For example, in Section 2, creating a new entry (a) updates the list of entries, (b) updates the list of selected entries, and (c) causes the UI to refresh. Explicit relations are used to determine when the transitive changes caused by an input event are fully applied to all reachable reactives and a snapshot is created after one such *update* to the dataflow graph. As a result, each snapshot corresponds to a single user perceived update, e.g., all changes of the state triggered by a new calendar entry belong to the same snapshot.

Incremental snapshots. Storing a full snapshot of all the essential state after each update is wasteful, since an update only affects parts of the application. REScala knows all updated reactives – they constitute the transitive closure of changed inputs, e.g., the highlighted reactives in Figure 2, starting with `automaticEntries`. REScala stores snapshots incrementally, by only changing the values in the snapshot that correspond to updated reactives. As a result, the cost of creating snapshots only grows linearly with the size of updates, instead of linearly with the size of the application. This scaling behavior supports efficiently composing large applications out of multiple parts, such as network, UI, and background services, in case the added parts do not increase the size of updates.

Recovering state. For recovery, REScala re-executes the application to restore the dataflow graph. During this recovery process, the value of each reactive is restored to the state before the crash. Events do not have state, so no value is restored. Fold signals and vars recover their values directly from the snapshot. Derived signals recompute their values from their inputs. The acyclic dataflow graph ensures that inputs are restored before derived signals, hence they can be used to recompute the derived signals. Like snapshot creation, the recovery process is incremental as reactives are restored as soon as they are created during the re-execution of the application. Thus, REScala allows the restored parts of the application to already handle new interactions, while other parts are still recovering.

Observers. REScala only restores state that is part of the dataflow graph. To ease integration with external libraries, REScala executes observers on signals during restoration. For example, when the list of selected entries of the calendar is restored, the observer that informs the UI about updates is executed. Observers allow the application to implement an invariant between the state in the dataflow graph and external state. Executing the observers during recovery allows the application to uphold its invariants, i.e., that the imperative state that is modified by the handler corresponds to the latest value of the signal. However, it is ultimately the responsibility of the application to use correct handlers. Events have no state to be restored (because snapshots are only stored between updates), so the handlers on event observers are not executed during restoration.



■ **Figure 3** Full dataflow graph of a distributed application (left) and abstract dataflow (right).

Recomputation versus full snapshots. We make two arguments why recomputation is preferable over storing more values in the snapshot. First, a snapshot is created every time an update occurs, while restoration only happens when a device fails. Hence, storing only necessary state has performance benefits if the latter is only a small portion of the overall state (cf. Section 7.2 for an empirical evaluation). In a previous study [44], we reported that in a typical reactive application only 14% of the dataflow graph contains essential state. Second, only the essential parts of the state need to be serializable, thus allowing the use of data types that cannot be (efficiently) serialized for the rest of the application. REScala uses the type system together with type inference and implicit parameters to ensure that the static type of each reactive containing essential state is serializable.

4 Managing Distributed State

This section presents how REScala keeps the application responsive when network connections are not reliable. The key idea is to make the dataflow graph fault-tolerant and eventually consistent, instead of handling fault tolerance at the level of individual messages.

Replicated signals. Fault-tolerant dataflow graphs use *replicated signals* to model shared state among multiple devices. For illustration, consider Figure 3, which shows a dataflow graph spanning four devices (left), and the dataflow without distribution that is represented by this graph (right). Reactives A, B, C are replicated signals representing data shared across the devices (C is replicated in Dev 2 and Dev 4, A in Dev 1 and Dev 2, and B in Dev 1, Dev 2, Dev 3).

Replicated signals behave as normal signals with regard to device-local dataflow, e.g., their state is stored in the local snapshot. However, unlike directed dataflow connections between reactives on the same device, connections between replicas work in any direction, i.e., each device can change the state of its replica independently, even while being disconnected from the rest. As a result, the state of the replicas of a replicated signal can diverge in the presence of disconnects. Replicated signals use eventually consistent synchronization to support fault tolerance, i.e., the state of replicas diverges when disconnected to remain responsive and eventually converges when connections are available. Replicated signals are implemented using state-based conflict-free replicated data types (CRDTs) [45]. CRDTs provide automatic conflict-free merging of diverged state for a wide range of common data types [45]. For instance, the example in Figure 1 illustrates the usage of a replicated reactive of type `ReplicatedSet`:

```
9 ReplicatedSet("SharedEntries").collect(allEntries)
```


The underlying CRDT of `ReplicatedSet` is a set with a single commutative, associative, and idempotent operation, which adds to the set each value associated with an occurrence of the collected event. The set of entries is synchronized between all devices that use a replicated signal with the same name and type. Due to the properties above, the state of replicas can always be synchronized, and eventually all devices will converge to the same set containing all added elements. In addition to `ReplicatedSet`, `REScala` currently supports replicated counters, last-writer-wins registers, ordered lists, and replicated data types that allow adding and removing elements from sets and lists. By using conflict-free data types – an existing technique already known to programmers – we provide simple and intuitive semantics for sharing state across devices.

Integration with the overall dataflow graph. The well-defined set of operations on CRDTs enables their integration into the update propagation and enables state restoration, because operations of each CRDT, and how it changes based on local and remote events, are visible to `REScala`. As a result, changes to the state of a replicated signal are immediately propagated to local derived signals in the same manner as local changes update the whole reachable part of the dataflow graph at the same time, i.e., each update either is visible by all reactives of the local device or not visible at all. In contrast, the state of reactives on different devices can temporarily diverge. For example, consider the graph in Figure 3. An update that affects reactive A on Dev 2 will immediately affect reactive B on Dev 2 because they are connected by the local dataflow graph. However, if A is updated on Dev 1, B is only indirectly affected and synchronization with Dev 2 is required to complete the update. Such an inconsistency, where an update is applied to A but not yet to a connected reactive B, is called a glitch. To prohibit distributed glitches, Dev 1 would have to wait for the update on B to arrive, whenever A is changed. `REScala` allows distributed glitches in favor of availability.

Replicated signals are stored and restored when a device fails. Replicated signals have unique names shared between devices, which are used as keys in the snapshot, allowing multiple devices to include their replica in their snapshot and to synchronize changes after restoration. Thus, snapshots combined with conflict-free replication allow a device to disconnect, store and restore local modifications that survive crashes, and merge the local snapshot with the current state in the replicas on reconnect.

Publishing signals. Using CRDTs to implement replicated signals allows bidirectional communication, but the changes that one can perform on a signal are limited to the operations implemented by the CRDT. Alternatively, `REScala` allows to *publish* any signal – not only those based on CRDTs – but the published signal may only be changed by the publishing device. To prevent conflicting changes, other devices can only read the published signal. For example, each individual calendar entry (title and date) in the shared calendar is published by the device that puts the entry into the calendar, thus only the creator of an entry can change it. Publishing is a special case of eventually consistent replication. To publish a signal, `REScala` creates a replicated signal with a last-writer-wins CRDT, a data type where the merge function always selects the most recent value. Since only one device is allowed to write, there are no races between writes.

Distributed event propagation. Events are not distributed directly, but have to be converted to signals. However, we leave the responsibility to decide about the concrete conversion to the programmer, because of the trade-off between reliability and communication overhead involved in the conversion. For example, the `latest(n)` operator can be used to create a

signal containing the latest `n` occurrences of the event. As a result, the connection can be lost for the duration of `n` event occurrences without loss of data. If more than `n` events occur when the device is disconnected, the oldest events will be lost. Similar operators can be used to define time or priority based policies, allowing the application developer to tune the software behavior as necessary.

5 Error Propagation

REScala uses CRDTs to achieve fault-tolerant replicated state in an automatic way. However, by default, a disconnect of a CRDT does not trigger any action. The application eventually receives new updates after the reconnection and the inconsistency is resolved. In some scenarios, the application cannot simply wait for an eventual update, but has to act sooner. For this reason, developers should be able to program application-specific behavior in case of faults. For example, if a connection fails, a different replica may be selected manually, or the missing values are approximated, e.g., using a default value, if the application can continue the execution tolerating the inaccuracy. To support custom fault handling, we introduce errors as a programming abstraction in REScala. Errors are pushed into the dataflow graph when a device becomes disconnected (such a condition is established using timeouts). Errors are propagated along the same path as values in the dataflow graph, similar to how exceptions propagate along the path of the values returned by function calls.

In the following, we describe the API of our error propagation and handling mechanism from a user's point of view, and show how to handle errors that occur due to faults in the underlying distributed system as well as local errors due to faults like missing files or exceptions in external libraries. The new error-aware semantics of the reactivities is a superset of their original semantics, thus existing code carries over unchanged.

Injecting errors into the dataflow graph. We extend the API of `Evt` and `Var` with operations for firing errors. `Evt.admit(error)` behaves similar to the existing `Evt.fire(value)` (similar for `Var.set`), but it starts the propagation with an error instead of a value. The main use of this API is to support the integration of existing frameworks, e.g., converting an error of a networking library to an error in the dataflow graph. Consider an existing networking library with a callback-based API. When a timeout occurs in the network, the imperative library callback is converted into a reactive propagation:

```
val fromNetwork = Evt[NetworkMessages]()
Network.onTimeout { error => fromNetwork.admit(error) }
```

Observe and recover. We extend the observer's API to accept an additional handler parameter called `onError`, which is used to observe propagated errors. This handler has the same purpose as `catch` blocks, and, similar to the standard `observe` call, has the goal of producing a side effect, e.g., displaying an error message. The error handler on observers can be missing: any unhandled error terminates the program in the same way as traditional uncaught exceptions. In the calendar example in Figure 1, any error is displayed to the user by the error handler defined on the signal in Line 22 using the extended observer API:

```
22 selectedEntries.observe(
23   onValue = Ui.displayEntryList,
24   onError = Ui.displayError)
```

Instead of simply observing errors, the application developer can recover the error inside the dataflow graph using the `recover` operator for signals and events, which is parameterized with a recovery function that converts an error to a normal value. The value is then propagated as the output of the `recover` operator. Any normal value that flows through the `recover` operator in the dataflow graph is propagated without change. The `recover` operator handles errors while they are propagated through the dataflow graph and before they reach an observer. Recovering from an error is most useful, when errors can be locally converted back into normal values. This case is relevant in several applications. For example, values can have local fallbacks, such as an unavailable location service that can be replaced by using more expensive or inaccurate local data. Another example is a signal holding an UI widget, where an error can be handled by displaying it to the user.

Signal expressions. A user-defined computation in a signal expression may access any number of dependencies. When any of the dependencies propagates an error, the error is raised as a Scala exception by the `.value` call performing the access. The exception may be handled by the application using the default Scala exception handling mechanisms. Unhandled exceptions in a user-defined computation are propagated along the dataflow graph. The use of Scala exceptions enables our error handling scheme to integrate well with most libraries in the JVM. For example, the shared calendar in Figure 1 filters the list of all calendar entries to only include entries of the current week in Line 14, and all entries containing an error are removed using a Scala `try/catch` block:

```
14 val selectedEntries = Signal {  
15   entrySet.value.filter { entry =>  
16     try selectedWeek.value == Week.of(entry.date.value)  
17     catch { case e: NetworkError => false }  
18   }  
19 }
```

When the `entry.date()` in Line 17 contains an error, the error is thrown as a Scala exception and handled in the `catch` by returning `false`, causing the filter to drop the entry.

Folds. Recall that the `fold` operation supports converting events into signals. Given an event `e`, an initial value `init` and a function `f`, which are passed to it as parameters, `fold` returns a signal that is initialized with `init` and gets updated every time `e` fires by applying `f` to the current value of the signal. Thus, unlike other derived reactives, a `fold` signal accesses its own current value, i.e., the fold (indirectly) depends on the complete history of the event. In our example, the signal `allEntries` (Line 10) is constructed by reading the list of all entries and appending each read entry to create a value that accumulates all event occurrences:

```
10 allEntries.fold(Set.empty) { (entries, entry) => entries + entry }
```

The current accumulated value of the fold is treated like any ordinary dependency. If it is accessed and it holds an error, the error is thrown as a Scala exception. If the exception is not handled inside the user-defined computation (i.e., the function body of `fold`), then an error is propagated by the `fold` reactive to other reactives that depend on it. On the other hand, by handling the exception a developer can resume the computation of the `fold` reactive after an error. We present an example for fold with custom error handling next.

Example: Fold with custom error handling. To illustrate the use of the error handling API in fault-tolerant REScala, in the following we present and discuss the implementation of a user-defined operator on events. The operator, called `count`, is defined in terms of `fold`. It counts the number of non-error event occurrences and forwards error event occurrences without increasing the count. The implementation of `count` is shown below:

```

1 def count() = fold(0) { (state, occurrence) =>
2   occurrence // access the (unused) value to propagate potential errors
3   try state + 1 // increase count in non-error case
4   catch { case (value, error) => value + 1 } // continue counting after errors
5 }
```

The `count` signal starts with its initial state initialized to zero (Line 1). The folding function takes the current `state` of the fold and the incoming event, called `occurrence`, as parameters. When `occurrence` is accessed in Line 2, there are two possibilities: the access raises an error or a normal value. (a) in case of an error, the execution of the user-defined computation is aborted (because the access is not enclosed in a `try/catch` block), the state of the fold is not increased and the error is stored in the `fold` for future processing. (b) If the access of `occurrence` returns a normal value, the latter is ignored (we only count the number of non-error occurrences) and the execution attempts to access `state` in Line 3. If the current `state` is a normal value, it is incremented, and the increased count is returned (Line 3). If the current `state` is an error, the latter is thrown when `state` is accessed and immediately caught in Line 4. The pattern match in the `catch` block binds the last non-error value stored in the fold and the current error. Our example handler ignores the error and continues by incrementing the last non-error state, thus implementing a `counter` that resumes counting when a new occurrence arrives after an error.

6 Implementation

Our fault tolerance mechanisms are an extension of REScala, which in turn is implemented as a Scala library. We added efficient support for fault tolerance while preserving compatibility with existing applications.

Distribution. REScala uses a custom message passing mechanism for distribution based on TCP and Websockets (for Web clients); it does not provide any specific mechanism for peer discovery – the latter has to be implemented by the application. The synchronization mechanism of REScala supports any topology, e.g., client-server or peer to peer.

The mechanism for detecting changes of replicated signals is the same as the one used for local propagation of updates, i.e., a change is detected when the value of a signal is replaced by a new one. Change detection relies on immutability of values in signals, i.e., changes via side effects are not detected. When a change is detected, the new value of the replicated signal is serialized and sent over the network. Values are serialized using Circe¹, which supports type-safe serialization for most built in immutable Scala data types. Custom serializers can be provided using typeclasses. The serializer for signals is special and causes the signal to be published as described in Section 4.

¹ <http://circe.io/>

Snapshots. Our snapshot and restoration mechanism supports storing snapshots in arbitrary key-value stores. We have two implementations for in-memory stores: one that writes directly to disk (for JVM and Android) and another one that uses HTML5 `localStorage` [3] for web browsers.

Using fault-tolerant reactivities is thread-safe, but care must be taken when reactivities are created in a multi-threaded environment. Snapshots in REScala require unique IDs to identify reactivities, and our current implementation uses thread-local counters to generate IDs, e.g., the IDs `UI-0` and `UI-1` are associated to the first and to the second reactive created in the context of the UI thread. This ID generation strategy is well suited for applications with a fixed set of threads, but it cannot cope with the case in which threads are created and used dynamically in a thread pool, because the IDs generated for those threads do not remain the same between restarts of the application. REScala requires the developer to explicitly handle the assignment of IDs to reactivities if the automatic mechanism is insufficient. In practice, it is in most cases sufficient to ensure that dynamically scheduled tasks (e.g., those in thread pools) are assigned deterministic names to enable correct automatic generation of unique IDs.

Errors. The error propagation mechanism is integrated into the implementation of the dataflow graph through the following extensions. First, we extend the types of the values held by reactivities. The REScala implementation (without support for errors) distinguishes between `Changed[T]` and `Unchanged[T]` for the data type of the values held by reactivities – these two different types of values are propagated differently. To support error propagation, we introduce a third type, `Error`, and update the case distinctions in the propagation logic, whenever any of the types is accessed. Second, we modify the reevaluation function such that any exception thrown during the execution of user-defined computations, is propagated as an `Error`. Overall, our implementation strategy for errors induces little performance overhead when no faults are present, as shown in the empirical evaluation in Section 7.2.

7 Evaluation

Our integration of fault tolerance mechanisms into the reactive language runtime comes with synergetic effects between the two. On the one hand, snapshots and restoration maintain the consistency guarantees of reactive programming on individual devices in the presence of faults, distributed signals bridge dataflow graph across devices, and our error propagation mechanism enables principled handling of exceptional cases. On the other hand, the dataflow graph is instrumental in enabling fault tolerance in distributed applications at little cost in terms of both the burden on the programmer and the performance overhead. Specifically, the language runtime ensures that (a) the graph is consistent between updates, providing a point in the execution where snapshots can be taken efficiently, (b) derived values are automatically and consistently recomputed during restoration and remote updates, and (c) the application cannot change the state arbitrarily, so snapshots always remain consistent with the current state and changes are detected and distributed.

In the following, we empirically evaluate the claim that our fault tolerance features come at little cost in terms of both the burden on the programmer and the performance overhead.

Case study	observe		fire		change		Total		Description
	ok	nok	ok	nok	ok	nok	ok	nok	
CRDTs			9				9		CRDT integration
Datastructures			5				5		Reactive collections
Dividi			1		3		4		P2P distributed ledger
Editor			42		10	1	52	1	Swing text editor
Examples			39	2	9	19	48	21	Swing/console examples
Mill game			14		7	1	21	1	Turn based swing UI
Pong game	3		15	5	4	5	22	10	Multiplayer swing game
Reactive streams			1				1		Interface integration
Scalafx	3		1				4		JavaFX integration
Scalatags	2				1		3		HTML DOM integration
Swing			2			2	2	2	Swing integration
RSS			15		4		19		Swing RSS reader
Shapes	1		17		4	1	22	1	Swing drawing app
Todolist			11				11		TodoMVC app
Universe		1	8		2	9	10	10	Console simulation
Total	9	1	180	7	44	38	233	46	

■ **Figure 4** Possibly problematic operators in case studies and extensions.

7.1 Non-invasive Fault Tolerance

Our extensions for fault tolerance are non-invasive, meaning that existing applications implemented with REScala are made fault tolerant with minimal effort. To validate this claim, we answer the following research questions:

- (RQ1) To what extent do snapshots and restoration affect the application semantics?
- (RQ2) To what extent does the integration of replicated signals into the dataflow graph affect the application semantics?
- (RQ3) How many changes to a reactive application are necessary to support error propagation and handling?

To answer these questions, we analyze a set of case studies, consisting of ten applications (including games, simulations, and GUI applications) and five integrations with external libraries (e.g., an API to access the HTML DOM, bindings for JavaFX and for Java Flow), comprising a total of 13.000 LoC². The case studies are listed in Figure 4 and their code is publicly available³.

(RQ1) Effects of state snapshotting/restoration on application semantics. Snapshotting is invisible to an application, since snapshots are automatically created at the end of an update propagation. Restoration, on the other hand, is visible to the application, since restoration re-executes the application to restore the dataflow graph (cf. Section 3). The value of signals may differ between its first (normal) start and a restoration, causing different application behavior. Furthermore, certain inputs to the dataflow graph may be duplicated while restoring. For example, if a new calendar entry is added to the shared calendar via `newEntry.fire(startedEntry)` during startup of the application, then the `startedEntry` would

² Lines are counted with CLOC (cloc.sourceforge.net) excluding comments and blank lines

³ Repository available at www.rescala-lang.com.

be added every time the application is restored, resulting in multiple such entries in the list of `allEntries`, which is obviously not the desired behavior. We refer to problems with different behavior during restoration as *restoration inconsistency*.

To quantify the extent of restoration inconsistencies, we inspect all input and output interactions of imperative code with the dataflow graph in our case studies. These interactions are easy to localize, since they occur via a well-defined interface of the dataflow graph, consisting of the operations `fire`, `set`, and `observe`. The columns for `fire` (input interactions, also containing `set`) and `observe` (output interactions) of Figure 4 summarize our findings.

Firing events on some occurrence in the external world via the `fire` and `set` operations serves the purpose of entering new values into the dataflow graph, e.g., a user clicking a button, time passing, or receiving a network message. In our case studies, 180 out of 187 `fire` calls serve such a purpose and are not affected by state restoration. The 7 remaining calls that do exhibit the restoration inconsistency problem are instances of the same event usage anti-pattern: they incrementally build state during application startup. For example, the Pong game initializes the UI elements, and adds them one by one to a list of all UI elements, as shown below. As a result, this list would grow after each restoration.

```
val addElement = Evt[UIElement]
val allUIElements: Signal[List[UIElement]] = addElement.list()
addElement.fire(ball); addElement.fire(player1); addElement.fire ...
```

Firing of events must not be misused for initializing reactivities. Manual inspection of usages of the `fire` method is required to find such misuses.

We also analyzed if `observe` calls on signals cause inconsistencies during restoration. We found a total of 10 usages of signal observers in the case studies (event observers are more common with 150 usages). Out of those 10 signal observers, 9 are not affected by restoration inconsistencies. 7 of them are in bindings for external libraries and are used to set properties of UI toolkits, e.g., the window title as in `titleText.observe(UI.window.setTitle)`. Triggering these observers during restoration correctly causes the UI to display the restored state. Two observers execute cleanup code, which is not affected by restoration either. The only observer that is affected by restoration inconsistency is in a simulation application (`Universe` row in Figure 4). The simulation uses mutable state outside of REScala, and if a fault occurs during a simulation step, this state is not restored.

We conclude that the state snapshotting/restoration feature of our approach operates mostly transparently. This means: (a) most of the potentially problematic interactions (181 out of 189, roughly 96%) are unproblematic in our fault-tolerant runtime, (b) the few problematic cases can be avoided, if application developers use the correct APIs of the dataflow graph, and ensure that mutable state outside of REScala is also able to tolerate faults.

(RQ2) The effect of introducing eventually consistent updates. Eventually consistent updates may affect the behavior of existing applications in two ways. First, they break the invariant that each occurrence of an input `Evt` is handled individually. Instead, after devices were disconnected for a while, all changes are replicated as a single large change to other devices. These combined changes cause problems when the application expects each change individually, e.g., if our shared calendar were to display a notification each time an entry is added, the notification may be triggered for a group of entries, instead of each individual entry, and as a result, the notification system has to be able to handle multiple entries at once.

Second, they break assumptions that usages of the `change` operator on signals may make about its behavior. The `change` operator is used to reify and handle each individual change of a signal, and usages of `change` may assume that every intermediate change of the signal will occur individually. However, with eventual consistency intermediate changes may be grouped as described above, hence assumption changes become invalid. For illustration, consider a simple clock implemented as below. The computation of `minutes` relies on `seconds` change to 0. However, with eventually consistent propagation `seconds` could change from 59 to 2 skipping the intermediate step, because an aggregated update is received over the network, resulting in a missed minute.

```
val tick: Event[Unit] = ... // fires once per second
val seconds = Signal { tick.count() % 60 }
val minutes = Signal { seconds.change.filter(_ == 0).count() % 60 }
```

To quantify to which extent the introduction of replicated signals affects the application semantics due to the existence of `change` operations on signals, we investigate whether the semantics of our case studies relies on each individual signal change being visible, as opposed to relying on a notification about its latest change. The results of this analysis are shown in the `change` column of Figure 4. Roughly 46% of `change` operators (38 out of 82 in 7 out of 15 case studies) have different behavior when individual changes are grouped or skipped due to eventual consistency. The results indicate that replicated signals with eventual consistent semantics cannot be introduced transparently, which, in fact, is not surprising. One way to mitigate the problem is to keep computations that require strong consistency on a single device, and only distribute their results via replicated signals. As discussed in Section 9, manual handling of network errors has the potential to enforce consistency at the cost of availability, but this is not currently supported.

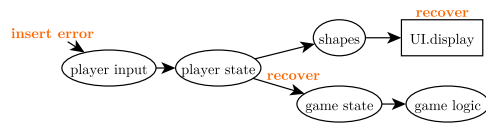
(RQ3) Changes to application code needed to propagate and handle errors. The integration of error propagation into the normal change propagation allows to propagate errors mostly transparently – additional code is required only at specific places where the developer wants to handle errors. The key point is that intermediate reactivities do not have to be updated to propagate the error, minimizing the total amount of application code that requires modification. To demonstrate that error propagation does not “pollute” application code, in the following, we discuss how we refactored one of the existing case studies – a simple two player Pong game – to add support for handling application-level errors.

The case study consists of two application windows, one for each player. Without handling faults, if one player dropped, the game would get stuck or simply terminate. Figure 5 shows an abstract representation of the dataflow graph of the case study. Altogether, we update the game at three locations out of the 250 total LoCs.

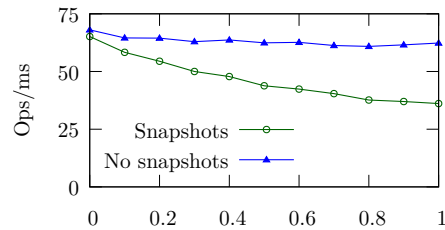
To evaluate error handling in REScala, we added functionality to allow players to leave and join the game. When a player disconnects, an error gets inserted into the position signal of the racket of that player:

```
UI.onClose{ Racket.pos.admit(PlayerDisconnected) } // set position to error
```

Following the dataflow of `Racket.pos` through the dataflow graph of the application, one can identify the places where the error needs to be handled. There are two such locations: when displaying the players on the screen and inside the game logic handling cleanup of data structures for disconnected players.



■ **Figure 5** Recovering from errors in Pong.



■ **Figure 6** The cost of snapshots.

For handling the error when displaying the players, we reused an existing try/catch block that handled missing game objects and added a handler for the `PlayerDisconnected` exception.

```

case _: NoSuchElementException | _ : PlayerDisconnected =>
  // remaining handler unchanged
  
```

As a final modification to the code, failed connections are observed and the corresponding player is removed from the game. To remove the player, a list of disconnected players is derived from the the list of players, by filtering on the player connection:

```

val disconnectedPlayers = Signal{ players.value.filter { p =>
  Try(p.connection.value).isFailure} }
disconnectedPlayers.observe(Game.removePlayers)
  
```

If accessing the connection raises an error (checked with `Try(...).isFailure`), then the player is considered disconnected. The resulting list of failed players is observed and these players are removed from the game (closing the connection and updating the list of players).

7.2 Performance Evaluation

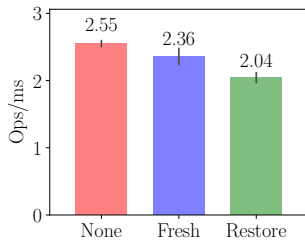
We use microbenchmarks to evaluate the performance of different features of REScala. We evaluate each feature individually, since they do not influence each other and can be disabled by applications as required. Specifically, we answer the following questions:⁴

- (RQ4) What is the performance overhead introduced by our snapshotting mechanism?
- (RQ5) What is the performance tradeoff between restoring state from the snapshot versus recomputing the state?
- (RQ6) How does the performance of our recovery mechanism compare to the performance of the recovery mechanism of an industrial-strength data streaming system?
- (RQ7) How does language-integrated error propagation affect application performance?

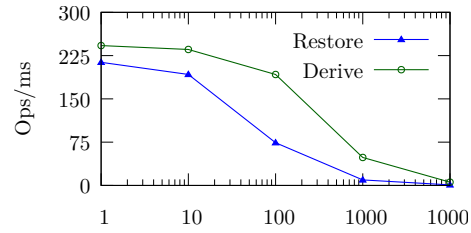
Experimental setup. We use existing microbenchmarks of the base reactive language, which are available from the Github repository⁵ in version `v0.21.1`. The benchmarks are implemented using the OpenJDK benchmarking framework Java Microbenchmark Harness [4] version 1.19. We perform 25 iterations of each benchmark and report the average. To reduce the influence of non-deterministic optimizations, we fork the JVM 5 times, each doing 5 iterations with proper warm-up. Each iteration runs for about 1 second. We run the

⁴ We do not evaluate the efficiency of our CRDT as we do not contribute performance improvements over existing work [5, 45].

⁵ See www.rescala-lang.com.



■ **Figure 7** Cost of restoration.



■ **Figure 8** Restoring vs. recomputing lists of various sizes.

benchmarks on an Intel Xeon CPU E5-2670 @ 2.60GHz, using one core only, since the benchmarks are not multi-threaded, and we use the OpenJDK 1.8.0_141 Server VM with default parameters on CentOS Linux (Kernel 3.10).

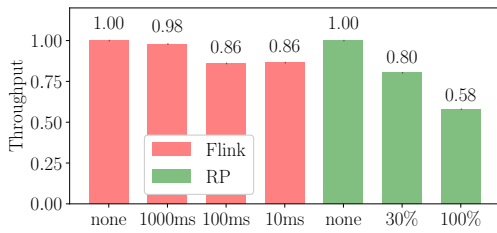
(RQ4) Overhead of snapshots. Snapshots happen after every update to the dataflow graph and affect the overall application performance. Snapshot overhead consists of the internal overhead for determining all the updated state and of the overhead for serializing that state. The snapshot is stored in an in-memory database, because we do not want to measure time spent writing to disk, since this overhead is not specific to our solution. We quantify the snapshot overhead as a function of the number of folds in an application, since only the state of fold signals is included in a snapshot. For this purpose, we parameterize our benchmarks with the number of fold signals in the graph.

Figure 6 shows the throughput for a dataflow graph consisting of a single input event with 100 reactives derived from it, on the x -axis is the percentage of folds out of these derived reactives, the other reactives are stateless. We selected this topology since it allows us to create a full snapshot of all fold reactives with a single input change. To factor out the influence of computations not involved in snapshotting, user-defined computations of both folds and stateless derived reactives only do simple integer arithmetics with negligible overhead. We executed the benchmark twice, with and without snapshots enabled. The relative throughput is on the y -axis of Figure 6 (higher is better).

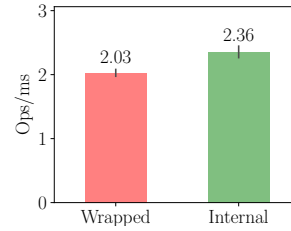
We observe that the throughput of the benchmark with snapshotting is overall lower than without and further decreases when the number of fold signals is higher. In the best case, i.e., there are no fold signals, the overhead is minimal; our solution incurs performance overhead only when state is actually stored, i.e., there is no overhead for an active but unused feature. In the worst case, i.e., when every reactive is a fold, the throughput of the run with snapshots is still about 58% of that with no snapshot. For typical reactive programs, however, which contain roughly 14% fold signals [44], the relative throughput is 85%. Moreover, the numbers reported so far are rather conservative and the real average throughput is higher, because typically only part of the graph, i.e., only a subset of the folds in each benchmark configuration, changes its state during an update. To recap, we consider the overhead of our snapshot mechanism reasonable.

(RQ5) Restoring from snapshots versus recomputing. We first quantify the overall cost that recovery adds when restarting the application (1). We also quantify the tradeoff between taking minimal snapshots versus taking bigger snapshots (2).

Regarding (1), Figure 7 shows the results of measuring the cost of recovery for the graph from (RQ4). Each bar on the x -axis shows the throughput of creating a graph (a) without



■ **Figure 9** Flink vs. REScala snapshot performance.



■ **Figure 10** Integrated error propagation versus Try-based solution.

any support for fault tolerance, (b) with support for fault tolerance but when restoring from an empty (fresh) snapshot, as is the case when an application is started for the first time, and (c) when restoring the graph from an existing fully populated snapshot. The overhead we observe in the last case is the result of creating the initial snapshot and restoring the (serialized) values from the snapshot. We conclude from Figure 7 that while restoration has a certain overhead, the cost is comparable to normal application startup times, since REScala restores the graph of 100 reactivities twice per millisecond, compared to starting the application, which is performed 2.5 times per millisecond.

Regarding (2), as already mentioned, our approach minimizes the amount of state that is stored in snapshots, hence we tradeoff restoring derived state against recomputing it. Intuitively, one would assume that our restoration has higher overhead compared to one that starts from a maximal snapshot, as it has to recompute more. However, a small experiment indicates that this does not necessarily have to be the case. In the experiment, we run two versions (labeled Restore and Derive) of a benchmark with a dataflow graph that stores a list containing integers 1 to N . In the Restore version the list is part of the snapshot, while in the Derive version the snapshot only contains the size of the list and the list itself is recomputed during restoration. The graphs in Figure 8 show the results, with N in the x-axis and throughput in the y-axis. We observe that (a) both restoring and recomputing derived state get linearly more expensive with the size of N and (b) recomputing the list given its size is faster than restoring from a complete snapshot of it. This indicates that our approach of deriving as much state as possible from minimal snapshots during recovery does not only make snapshotting efficient, but can also be beneficial to restoration performance.

(RQ6) Comparison to an industrial-strength data streaming system. Our objective in this experiment is to compare the performance of our prototype implementation for snapshots and recovery to a functionally similar industrial-strength system. The objective is to measure an upper bound for the performance of our system. We chose Flink [6], a state-of-the-art, industrial strength, big data processing engine for real-time analytics used, among the others, in the Alibaba real-time search ranking, in Zalando’s business process monitoring and in Netflix’s complex event processing system [2]. Flink is suitable as a reference due to the following reasons: (a) it is functionally similar to reactive applications in that it also manages state inside of a dataflow graph (a property it shares with other streaming systems), (b) it is implemented in Scala, hence the runtime environment is similar to ours, (c) it is well known for its focus on fault tolerance, (d) it is also possible to enable/disable snapshots, and (e) both Flink and REScala serialize snapshots to memory.

We implemented a similar graph structure as in (RQ4) for Flink. However, Flink and REScala target different usage scenarios, where REScala immediately reacts to individual occurrences of input events, such as button clicks, Flink processes and aggregates complete

input streams of data. Hence, we do not compare the absolute performance of Flink and REScala, but only measure the relative overhead of creating snapshots.

In Figure 9, we show the throughput relative to execution without snapshots (checkpoints in Flink terminology). Snapshots in Flink are created periodically instead of after each update (we have created them every 10 ms, 100 ms, and 1000 ms, respectively), and always include the complete state of the system. While the overhead of REScala is higher when a full snapshot is created, in the case when only 30% of the dataflow graph is stored in the snapshot – which is the realistic case – the relative overheads of both systems are similar.

We conclude that the performance of our snapshot algorithm is comparable to Flink. Yet our prototype is a proof-of-concept, and has not been extensively optimized. This observation is an indication of the benefits of exploiting features of the reactive programming paradigm, specifically automatically managed state, in the design of REScala.

(RQ7) Performance effects of language-integrated error propagation. In Section 5, we motivated the need for language-integrated error propagation for the quality of application design. By answering RQ3, we empirically provided evidence that our approach to error propagation indeed barely pollutes the application code. The experiments presented below analyze the potential performance effects of this non-invasive error handling. Specifically, we analyze (a) the potential performance tradeoffs of the language-integrated error propagation compared to programmatic error handling, and (b) the overhead of the error propagation system in the absence of errors.

These experiments show that there is no additional cost. As discussed in Section 6, this is due to the tight integration of errors into the existing runtime. Moreover, language-integrated error propagation exhibits better application performance compared to programmatic error handling.

For (a), we implemented a reactive program with programmatic error handling by using Scala’s `Try` to propagate errors, so every `Signal[A]` becomes a `Signal[Try[A]]`. Using `Try` is the idiomatic way to represent errors as values in Scala, similar to the `Maybe` data type in Haskell. As shown in Figure 10, our solution outperforms the solution that uses `Try`-wrappers. This improvement is due to the fact that language integration merges error propagation into the internal data structures of the language runtime, while `Try`-wrappers require an additional layer of indirection; in addition, the solution that uses wrappers requires unwrapping code at every signal expression⁶. The first line of code below just adds two values, compared to the second line of code where adding two values becomes unwieldy when nested `Try` expressions need to be unwrapped, even when using Scala’s special `for` syntax:

```
Signal { a.value + b.value } // without wrapper
Signal { for (av <- a.value; bv <- b.value) yield av + bv } // wrapped in Try
```

For (b), we use the REScala benchmark *natural graph*, a graph with 25 reactivities that are connected in a way to mimic real applications [44], to show how the performance of an average application is affected. All user-defined computations only perform arithmetic additions to minimize the amount of work that is spent on actual computation and maximize the relative overhead of the error propagation. We did not measure any performance degradation when error propagation is enabled but the application does not use it, thus developers only have to pay for what they use.

⁶ Other representations of errors are possible, but they all have to share the same pattern of `Try`, both values and errors need to be represented in a single data type, and the application developer has to manually differentiate the two cases in signal expressions.

7.3 Threats to Validity

There are both internal and external threats to the validity of our results. Internal threats are due to the inspection of case studies for analyzing the non-invasiveness of our approach, as what is non-invasive is subjective and depends on our experience in developing REScala applications. Also, the results are not confirmed by subjects without experience with REScala.

An external threat is that the benchmarks may be too small and not sufficiently diverse to be representative of reactive programming applications for the results to be generalizable. Unfortunately, at the time being there are no standard benchmarks for reactive programming languages. Given the lack of any widely accepted benchmark suite (to the best of our knowledge), our selection of the benchmarks is strongly based on our experience with REScala applications. Extending the benchmark suite with more, diverse and larger-scale case studies will be addressed in future work.

8 Related Work

Languages for reactive applications. Non-distributed languages for traditional desktop applications are usually concerned with I/O errors during execution, but typically do not provide facilities to snapshot or restore program state. In the object-oriented paradigm, reactive software is often developed using the Observer design pattern. This approach, extensively discussed in the literature [17, 27, 31, 13], leads to the inversion of the control flow, which complicates code analysis and induces highly error-prone broadly scoped side-effecting operations: since observers do not return a value, computational results need to be passed through imperative state changes, prohibiting all of the techniques for fault tolerance discussed in this paper.

Functional reactive programming (FRP) [18] models time-changing values, whose denotations are functions focusing on the problem of formally modelling continuous time. FRP has been used in a number of areas, including robotics [23], network switch programming [20, 48], wireless sensor networks [36], and reliable software for spacecraft [37]. In general, FRP seems to be a natural fit for distributed applications [29, 40, 41, 15, 11], with events representing messages from the network or user input. However, many functional reactive languages and frameworks do not provide support for unreliable networks. Typically, reactive languages [27, 31, 14] simply delegate the responsibility for error handling to the host language, and ultimately to the programmer. In distributed reactive programming [15, 28, 47], reactives on different devices are connected to each other and update messages are sent over the network whenever a remote dependency changes. In the presence of faulty devices and unreliable connections, such update messages may get lost causing several problems, such as (a) glitches, (b) changes that are visible on one host but not on another host, or (c) application unresponsiveness when new changes cannot be processed while messages are being resent to a device that failed and is restored.

Unreliability has been partially investigated in the context of some FRP derivatives. Timeouts have been introduced to a distributed runtime and dataflow [35]. ReactiveExtensions (Rx) [26] integrate and propagate errors into the dataflow. However, to the best of our knowledge, no solution exists to automatically restore and reconnect a dataflow graph after a crash. DREAM [28, 29] is a middleware for distributed reactive programming, which lets the programmer choose among different levels of consistency guarantees in distributed reactive systems, including FIFO consistency, causal consistency, glitch freedom and atomic consistency. However, none of these approaches provides the consistency guarantees of REScala automatically. Ur/Web [12] is a multitier programming language that uses reactive

programming to update the client UI. However, to the best of our knowledge, there is no integration of RPC errors and the reactive part, hindering application-wide reasoning and lacking common abstractions for distribution and reactivity.

Actor and cloud languages. Actors [1, 7, 9, 33, 46, 24] are well-known abstractions to model concurrent and distributed systems. Actors do not share mutable state and communicate only via message passing. The result is a loose application structure that makes automatic reasoning about overall system consistency very hard. Furthermore, we consider message-passing to be rather an implementation mechanism for enabling communication, which is by no means a proper substitute for providing first-class composable and programmable abstractions in the language, as it is the case with REScala.

Languages such as Erlang and Akka support restarts of crashed actors, possibly on a different device, but it is the responsibility of the application logic to be robust against such crashes. Otherwise, state on restarted actors is lost, and application properties are violated. Orleans [9] and extensions to Akka [1] can automatically restore the state of single actors after a crash. However, state is stored without consistency guarantees between multiple actors, and it is still difficult to reason about application properties. Akka additionally requires manual changes to each actor that requires fault-tolerant state, making it impossible to reuse existing actors developed without support for fault tolerance.

AmbientTalk [46] is an actor language specifically designed for mobile ad hoc networks, and Direst [34] builds on top of AmbientTalk and adds reactive abstractions and automatic eventually consistent state distribution. However, Direst uses a centralized replica to provide eventual consistency, hindering any communication between devices when the centralized replica is unavailable. Furthermore, applications in Direst cannot dynamically reconfigure their dependencies – a necessary concept for existing dynamic applications. Hence, Direst cannot support common reactive patterns, such as dynamically selecting the current view of an application at runtime, thus limiting reusability of components.

MBrace [16] extends F# with expressions for cloud computations. The use of immutable global references allows the distributed runtime to automatically re-execute tasks on failed devices without causing inconsistencies. Errors that are raised during the evaluation of cloud expressions, e.g., because a remote resource is unavailable, are transparently propagated along the dataflow path of the expression, even across the distribution boundaries, allowing non-localized error handling. However, since the distributed state is immutable, the abstractions are not well suited to reactive applications where the program state changes dynamically in response to input from the user or the execution environment/context.

Batch and stream processing languages. Frameworks for big data processing, such as Spark [49] and Flink [6], handle crashes of worker machines to minimize lost work when machines fail. They have recently also adapted syntax similar to FRP, but are not suited for reactive applications. Running in cluster environments with full control of communication and distribution of work among machines, Spark and Flink can offer abstractions for distribution and fault tolerance with suitable correctness guarantees. However, to provide these guarantees, applications are written in specific DSLs, and the execution runtime is not connected to the embedding application. The use of a DSL limits the capability to integrate with other libraries, and the DSL is not designed for reactive applications.

Building blocks for distributed applications. Several approaches provide building blocks to develop applications in distributed systems.

The counterpart of observers in non-distributed software are pub-sub systems in distributed software, with similar problems [19, 38, 30].

Eventual consistent data types such as CRDTs [45] or CloudTypes [10] are important building blocks providing well-understood tradeoffs between consistency and responsiveness. In case of connection failures, eventual consistent data types become out of sync with other replicas, but when the failures are only temporary, a consistent state can be automatically restored. However, on their own these data types cannot provide any application-wide correctness guarantees.

Function passing [32] is a style of distributed programming that defines a graph of immutable values and operations over these values. The result is a graph similar to Spark RDDs, but using arbitrary Scala functions instead RDD transformations, combining an abstraction for distributed systems with reusability of most Scala functions. However, since fault tolerance and reactivity are not part of the language, the language cannot enforce or check any properties.

9 Conclusion

In this paper, we presented REScala, a reactive programming language to support the development of fault-tolerant distributed reactive applications. REScala automatically handles crashes and disconnects between devices, supporting application specific recovery strategies. The fault tolerance mechanism provided by REScala is mostly transparent to the programmer, it preserves strong consistency on local devices in the presence of faults, and it ensures eventual consistency across distributed devices. It has no performance overhead when no faults occur and acceptable overhead otherwise. Our evaluation shows that creating snapshots and recovering from them has comparable overhead to similar existing solutions.

There are several areas for future work. We have discussed distributed glitch freedom in Section 4. In future work, we plan to adapt the propagation algorithm of Drechsler et al. [15] to detect such glitches and to use the error propagation mechanism to enable developers to compromise between availability and correctness. Finally, we plan to formalize our programming model to provide rigorous guarantees about application correctness in the presence of crashes and disconnects.

References

- 1 Akka documentation, 2017. URL: <http://akka.io/docs>.
- 2 Flink success stories, 2017. URL: <https://cwiki.apache.org/confluence/display/FLINK/Powered+by+Flink>.
- 3 HTML5 localStorage, 2017. URL: https://www.w3schools.com/html/html5_webstorage.asp.
- 4 Java microbenchmark harness, 2017. URL: <http://openjdk.java.net/projects/code-tools/jmh/>.
- 5 Mehdi Ahmed-Nacer, Claudia-Lavinia Ignat, Gérald Oster, Hyun-Gul Roh, and Pascal Urso. Evaluating CRDTs for Real-time Document Editing. In *Proceedings of the 11th ACM Symposium on Document Engineering*, DocEng '11, 2011. doi:10.1145/2034691.2034717.
- 6 Alexander Alexandrov, Rico Bergmann, Stephan Ewen, Johann-Christoph Freytag, Fabian Hueske, Arvid Heise, Odej Kao, Marcus Leich, Ulf Leser, Volker Markl, Felix Naumann, Mathias Peters, Astrid Rheinländer, Matthias J. Sax, Sebastian Schelter, Mareike Höger, Kostas Tzoumas, and Daniel Warneke. The Stratosphere Platform for Big Data Analytics. *The VLDB Journal*, 23, 2014. doi:10.1007/s00778-014-0357-y.

- 7 Joe Armstrong. Erlang. *Communications of the ACM*, 53, 2010. doi:10.1145/1810891.1810910.
- 8 Engineer Bainomugisha, Andoni Lombide Carreton, Tom van Cutsem, Stijn Mostinckx, and Wolfgang de Meuter. A Survey on Reactive Programming. *ACM Computing Survey*, 45(4), 2013. doi:10.1145/2501654.2501666.
- 9 P. Bernstein, Sergey Bykov, Alan Geller, Gabriel Kliot, and Jorgen Thelin. Orleans: Distributed Virtual Actors for Programmability and Scalability. Technical report, (MSR-TR-2014-41, 24), 2014. URL: <http://aka.ms/Ykyqft>.
- 10 Sebastian Burckhardt, Manuel Fähndrich, Daan Leijen, and Benjamin P. Wood. Cloud Types for Eventual Consistency. In *European Conference on Object-Oriented Programming (ECOOP)*, 2012.
- 11 Andoni Lombide Carreton, Stijn Mostinckx, Tom Van Cutsem, and Wolfgang De Meuter. Loosely-coupled Distributed Reactive Programming in Mobile Ad Hoc Networks. In *Proceedings of the 48th International Conference on Objects, Models, Components, Patterns, TOOLS'10*, 2010. URL: <http://dl.acm.org/citation.cfm?id=1894386.1894389>.
- 12 Adam Chlipala. Ur/Web: A Simple Model for Programming the Web. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '15*, 2015. doi:10.1145/2676726.2677004.
- 13 Gregory H. Cooper and Shriram Krishnamurthi. Embedding Dynamic Dataflow in a Call-by-value Language. In *Proceedings of the 15th European Conference on Programming Languages and Systems, ESOP*, 2006. doi:10.1007/11693024_20.
- 14 Evan Czaplicki and Stephen Chong. Asynchronous Functional Reactive Programming for GUIs. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*, 2013. doi:10.1145/2491956.2462161.
- 15 Joscha Drechsler, Guido Salvaneschi, Ragnar Mogk, and Mira Mezini. Distributed REScala: An Update Algorithm for Distributed Reactive Programming. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA*, 2014. doi:10.1145/2660193.2660240.
- 16 Jan Dzik, Nick Palladinos, Konstantinos Rontogiannis, Eirik Tsarpalis, and Nikolaos Vathis. MBrace: Cloud Computing with Monads. *Proceedings of the Seventh Workshop on Programming Languages and Operating Systems*, 2013. doi:10.1145/2525528.2525531.
- 17 Jonathan Edwards. Coherent Reaction. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications, OOPSLA*, 2009. doi:10.1145/1639950.1640058.
- 18 Conal Elliott and Paul Hudak. Functional Reactive Animation. In *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming, ICFP*, 1997. doi:10.1145/258948.258973.
- 19 Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The Many Faces of Publish/Subscribe. *ACM Computing Surveys*, 35(2), 2003. doi:10.1145/857076.857078.
- 20 Nate Foster, Rob Harrison, Michael J. Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. Frenetic: A Network Programming Language. In *Proceedings of the 16th ACM SIGPLAN international conference on Functional programming, ICFP*, 2011. doi:10.1145/2034773.2034812.
- 21 Seth Gilbert and Nancy Lynch. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services. *ACM SIGACT News*, 33(2), 2002. doi:10.1145/564585.564601.
- 22 Rachid Guerraoui, Matej Pavlovic, and Dragos-Adrian Seredinschi. Incremental Consistency Guarantees for Replicated Objects. *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016. arXiv:1609.02434.

- 23 Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. Arrows, Robots, and Functional Reactive Programming. In *Lecture Notes in Computer Science*, volume 2638, 2003.
- 24 Rajesh K. Karmani and Gul Agha. Actors. In *Encyclopedia of Parallel Computing*. Springer, 2011. doi:10.1007/978-0-387-09766-4_125.
- 25 Jean-Claude Laprie. Dependable Computing: Concepts, Challenges, Directions. *International Symposium on Fault-Tolerant Computing, FTCS*, 1995.
- 26 Jesse Liberty and Paul Betts. *Programming Reactive Extensions and LINQ*. Apress, 2011.
- 27 Ingo Maier and Martin Odersky. Deprecating the Observer Pattern with Scala.react. Technical report, EPFL, 2012.
- 28 Alessandro Margara and Guido Salvaneschi. We Have a DREAM: Distributed Reactive Programming with Consistency Guarantees. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems, DEBS*, 2014. doi:10.1145/2611286.2611290.
- 29 Alessandro Margara and Guido Salvaneschi. On the Semantics of Distributed Reactive Programming: The Cost of Consistency. *IEEE Transactions on Software Engineering*, 2018.
- 30 R. Meier and V. Cahill. Taxonomy of Distributed Event-based Programming Systems. In *22nd International Conference on Distributed Computing Systems Workshops*, 2002. doi:10.1109/ICDCSW.2002.1030833.
- 31 Leo A. Meyerovich, Arjun Guha, Jacob Baskin, Gregory H. Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. Flapjax: A Programming Language for Ajax Applications. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA*, 2009. doi:10.1145/1640089.1640091.
- 32 Heather Miller, Philipp Haller, Normen Müller, and Jocelyn Boullier. Function Passing: A Model for Typed, Distributed Functional Programming. In *Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward!*, 2016. doi:10.1145/2986012.2986014.
- 33 Mark S. Miller, E. Dean Tribble, and Jonathan Shapiro. Concurrency Among Strangers. In *Proc. Int. Symp. on Trustworthy Global Computing*. Springer, 2005. doi:10.1007/11580850_12.
- 34 Florian Myter, Tim Coppieters, Christophe Scholliers, and Wolfgang De Meuter. I Now Pronounce You Reactive and Consistent: Handling Distributed and Replicated State in Reactive Programming. In *Proceedings of the 3rd International Workshop on Reactive and Event-Based Languages and Systems, REBLS*, 2016. doi:10.1145/3001929.3001930.
- 35 Florian Myter, Christophe Scholliers, and Wolfgang De Meuter. Handling Partial Failures in Distributed Reactive Programming. *4th Workshop on Reactive and Event-based Languages & Systems*, 2017.
- 36 Ryan Newton, Greg Morrisett, and Matt Welsh. The Regiment Macroprogramming System. In *2007 6th International Symposium on Information Processing in Sensor Networks*, 2007. doi:10.1109/IPSN.2007.4379709.
- 37 Ivan Perez. Fault Tolerant Functional Reactive Programming. *International Conference on Functional Programming (ICFP)*, 2018.
- 38 Peter R. Pietzuch and Jean M. Bacon. Hermes: A Distributed Event-based Middleware Architecture. In *Proceedings. 22nd International Conference on Distributed Computing Systems Workshops*, 2002. doi:10.1109/ICDCSW.2002.1030837.
- 39 J. Ploski and W. Hasselbring. Exception Handling in an Event-Driven System. In *Availability, Reliability and Security. ARES.*, 2007. doi:10.1109/ARES.2007.85.

- 40 José Proença and Carlos Baquero. Quality-Aware Reactive Programming for the Internet of Things. In *Fundamentals of Software Engineering - 7th International Conference, FSEN*, 2017.
- 41 Bob Reynders, Dominique Devriese, and Frank Piessens. Multi-Tier Functional Reactive Programming for the Web. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software, Onward!*, 2014. doi:10.1145/2661136.2661140.
- 42 G. Salvaneschi, S. Proksch, S. Amann, S. Nadi, and M. Mezini. On the Positive Effect of Reactive Programming on Software Comprehension: An Empirical Study. *IEEE Transactions on Software Engineering*, 43(12), Dec 2017. doi:10.1109/TSE.2017.2655524.
- 43 Guido Salvaneschi, Sven Amann, Sebastian Proksch, and Mira Mezini. An Empirical Study on Program Comprehension with Reactive Programming. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE*, 2014. doi:10.1145/2635868.2635895.
- 44 Guido Salvaneschi, Gerold Hintz, and Mira Mezini. REScala: Bridging Between Object-oriented and Functional Style in Reactive Applications. In *Proceedings of the 13th International Conference on Modularity, MODULARITY*, 2014. doi:10.1145/2577080.2577083.
- 45 Marc Shapiro, Nuno Pregui, Carlos Baquero, and Marek Zawirski. A Comprehensive Study of Convergent and Commutative Replicated Data Types. Technical report, INRIA, 2011.
- 46 Tom Van Cutsem, Elisa Gonzalez Boix, Christophe Scholliers, Andoni Lombide Carreton, Dries Harnie, Kevin Pinté, and Wolfgang De Meuter. AmbientTalk: Programming Responsive Mobile Peer-to-peer Applications with Actors. *Computer Languages, Systems & Structures*, 40(3-4), 2014. doi:10.1016/j.cl.2014.05.002.
- 47 Tom Van Cutsem, Stijn Mostinckx, Elisa Gonzalez Boix, Jessie Dedecker, and Wolfgang De Meuter. AmbientTalk: Object-oriented Event-driven Programming in Mobile Ad Hoc Networks. *Proceedings - International Conference of the Chilean Computer Science Society, SCCC*, 2007. doi:10.1109/SCCC.2007.4396972.
- 48 Andreas Voellmy, Hyojoon Kim, and Nick Feamster. Procera: A Language for High-level Reactive Network Control. In *Proceedings of the First Workshop on Hot Topics in Software Defined Networks, HotSDN*, 2012. doi:10.1145/2342441.2342451.
- 49 Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation, NSDI*, 2012. URL: <https://www.usenix.org/system/files/conference/nsdi12/nsdi12-final138.pdf>.

ContextWorkflow: A Monadic DSL for Compensable and Interruptible Executions

Hiroaki Inoue¹

Graduate School of Informatics, Kyoto University, Kyoto, Japan
hinoue@fos.kuis.kyoto-u.ac.jp

Tomoyuki Aotani

School of Computing, Tokyo Institute of Technology, Tokyo, Japan
aotani@c.titech.ac.jp

Atsushi Igarashi

Graduate School of Informatics, Kyoto University, Kyoto, Japan
igarashi@kuis.kyoto-u.ac.jp

Abstract

Context-aware applications, whose behavior reactively depends on the time-varying status of the surrounding environment – such as network connection, battery level, and sensors – are getting more and more pervasive and important. The term “context-awareness” usually suggests prompt reactions to context changes: as the context change signals that the current execution cannot be continued, the application should immediately abort its execution, possibly does some clean-up tasks, and suspend until the context allows it to restart. Interruptions, or asynchronous exceptions, are useful to achieve context-awareness. It is, however, difficult to program with interruptions in a compositional way in most programming languages because their support is too primitive, relying on synchronous exception handling mechanism such as `try-catch`.

We propose a new domain-specific language *ContextWorkflow* for interruptible programs as a solution to the problem. A basic unit of an interruptible program is a workflow, i.e., a sequence of atomic computations accompanied with compensation actions. The uniqueness of ContextWorkflow is that, during its execution, a workflow keeps watching the context between atomic actions and decides if the computation should be continued, aborted, or suspended. Our contribution of this paper is as follows; (1) the design of a workflow-like language with asynchronous interruption, checkpointing, sub-workflows and suspension; (2) a formal semantics of the core language; (3) a monadic interpreter corresponding to the semantics; and (4) its concrete implementation as an *embedded domain-specific language* in Scala.

2012 ACM Subject Classification Software and its engineering → Domain specific languages

Keywords and phrases workflow, asynchronous exception, checkpoint, monad, embedded domain specific language

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2018.2

Supplement Material ECOOP Artifact Evaluation approved artifact available at <http://dx.doi.org/10.4230/DARTS.4.3.4>

Funding This work was supported in part by Kyoto University Design School (Inoue).

Acknowledgements We thank Hidehiko Masuhara and anonymous reviewers for valuable comments.

¹ The current affiliation is Mitsubishi Electric Corporation.



1 Introduction

As mobile computing devices have spread, recent applications tend to depend on external information (called *context*) that is time-varying, such as battery level, heat, human input, network connection, and availability of external modules; these applications are so-called *context-aware applications* [3, 7]. Context-aware applications are usually required to promptly react to context changes; hence, they have to be *interruptible* or support *asynchronous interruption*.

An example is a package manager application that updates packages in an operating system or a software development environment. It tends to be running for a long time because, even if only one package is selected by the user for the update, it is necessary to resolve package dependency, download archive files, unpack them, and more: the whole task takes a considerable amount of time. Examples of the interruptions are network disconnection and a press of the “cancel” button. Another example is a battery-powered robot that moves around to do some task such as cleaning rooms. Examples of the interruptions are a low battery level and sensor malfunction.

Reactions to interruptions cannot be simple. In the package manager, for example, it is not desirable just to abort the package manager promptly in response to a press of the “cancel” button because the package dependency may be broken, i.e., packages may be partially updated/installed. A desirable package manager must ensure the consistency of packages by performing some recovery actions, e.g., reverting the update by re-installing the previous versions of the packages. It may also be preferable in the case of network disconnection to suspend the execution until the connection comes back. In the robot example, a desirable reaction to a low battery level is stopping the task and returning to a base for charge.

The two examples show that, if an interruption occurs, it is necessary for context-aware applications to be able to promptly (1) abort with reverting the “effects” that comes from uncompleted tasks (file replacement in the first example and robot movement in the second example) or (2) suspend the program until we can run the continuations or reversions.

Developing context-aware applications in existing mainstream languages is difficult because of the following two problems. First, as Bainomugisha et al. indicated [7], the languages lack constructs for promptly reacting to context changes. Inserting the code for the context checks manually is not desirable from a modularity perspective. *Asynchronous exceptions* [40], which enable us to throw exceptions to other threads, could be a solution to the point. It is, however, still weak for context-aware applications because the context usually depends on multiple time-varying data and asynchronous exceptions themselves are not helpful for tidying them up.

Second, support for recovery from asynchronous interruption in the existing languages is weak. Although today’s standard approach to handling interruptions is to use the exception handling constructs such as `try-catch-finally`, they are not useful for reversion and suspension; in particular, reversion is similar to resource handling with exceptions, which is hard with the constructs [60]. A more complicated and difficult reaction is *partial abort* [25], which is a combination of reversion and suspension and is realized by using *checkpoints* [48, 62, 17]. Checkpoints are useful to make applications robust [41] and avoid wasteful recomputation [14].

Our solution to the problems is based on the ideas of *Flute* [7] and *workflow* [22, 12]. Flute is a programming language originally proposed to solve the first problem. To represent the context depending on multiple time-varying data, Flute uses *functional reactive programming* (FRP) [19, 6] that represents time-varying values as streams and provides operations over them, which are useful to unify multiple sensory data into one stream. Flute also supports suspending the program execution.

Workflow [22, 12] represents a long-running interruptible transaction that consists of several atomic transactions. The typical applications are web applications and business process management, and recently workflow is adapted to context-aware applications [44, 4, 54]. One important idea of workflow for us is *compensation* [60], where each action of a program is accompanied by a compensation action, meaning a recovery action; and program execution takes account of its progress and automatically constructs its recovery action.

1.1 Contributions

In this paper, we propose a language *ContextWorkflow* as a solution to the two problems. *ContextWorkflow* is a workflow-based language that supports compensation, asynchronous interruption, suspension, and checkpoints. It also provides *sub-workflows* and *programmable compensations* [9, 12] that ignore and replace the compensations of completed portions of workflow, respectively.

Our approach to implementing *ContextWorkflow* is to embed it in other “host” languages [31]. The benefit of the approach is that the language itself remains small but can be powerful because any features of the host language are still available.

Our technical contributions are (1) a design of the workflow-based programming language with asynchronous interruption, (2) a formalization of the language, including the big-step operational semantics, (3) monadic interpreters corresponding to the semantics, and (4) an implementation of *ContextWorkflow* by embedding into Scala. The details are as follows.

Asynchronous Interruption in Workflow. Our approach to asynchronous interruption uses signals of FRP [19, 6] and polling [20], and our novel finding is that the idea of workflow and compensation fit with the approach. A workflow in *ContextWorkflow* is executed under some *context*, which changes over time asynchronously and indicates how the execution of workflow proceeds. An asynchronous interruption is detected by checking the context. We suppose that each atomic transactions should not be interrupted asynchronously; and we regard atomic transactions as a primitive construct of our language. The context is checked at the beginning of each atomic transaction similarly to transactions in database [24] and software transactional memory [53]. The difference between our workflow and the transactions is with regard to the time when a check runs. In the transactions, a check runs at the end. We also introduce constructs for blocking interruptions as in Concurrent Haskell [40] for avoiding unnecessary context checks.

Formalizing ContextWorkflow. We develop a big-step operational semantics that models the essential constructs of *ContextWorkflow*, that is, workflow, compensation, asynchronous interruption, sub-workflows, programmable compensations, checkpoints, and suspension. The semantics is inspired by Bruni et al.’s formalization [9] of Sagas [22], which is a foundation of workflow. Our main contribution is to add checkpoints and suspension to the existing semantics, especially considering sub-workflows. We also provide and prove basic properties of the new calculus and describe small extensions. In addition, we discuss whether the polling code should be inserted before or after an atomic transaction using the core calculus.

Monadic Interpreter. We develop two monadic interpreters in lazy and eager languages that closely correspond with the big-step operational semantics. We define two *CW* monads using the reader, exception monads and free monad transformers that represent the abstract syntax trees of *ContextWorkflow* programs. One could define the *CW* monad based on the free monad [5] over the compensation functor [47] that consists of the exception and continuation

monads. Such a definition is, however, not desirable because it is hard to support sub-workflows, programmable compensations, and checkpoints while keeping correspondence with the big-step operational semantics straightforward. We instead use the free monad transformers to define the CW monads. Note that the functions that collapse, or fold, free monad transformers are different between eager and lazy languages due to efficiency and stack safety [56]. Two monads and monadic interpreters are therefore necessary.

Embedding in Scala. We carefully embed ContextWorkflow in Scala based on the monadic interpreter. In our embedding, one can throw Scala exceptions using `throw` in atomic actions and handle them using Scala’s standard exception handling mechanism. We use the macro system in Scala to make the ContextWorkflow program syntax look more natural.

The rest of this paper is organized as follows. In Section 2, we informally introduce ContextWorkflow with a running example of a maze search robot. Section 3 provides a formal calculus of the core ContextWorkflow. In Section 4, we construct a monadic interpreter and show further implementation techniques in Scala. Section 5 presents related work, followed by future work and conclusion.

2 ContextWorkflow Constructs.

In this section, we look at the basic constructs of ContextWorkflow using a maze search program as a running example. Here, the notation is based on our implementation, which is an EDSL in Scala.

A program in ContextWorkflow is a workflow that is a sequence of *primitive workflows* (similar to atomic transactions). When an interruption takes place – it can only occur between primitive workflows – the whole workflow is aborted after running the compensations of the already completed primitive workflows in the reverse order, or is suspended (and the rest of the computation is returned).

2.1 Example: Explorer Robot

As a running example, we consider a battery-powered robot that explores a (physical) maze. Our goal is to program the following context-dependent behavior:

1. The robot must get back to the start or a special point equipped with a battery charger, at which the robot can recharge its battery. (We call such a special point simply a charger.)
2. When it starts to rain, the robot should suspend its exploration.

Our basic exploration strategy is to visit every place in the maze in the depth-first search (DFS) manner. We assume that the maze is represented by a graph; the graph is represented as a set of nodes, which consist of two-dimensional coordinates of integers. A node is connected to another node if and only if the distance between the two nodes is one, e.g., (1,0) and (1,1) are connected, but (1,0) and (1,2) are not. This means that if a pair of coordinates is not in the node set, there is a wall at that position. We define the class `Node` for nodes and functions as follows.

```
case class Node(loc:(Int,Int), var visited:Boolean)
def neighbors(n: Node, maze: Set[Node]): List[Node] = // getting the neighbors of n
def visited(n: Node): Unit = {n.visited = true} // setting the visited flag of n on
def unknown(n: Node): Unit = {n.visited = false} // setting the visited flag off
def move(n: Node): Unit = /* actually moving the robot to n */
def visit(n: Node, maze: Set[Node]): Unit = { // main search program
```

```

visited(n);
neighbors(n, maze).foldLeft() { (_, neighbor) =>
  if (!neighbor.visited) { move(neighbor); visit(neighbor, maze); move(n); }
} }

```

A `Node` has coordinate information `loc` and a flag `visited` that is used to remember whether the node has been visited or not. The function `neighbors` returns the neighboring nodes of a given node `n`. The functions `visited` and `unknown` mark the given node `n` as visited and unvisited, respectively. The function `move` takes a node as an argument and moves the robot to the position it represents. It works only if the robot is currently at its neighbor or the node itself. The function `visit` is the main function that must be refined as our development proceeds; it takes a node `n` and a graph `maze`, and just visits every node in `maze` from `n` recursively in a DFS manner without allowing any interruptions.

In the rest of this section, we revise `visit` using the features of `ContextWorkflow`. We use compensations to move the robot back; suspension to stop the robot when it starts to rain; nested workflow to skip some compensation actions; blocking constructs `atomic` and `nonatomic` to avoid redundant/unnecessary context checks; and checkpoints to stop the robot at a charger while it is getting back.

2.2 Interruptible and Compensable Workflow

To make `visit` interruptible and compensable, we change it to a sequence of primitive workflows. We write a primitive workflow, which consists of a normal action `n` and a compensation action `c`, as `n /+ c` in `ContextWorkflow`. Normal and compensation actions can be any Scala code (of certain types).

Each function call to `visited`, `move`, and `visit` should be lifted to a primitive workflow because it changes the “state,” i.e., the flags of nodes and the position of the robot. If an interruption occurs, the changes have to be reverted by compensations. The compensation action of each function call is basically its inverse in our example.² For example, we define a primitive workflow `moveFromTo` for `move` with its reverse as follows:

```

def moveFromTo(from: Node, to: Node): CW[Unit] = move(to) /+ (_ => move(from))

```

The normal action `move(to)` is of the type `Unit`, and the compensation `_ => move(from)` is of the type `Unit => Unit`; a compensation action takes the result of the corresponding normal action – which has been finished – as an argument. The whole primitive workflow is of the type `CW[Unit]` where `CW` is the class representing a workflow and means the workflow returns a value of `Unit` after its successful execution. A workflow, which is an instance of `CW[T]`, can be run by invoking `exec`, which will be explained shortly.

`ContextWorkflow` provides the `workflow` block and the operator `!!` to combine two or more (primitive) workflows. The `workflow` block is used to build a long workflow, and the `!!` operator is used to sequence workflows in the `workflow` block.

```

def workflow[T](body: T): CW[T]
def !![T](m: CW[T]): T

```

For example, we write like `workflow{ val x = !!(m); !!(f(x)); ... }`, where `x` becomes the result of the workflow `m`. If unnecessary, `val * =` can be omitted. This notation is

² The compensation action is not necessarily the inverse of the normal action in general. The purpose of the compensation action is to ensure the “state” is acceptable even if an interruption occurs and the program stops or rolls back.

2:6 ContextWorkflow

almost the syntactic sugar of for-comprehension in Scala; e.g., the foregoing code is equal to `m.flatMap(x => f(x).flatMap(_ => ...))`. We can also use ordinary `if` for branching and `fold` (called `foldCW`) for iteration in `ContextWorkflow`.

```
def foldCW[A,B](l: List[A])(z: B)(f: (B,A) => CW[B]): CW[B] // fold for CW
```

Then, the interruptible version of `visit` is as follows.

```
def visit(n: Node, maze: Set[Node]): CW[Unit] = workflow {
  !!(visited(n) /+ (_ => unknown(n)) // reversible visited
  !!{foldCW(neighbors(n, maze))(()){(_, neighbor) =>
    if(!neighbor.visited) workflow{
      !!(moveFromTo(n, neighbor)) // the robot moves to the neighbor
      !!(visit(neighbor, maze))
      !!(moveFromTo(neighbor, n)) // the robot gets back to the original node n
    }
  } else () /+ () } } }
```

Note that compensation actions are inverses of their corresponding normal actions.

To execute a workflow, we invoke the method `exec` of `CW` class:

```
def exec(...): \[Option[CW[A]], A]
```

where `\` introduces disjunctions of two types whose constructors are `-\[l]` (meaning the left value) and `\-(r)` (meaning the right value); `Option` is the type of optional values consisting of `Some(a)` and `None`. The type `\[Option[CW[A]], A]` represents that the result may be abort `-\[None]`, suspended workflow `-\[Some(cw)]` or successful execution `\-(a)`. The argument of `exec` is optional and will be explained in detail later.

2.3 Interruption and Context

We need contexts to interrupt execution of the workflow in `ContextWorkflow`. A context signals how the execution of a workflow proceeds and changes over time asynchronously.

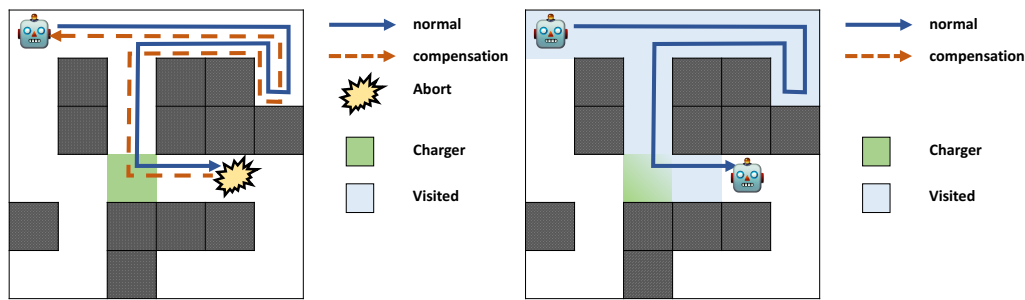
The context is represented by a stream of values of type `Context`, which can be any of `Continue`, `Abort`, `PAbort`, or `Suspend`. Their meanings are as follows:

- `Continue` continues the execution; normal actions are executed with their compensations recorded.
- `Abort` aborts the execution after executing the recorded compensations.
- `PAbort` means partial abort, which is similar to `Abort` but sensitive to checkpoints: it rolls back by executing the recorded compensations until the checkpoint most recently passed and returns the continuation at the checkpoint.
- `Suspend` suspends the execution and returns the rest of the workflow.

The current context is checked periodically (similarly to polling). More concretely, this periodic checking, called *context checking*, takes place before the execution of the normal action of each primitive workflow; if the current context is not equal to `Continue`, it is interrupted immediately.

To create a stream of `Contexts`, we use a signal in the FRP library `REScala` [51]. For example, we can represent an interruption due to a low battery level as a signal of `Context` as follows, assuming that there is another signal `battery` indicating the battery level.

```
val battery: Signal[Int] = /* a signal indicating the battery level */
val lowbattery: Signal[Context] = Signal{ if(battery() < 20) Abort else Continue }
```

■ **Figure 1** Maze search simulation: Abort (left) and Suspend (right).

The signal `lowbattery` is of the type `Signal[Context]`, whose value is `Continue` while the battery level is higher than 20% and `Abort` otherwise.

The context may depend on multiple sensory data. Such a context is easy to represent, owing to the expressiveness of REScala. For example, to suspend the robot when the rain starts, we need another sensory data that reflects the weather condition. It is achieved by creating another signal that relates to both the battery level and the weather.

```
val weather: Signal[Context] = Signal{ if(/* badWeather */) Suspend else Continue }
val mazectx: Signal[Context] = Signal { (lowbattery(), weather()) match {
  case (Continue, Continue) => Continue
  case (Abort, _) => Abort
  case _ => Suspend } }
```

The signal `mazectx` depends on not only `lowbattery` but also `weather`, which is another context related to the weather. Notice that we also give precedence between the two contexts here: `Abort` from `lowbattery` supersedes `Suspend` from `weather`.

To make our workflow depend on `mazectx`, we need to give it as the argument to `exec`:

```
visit(...).exec(mazectx)
```

Fig. 1 illustrates an execution of `visit`, where it is aborted (left) or suspended (right) halfway. Currently, a partial abort at the same place results in the same trace as the aborted case, since chargers (checkpoints) are not set yet.

A suspended workflow is also a workflow and we can start it by writing as follows:

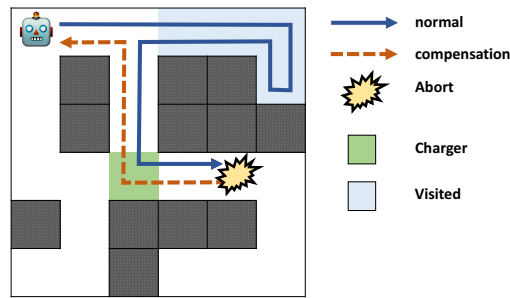
```
val r = visit(...).exec(...)
sleep(/*until it is ready to resume the program*/);
r match { case -\/(Some(s)) => s.exec(...) } // restart if suspended
```

Here, `s` is the suspended workflow and its type is `CW[Unit]`.

2.4 Nested Workflow and Programmable Compensations

Sometimes we would like to skip some compensation actions. In our example, the behavior of the aborted case is not desirable because the robot follows exactly the path in which it came to the aborted point and does not come back straight to the start. A better compensation would be to take a shortcut to the start node as shown in Fig. 2.

This can be achieved by delimiting a part of a workflow and ignoring the compensation actions of the delimited part if the part is completed successfully. We call such a part *sub-workflow* and provide a construct `sub` that makes a part of workflow a sub-workflow:



■ **Figure 2** Maze search simulation: Abort (refined).

```
def sub[A](cw: CW[A]): CW[A]
```

We revise `visit` by using `sub` to skip undesirable compensation actions as follows:

```
1 def visit(n: Node, maze: Set[Node]):CW[Unit] = workflow {
2   ...
3   if(!neighbor.visited)
4     sub{ workflow{
5       !!(moveFromTo(n, neighbor))
6       !!(visit(neighbor, maze))
7       !!(moveFromTo(neighbor, n))
8     } } ... }
```

If a partial search from the `neighbor` is complete, compensations for it will be skipped.

It is possible to perform another compensation action instead of just skipping the compensation actions within sub-workflows by writing something like `sub(...)/+ comp`, which is the so-called programmable compensation [9, 12]. For example, we can add a log:

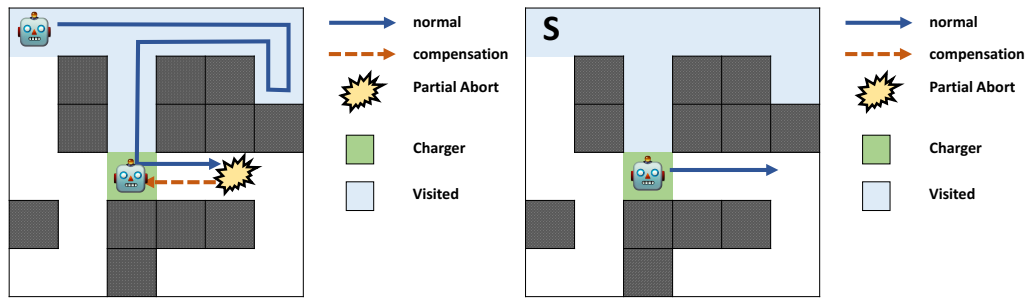
```
sub{ ... } /+ (_ => println("skipping compensations"))
```

2.5 Checkpoint

Using the above constructs, we still cannot realize the behavior of the robot so that it gets back to a charger. What we have to do is to let the robot *partially* roll back its move and suspend at the charger. For this purpose, we use checkpoints. A checkpoint saves the current execution state when it is passed. If a workflow is partially aborted, it executes only compensations until the checkpoint most recently passed and then suspends.

Let `Node` have another flag `hasCharger` that represents whether the node has a charger or not. We just add a `checkpoint`, which is a construct provided by `ContextWorkflow`, into the method `visit` as follows:

```
class Node(..., hasCharger:Boolean)
def visit(n: Node, maze: Set[Node]):CW[Unit] = workflow {
  !!(visited(n) /+ (_ => unknown(n)))
  if(n.hasCharger) !!(checkpoint) // checkpoint setting
  !!{foldCW(...)(...){...} }
}
```



■ **Figure 3** Maze search simulation: Partial abort (left) and its restart (right).

The left side of Fig. 3 illustrates a search being partially aborted and suspended at the checkpoint (charger). If `exec` on the suspended workflow, returned by the partial abort, is invoked, then the robot moves again from the charger (the right side of Fig. 3).

2.6 Blocking Context Checking

We would like to avoid redundant/unnecessary context checks from an efficiency perspective. In our example, it is not necessary to check the context at the beginning of (1) marking the node as visited and (2) skipping (i.e., `()/+()`) because they take little time. `ContextWorkflow` provides `atomic` and `nonatomic` blocks to activate and deactivate context checks.

```
def atomic[A](cw: CW[A]): CW[A]
def nonatomic[A](cw: CW[A]): CW[A]
```

An `atomic` block restrains context checking inside it, and a `nonatomic` block enforces context checking inside it. If they are nested inside each other, the innermost block takes effect.

Then, we refine the method `visit` as follows:

```
1 def visit(n: Node, maze: Set[Node]): CW[Unit] =
2   atomic{ workflow {
3     !!(!visited(n) /+ (_ => unknown(n))); ...
4     !!{foldCW(neighbors(n, maze))(())(_ , neighbor) =>
5     if(!neighbor.visited)
6     nonatomic{ sub{ workflow{ ... visit(...); ... } } }
7     else () /+ () } } }
```

By enclosing the whole workflow (except the sub-workflow) by `atomic`, context checks will not be performed on lines 3 and 7.

The purpose of `atomic` and `nonatomic` blocks is not only to improve the efficiency but also to control the atomicity of interruption. Such constructs are also very common in the languages supporting asynchronous exceptions and/or transactions; for example, Concurrent Haskell [40] has blocking constructs of asynchronous exceptions `block` and `unblock`.

3 Operational Semantics of Core ContextWorkflow

In this section, we describe the operational semantics of `ContextWorkflow` by formalizing a core calculus, which models compensation, checkpoints, sub-workflow, programmable compensations, and context-checking. Our calculus is inspired by Bruni et.al's formalization of Sagas [9]. Our main contribution is how to treat suspension and checkpointing considering sub-workflows, in the context of workflow languages.

t	::= $A/C \mid \text{check} \mid \text{cp} \mid \text{cp}\#E \mid \text{sub}(t)/C \mid t;t$	(workflows)
A, C	::= $\epsilon \mid \dots$	(atomic actions)
c	::= $C \mid \text{sub} \mid \text{ccp}\#E$	(compensations)
E	::= $[] \mid E[[];t]$	(evaluation context)

■ **Figure 4** Syntax of core ContextWorkflow.

3.1 Syntax

We show the syntax of our calculus in Fig. 4. Meta-variable t ranges over context workflows; s ranges over contexts; c ranges over compensations; A and C range over atomic actions, which are commands from the underlying programming language and so not specified. (We assume only that the empty atomic action ϵ is included.) We use A for normal and C for compensation actions.

A/C is a primitive workflow consisting of a pair of a normal action A and a compensation action C . $\text{sub}(t)/C$ is a sub-workflow with a programmable compensation; if $/C$ is omitted, the empty action will be assumed. check is the context checking code that asks the current execution status. The reason why check is explicit in the syntax is to point out where context checking occurs; actually, whether check appears before or after a primitive workflow is significant – see Section 3.4 for discussions. cp is a checkpoint declaration and $\text{cp}\#E$, which does not appear in the source program, is an automatically created checkpoint declaration that records an evaluation context E , and cp is replaced by $\text{cp}\#E$ at run time.

In compensations, sub is the marker that indicates the start point of a sub-workflow; $\text{ccp}\#E$ is a checkpoint automatically installed into a compensation sequence, where E is the evaluation context that is going to be executed when this checkpoint is executed.

3.2 Big-Step Semantics

In this section, we present a big-step semantics. We use overlines to denote sequences (with appropriate delimiters). For example, \bar{c} stands for a possibly empty sequence $c_1; \dots; c_n$. We also use $\bar{c} \setminus c$ to represent the sequence obtained by removing c from \bar{c} , and similarly for other metavariables. Moreover, we use \vec{A} for a sequence of atomic actions excluding ϵ , e.g., $A_1, \dots, A_n, C_n, \dots, C_1$.

The following relations give our semantics of core ContextWorkflow:

$\langle t, E, \bar{c} \rangle \Downarrow_{\vec{A}} \langle \bar{c}' \rangle$	workflow success
$\langle t, E, \bar{c} \rangle \Uparrow_{A P S}^{\vec{A}} \langle \bar{c}', E_s \rangle$	workflow interruption
$\langle \bar{c} \rangle \Downarrow_{\vec{A}} \langle \bar{c}', E_s \rangle$	compensations success
$\langle t, \bar{c} \rangle \Downarrow_{\vec{A}} \langle \rangle$	program commit
$\langle t, \bar{c} \rangle \Downarrow_A^{\vec{A}} \langle \rangle$	program abort
$\langle t, \bar{c} \rangle \Downarrow_P^{\vec{A}} \langle \bar{c}', E_s \rangle$	program partial abort
$\langle t, \bar{c} \rangle \Downarrow_S^{\vec{A}} \langle \bar{c}', E_s \rangle$	program suspend

where “ $A|P|S$ ” means that one of these symbols (A for abort, P for partial abort, and S for suspend) comes at this position and E_s is an evaluation context. These judgments basically mean that, if the left side of $\Downarrow_{\vec{A}}$ or $\Uparrow_{\vec{A}}$ is executed, it terminates after executing \vec{A} and returns the right side, which is a sequence of compensation actions \bar{c}' possibly with a suspended computation E_s . The first two relations are for the execution of t under evaluation

context E with compensation actions \bar{c} recorded by past commands; the first relation is for successful execution and the second relation is for interrupted execution, where E_s is empty ($[\]$) in the case of abort A or partial abort P . The third relation is for the execution of compensation actions that are returned when a workflow is aborted or partially aborted. The last four relations are the main relations for execution of a program, which is τ and compensation actions \bar{c} , which are in many cases empty. If the program is committed or aborted, it returns nothing; if the program is partially aborted or suspended, then it returns compensations \bar{c}' and the evaluation context E_s . The reason why a compensation sequence is also returned is that it is used when the suspended workflow restarts; in other words, if $\langle \bar{c}, E_s \rangle$ is returned by suspension, a restart of the suspended computation can be expressed by running a program $\langle E_s [\text{check}], \bar{c} \rangle - \text{check}$ means that the restart should check the context first to check if the context allows the restart.

The semantics is defined by the rules in Fig. 5; the auxiliary function *rmsub1* to forget compensations in the nearest sub-workflow is defined as follows.

$$\begin{aligned} \text{rmsub1}(\bullet) &= \bullet \\ \text{rmsub1}(c; \bar{c}) &= \text{if } c = \text{sub, then } \bar{c} \text{ else } \text{rmsub1}(\bar{c}) \end{aligned}$$

The rule CW-PW is for the primitive workflow that performs normal action A and adds compensation C . The rules CW-CHECK-* are for **check** and one of them is chosen non-deterministically. The rule CW-CHECKPOINT is for a checkpoint, which records the current continuation E (with symbol *ccp*) to the list of compensation actions. The hole in the evaluation context is filled with $[\]$; $\text{cp}\#E$, which means that, when the recorded continuation is executed under a different context, the original continuation is recorded (CW-CHECKPOINT-REVISIT). The rule CW-SUB is for a successful sub-workflow execution, which replaces compensations in the sub-workflow with c ; CW-SUB-INT is for interrupted sub-workflow execution. Both rules also add $(\text{sub } [\])/C$ onto the stack of frames (that is, the evaluation context) before executing τ . The rules CW-SEQ-* are for sequences, which push τ_2 on the stack of frames. The rules CW-PROGRAM-* are for program execution, where CW-PROGRAM-ABORT is to run compensations except *ccp* (represented by $\bar{c}' \setminus \text{ccp}$), meaning that checkpoints are simply ignored. CW-PROGRAM-PABORT performs compensations – if they include *ccp*, compensation will stop at the first *ccp* and return the evaluation context recorded there (see CW-COMP-CCP). The rules CW-COMP* are for the execution of compensations.

An example of workflow execution is shown as follows. The derivation tree for this relation is given in Appendix A.2.

$$\begin{aligned} &\langle \text{sub}\{\text{sub}\{\tau_1; \text{cp}; \text{sub}(\tau_2)/C_a; \text{check}\}/C_b; \tau_3\}; \tau_4, \bullet \rangle \Downarrow_P^{A_1, A_2, C_a} \\ &\quad \langle C_1; \text{sub}; \text{sub}, \text{sub}(\text{sub}([\]; \text{cp}\#E_1; \text{sub}(\tau_2)/C_a; \text{check})/C_b; \tau_3); \tau_4 \rangle \\ &\text{where } \tau_k = A_k/C_k \ (1 \leq k \leq 4) \text{ and } E_1 = \text{sub}(\text{sub}([\]; \text{sub}(\tau_2)/C_a; \text{check})/C_b; \tau_3); \tau_4. \end{aligned}$$

This is an example of partial abort at the **check**; hence, an evaluation context and compensations are returned. If we would like to restart the suspended workflow, we give **check** (or ϵ/ϵ , if the initial check can be omitted) to the evaluation context. Then, restarting it may perform normal actions A_2 , A_3 , and A_4 and terminate. In other words, the relation below can be derived.

$$\langle \text{sub}(\text{sub}(\text{check}; \text{cp}\#E_1; \text{sub}(\tau_2)/C_a; \text{check})/C_b; \tau_3); \tau_4, C_1; \text{sub}; \text{sub} \rangle \Downarrow^{A_2, A_3, A_4} \langle \rangle$$

$\frac{}{\langle A/C, E, \bar{c} \rangle \Downarrow^A \langle C; \bar{c} \rangle}$	(CW-PW)
$\frac{}{\langle \text{check}, E, \bar{c} \rangle \Downarrow^\epsilon \langle \bar{c} \rangle}$	(CW-CHECK-CONT)
$\frac{}{\langle \text{check}, E, \bar{c} \rangle \Uparrow_S^\epsilon \langle \bar{c}, E \rangle}$	(CW-CHECK-SUSPEND)
$\frac{}{\langle \text{check}, E, \bar{c} \rangle \Uparrow_A^\epsilon \langle \bar{c}, [] \rangle}$	(CW-CHECK-ABORT)
$\frac{}{\langle \text{check}, E, \bar{c} \rangle \Uparrow_P^\epsilon \langle \bar{c}, [] \rangle}$	(CW-CHECK-PABORT)
$\frac{}{\langle \text{cp}, E, \bar{c} \rangle \Downarrow^\epsilon \langle \text{ccp}\#E[[]]; \text{cp}\#E; \bar{c} \rangle}$	(CW-CHECKPOINT)
$\frac{}{\langle \text{cp}\#E_0, E, \bar{c} \rangle \Downarrow^\epsilon \langle \text{ccp}\#E_0[[]]; \text{cp}\#E_0; \bar{c} \rangle}$	(CW-CHECKPOINT-REVISIT)
$\frac{}{\langle t, E[(\text{sub } [])/C], \text{sub}; \bar{c} \rangle \Downarrow^{\vec{A}} \langle \bar{c}' \rangle}$	(CW-SUB)
$\frac{}{\langle \text{sub}(t)/C, E, \bar{c} \rangle \Downarrow^{\vec{A}} \langle C; \text{rmsub1}(\bar{c}') \rangle}$	
$\frac{}{\langle t, E[(\text{sub } [])/C], \text{sub}; \bar{c} \rangle \Uparrow_*^{\vec{A}} \langle \bar{c}', E_s \rangle}$	(CW-SUB-INT)
$\frac{}{\langle \text{sub}(t)/C, E, \bar{c} \rangle \Uparrow_*^{\vec{A}} \langle \bar{c}', E_s \rangle}$	
$\frac{}{\langle t_1, E[[]; t_2], \bar{c} \rangle \Downarrow^{\vec{A}_1} \langle \bar{c}' \rangle \quad \langle t_2, E, \bar{c}' \rangle \Downarrow^{\vec{A}_2} \langle \bar{c}'' \rangle}$	(CW-SEQ)
$\frac{}{\langle t_1; t_2, E, \bar{c} \rangle \Downarrow^{\vec{A}_1; \vec{A}_2} \langle \bar{c}'' \rangle}$	
$\frac{}{\langle t_1, E[[]; t_2], \bar{c} \rangle \Uparrow_*^{\vec{A}} \langle \bar{c}', E_s \rangle}$	(CW-SEQ-INT1)
$\frac{}{\langle t_1; t_2, E, \bar{c} \rangle \Uparrow_*^{\vec{A}} \langle \bar{c}', E_s \rangle}$	
$\frac{}{\langle t_1, E[[]; t_2], \bar{c} \rangle \Downarrow^{\vec{A}_1} \langle \bar{c}' \rangle \quad \langle t_2, E, \bar{c}' \rangle \Uparrow_*^{\vec{A}_2} \langle \bar{c}'', E_s \rangle}$	(CW-SEQ-INT2)
$\frac{}{\langle t_1; t_2, E, \bar{c} \rangle \Uparrow_*^{\vec{A}_1; \vec{A}_2} \langle \bar{c}'', E_s \rangle}$	
$\frac{}{\langle t, [], \bar{c} \rangle \Downarrow^{\vec{A}} \langle \bullet \rangle}$	(CW-PROGRAM-COMMIT)
$\frac{}{\langle t, \bar{c} \rangle \Downarrow^{\vec{A}} \langle \rangle}$	
$\frac{}{\langle t, [], \bar{c} \rangle \Uparrow_A^{\vec{A}} \langle \bar{c}', [] \rangle \quad \langle \bar{c}' \setminus \text{ccp} \rangle \Downarrow^{\vec{c}} \langle \bullet, [] \rangle}$	(CW-PROGRAM-ABORT)
$\frac{}{\langle t, \bar{c} \rangle \Downarrow_A^{\vec{A}; \vec{c}} \langle \rangle}$	
$\frac{}{\langle t, [], \bar{c} \rangle \Uparrow_P^{\vec{A}} \langle \bar{c}', [] \rangle \quad \langle \bar{c}' \rangle \Downarrow^{\vec{c}} \langle \bar{c}'', E_s \rangle}$	(CW-PROGRAM-PABORT)
$\frac{}{\langle t, \bar{c} \rangle \Downarrow_P^{\vec{A}; \vec{c}} \langle \bar{c}'', E_s \rangle}$	
$\frac{}{\langle t, [], \bar{c} \rangle \Uparrow_S^{\vec{A}} \langle \bar{c}', E_s \rangle}$	(CW-PROGRAM-SUSPEND)
$\frac{}{\langle t, \bar{c} \rangle \Downarrow_S^{\vec{A}} \langle \bar{c}', E_s \rangle}$	
$\frac{}{\langle C \rangle \Downarrow^c \langle \bullet, [] \rangle}$	(CW-COMP-ACTION)
$\frac{}{\langle \text{sub} \rangle \Downarrow^\epsilon \langle \bullet, [] \rangle}$	(CW-COMP-SUB)
$\frac{}{\langle c \rangle \Downarrow^{\vec{c}} \langle \bullet, [] \rangle \quad \langle \bar{c} \rangle \Downarrow^{\vec{c}'} \langle \bullet, [] \rangle}$	(CW-COMP-SEQ)
$\frac{}{\langle c; \bar{c} \rangle \Downarrow^{\vec{c}; \vec{c}'} \langle \bullet, [] \rangle}$	
$\frac{}{\langle \text{ccp}\#E \rangle \Downarrow^\epsilon \langle \bullet, E \rangle}$	(CW-COMP-CCP)
$\frac{E_s \neq []}{\langle c \rangle \Downarrow^{\vec{c}} \langle \bullet, E_s \rangle}$	
$\frac{}{\langle c; \bar{c} \rangle \Downarrow^{\vec{c}} \langle \bar{c}, E_s \rangle}$	(CW-COMP-SEQ-PABORT1)
$\frac{E_s \neq []}{\langle c \rangle \Downarrow^{\vec{c}_1} \langle \bullet, [] \rangle \quad \langle \bar{c} \rangle \Downarrow^{\vec{c}_2} \langle \bar{c}', E_s \rangle}$	(CW-COMP-SEQ-PABORT2)
$\frac{}{\langle c; \bar{c} \rangle \Downarrow^{\vec{c}_1; \vec{c}_2} \langle \bar{c}', E_s \rangle}$	

■ Figure 5 Big step semantics of core ContextWorkflow.

3.3 Properties

Here, we state some properties that hold of the semantics. The main aim of this section is to rigorously give the specification to the language. In particular, giving specifications about suspension and partial aborts (checkpoints) is important since these are unusual in the context of workflow languages.

In the following theorems, let $p_k = A_k/C_k$ for some k , and we define a function $b(\tau)$ and predicates *includes* and *nosub* as follows.

- Let $b(\tau)$ be a workflow obtained from τ by removing **sub**, **check**, **cp** and **cp#E** from τ .
- $includes(\tau, m, n)$ iff $b(\tau) = p_m; \dots; p_n$ and $m \leq n$; or τ has no primitive workflows and $m \not\leq n$.
- $includes(E, m, n) = includes(E[check], m, n)$.
- $includes(\bar{c}, m, n)$ iff $\bar{c} \setminus \{\text{sub}, \text{ccp}\#E\} = C_m; \dots; C_n$ and $m \geq n$; or \bar{c} has no atomic actions C_* and $m \not\leq n$.
- $nosub(\tau, m, n)$ iff $includes(\tau, m, n)$ and τ has no **sub**-workflow.

Theorems 1 and 2 state about the behaviors under contexts **Continue** and **Abort**. These are the basic properties of Sagas [9].

► **Theorem 1** (Workflow commits). *If $includes(\tau, m, n)$ and $\langle \tau, \bullet \rangle \Downarrow_{\vec{A}} \langle \rangle$ and $m \leq n$, then $\vec{A} = A_m, \dots, A_n$.*

► **Theorem 2** (Workflow aborts (Successful Compensation)). *If $nosub(\tau, m, n)$ and $\langle \tau, \bullet \rangle \Downarrow_{\vec{A}} \langle \rangle$ and $m \leq n$, then $\vec{A} = A_m, \dots, A_i, C_i, \dots, C_m$ for some i ($m \leq i \leq n$), or $\vec{A} = \epsilon$.*

Theorem 3 states that, even though a workflow is suspended in the middle by **Suspend**, the resulting normal actions after its final commit are always the same. Therefore, it ensures that a suspended workflow actually continues from the suspension point.

► **Theorem 3** (Restarted suspended workflow commits). *If $\langle \tau, \bullet \rangle \Downarrow_{\vec{A}} \langle \rangle$ and $\langle \tau, \bullet \rangle \Downarrow_S^{\vec{A}} \langle \bar{c}, E \rangle$ and $\langle E[check], \bar{c} \rangle \Downarrow_{\vec{A}'} \langle \rangle$, then $\vec{A} = \vec{A}', \vec{A}'$.*

Theorem 4 states that if a workflow suspends at a checkpoint by **PAabort**, it surely did compensations corresponding to completed normal actions successive to the checkpoint; moreover, the suspended workflow actually points to the continuation from the checkpoint.

► **Theorem 4** (Workflow partially aborts). *If $nosub(\tau, m, n)$ and $\langle \tau, \bullet \rangle \Downarrow_P^{\vec{A}} \langle \bar{c}, E \rangle$ and $m \leq n$, then either of the followings hold.*

- $\vec{A} = A_m, \dots, A_i, C_i, C_{i-1}, \dots, C_j$ and $includes(E, j, n)$ and $includes(\bar{c}, j-1, m)$ for some i and j ($m \leq j \leq i \leq n$).
- $\vec{A} = A_m, \dots, A_n$ and $includes(E, 1, 0)$ and $includes(\bar{c}, n, m)$.

Moreover, the following conditions hold.

1. (Suspended workflow commits) *If $\langle E[check], \bar{c} \rangle \Downarrow_{\vec{A}'} \langle \rangle$, then $\vec{A}' = A_j, \dots, A_n$, or $\vec{A}' = \epsilon$ (if $includes(E, 1, 0)$).*
2. (Suspended workflow aborts) *If $\langle E[check], \bar{c} \rangle \Downarrow_A^{\vec{A}'} \langle \rangle$, then $\vec{A}' = A_j, \dots, A_k, C_k, \dots, C_m$ for some k ($j \leq k \leq n$) or $\vec{A}' = C_{j-1}, \dots, C_m$.*

Theorem 5 provides the properties about a complex workflow including a sub-workflow, checkpoints and **PAabort**; it describes that a completed sub-workflow is skipped at the compensation time and a suspended workflow remembers the original program structure including checkpoints and the sub-workflow.

► **Theorem 5** (Partial abort, checkpoint and nested workflow). *Suppose that $\text{includes}(\mathfrak{t}, 1, n)$ and \mathfrak{t} without check is*

$p_1; \dots; cp; p_k; \dots; p_m; \text{sub}(p_{m+1}; \dots; cp; p_j; \dots; p_l) / C_a; p_{l+1}; \dots; p_n$ and $\langle \mathfrak{t}, \bullet \rangle \Downarrow_{\vec{A}}^{\vec{A}} \langle \bar{c}, E \rangle$.

1. (Partial abort skips compensations of complete sub-workflows) *If $A_{l+1} \in \{\vec{A}\}$, then $\vec{A} = A_1, \dots, A_i, C_i, \dots, C_{l+1}, C_a, C_m, \dots, C_k$ for some $i > l$.*
2. (A suspended workflow remembers checkpoints in complete sub-workflows) *If $A_{l+1} \in \{\vec{A}\}$ and $\langle E[\text{check}], \bar{c} \rangle \Downarrow_{\vec{A}}^{\vec{A}} \langle \bar{c}', E' \rangle$ and $A_j \in \{\vec{A}'\} \wedge A_l \notin \{\vec{A}'\}$, then $\vec{A}' = A_k, \dots, A_i, C_i, \dots, C_j$ for some i such that $(j \leq i \leq l)$.*
3. (A suspended workflow remembers checkpoints before a sub-workflow) *If $C_j \in \{\vec{A}\}$ and $\langle E[\text{check}], \bar{c} \rangle \Downarrow_{\vec{A}}^{\vec{A}} \langle \bar{c}', E' \rangle$ and $A_{l+1} \in \{\vec{A}\}$, then $\vec{A}' = A_j, \dots, A_i, C_i, \dots, C_{l+1}, C_a, C_m, \dots, C_k$ for some $i > l$.*

For the robot example, the first and the third items of Theorem 5 are significant; otherwise, the robot would move back to the whole path at the compensation time, and forget checkpoints. The second item is important in an example that needs re-calculation of a complete sub-workflow.

3.4 Discussion

Design Choice of Primitive Workflow with Context Checking. Although A/C is the primitive workflow in the calculus, it does not appear explicitly in the DSL. We regard A/C preceded by `check` as a primitive workflow and give another notation $A /+ C$ in the DSL, representing asynchronous interruption. Actually, another interpretation of $A /+ C$ would be to put `check` after A/C . The difference between these interpretations becomes clear when executing a sub-workflow. Let $\mathfrak{t}_k = A_k /+ C_k$ for $k = 1, 2$. Then, when we execute $\text{sub}(\mathfrak{t}_1; \mathfrak{t}_2)$, is it possible that the resulting action sequence A_1, A_2, C_2, C_1 appears? In the former choice (where $A/+C$ is `check; A/C`), such a result never occurs – possible sequences of actions are only \bullet , “ A_1, C_1 ”, or “ A_1, A_2 ” – while it may in the latter.

The former choice is looser than the latter in the sense that the whole execution may commit after the execution though context checking actually occurs during the execution of an atomic action. Such a behavior is critical in cases where an atomic action must be performed in the `Continue` context. For example, suppose that a workflow contains an atomic action to download something and the context relates to network availability; then the atomic action must commit only at the time when it is executed in the `Continue` context; otherwise, the downloaded file would be incomplete. Therefore, we can regard the latter choice as transactions.

Since we suppose that many context-aware applications such as robots are not strict, in our implementation, we adopt the former choice by default. Fortunately, we can switch between both semantics easily.

Atomic and nonatomic blocks. It is easy to extend with `atomic` and `nonatomic`. Their semantics is similar to sub-workflows and they basically control non-determinism in `check`.

Abnormal termination. We can consider *abnormal termination* [9], a stronger notion of abort that occurs when an atomic action (or a compensation action) fails without even performing any compensation. Though we do not include abnormal termination here, it is not difficult to add it; it is enough to add nondeterminism to rules CW-PW and CW-COMP-ACTION and the other relation for the abnormal signal. Later, we implement abnormal termination in the E-DSL, by using exceptions in Scala.

Differences with respect to the calculus [9]. Here, we describe the differences with respect to the existing calculus [9], by which ours are inspired.

- Ours adds the notions of checkpoint, partial abort, and suspension. Technically, our semantics introduces evaluation contexts in order to capture continuations of workflow executions.
- Ours omits abnormal termination and does not model parallelism.
- In ours, an abort inside a nested workflow results in an abort of the parent workflow. Although this design choice is not usual [12] (where our choice is referred to as *upward abortion propagation*), we intend that an abort signal means it is signaled to the whole workflow, because the workflow is executed on a single thread.

4 Monadic embedding to Scala

Our approach to implementing ContextWorkflow is to embed the language into another language. We use a free monad transformer for representing and building the abstract syntax trees and define a monadic interpreter that follows the semantics in Section 3.

There are two differences between the core calculus and the embedding, though they closely correspond with each other. First, the `sub-block` is represented by two marks in the embedding, to indicate the beginning and the end of a block. Second, the semantics of `check` is deterministic in the embedding while it is nondeterministic in the core calculus. Our interpreter checks the context when evaluating `check` and chooses one branch. We represent the context by a stream of `Context`, which is essentially the same as the signal of `Context` in Section 2.

The underlying monad of our free monad transformer is a combination of an exception monad and a reader monad. The exception monad represents aborts, partial aborts, and suspensions. The reader monad keeps the context that is checked when `check` is evaluated. In other words, we develop ContextWorkflow on top of a monadic language that supports exceptions and readable environments. The monadic interpreter translates ContextWorkflow programs to monadic programs.

The main contribution of this section is (1) a simple implementation, i.e., clear correspondence between the semantics and implementation, and (2) efficiency in eager languages. A naive approach would be to extend the compensation monad [47], but it is hard to make such an extension simple. See Section 5 for a detailed discussion.

We use Scala as the language for demonstration and explanation. Although our implementation in Scala heavily relies on scalaz [1], we here show language/library-independent definitions for comprehensibility and generality.

4.1 Free monad transformers

This section gives a brief introduction to the free monad transformers along with the basic definitions and notations for monadic programming in Scala. Readers who would like to learn about monads and monadic programming are referred to other papers [43, 58]. Most of the definitions are simplified; although scalaz uses implicit conversions to use objects as functors and monads, here we define functors and monads using simple inheritance.

A free monad transformer `FreeT[F, M, _]` is a monad that is freely constructed from the given functor `F` and underlying monad `M`. One can understand free monad transformers as abstract syntax trees and therefore the functor `F` defines the “commands” of the language. The difference from free monads is that the nodes are some computations of which semantics is given by the underlying monad.

2:16 ContextWorkflow

Functors and monads are defined by the traits `Functor` and `Monad`, respectively. Free monad transformers are defined by the abstract class `FreeT`. `Functor` provides `map` (`fmap` in Haskell) and `Monad` provides `flatMap` (`>=>` in Haskell) and `point` (`return` in Haskell).

```
trait Functor[F[_]]{ def map[A,B](f: A => B): F[B] }
trait Monad[M[_]] extends Functor[M]{
  def flatMap[A,B](f: A => M[B]): M[B]
  def point[A](a: => A): M[A] }
```

Because `Monad` provides `flatMap`, we can use the `for`-comprehension in Scala, similarly to the `do`-notation in Haskell. For example, for values `m1` and `m2` of the type monad `M`, the code

```
for{ a <- m1 ; b <- m2 } yield a + b
```

is equivalent to the following code.

```
m1.flatMap(a => m2.map(b => a + b))
```

`FreeT` is defined using the auxiliary trait `FreeF` and provides the two functions `iterT` and `interpretS`.³ Intuitively, `FreeT` is a list-like structure and `iterT` works as `foldr` over lists. `interpretS` replaces the “commands” of the language with other “commands.”

```
class FreeT[F[_],M[_],A](run: M[FreeF[F,A,FreeT[F,M,A]]])
  extends Monad[FreeT[F,M,?]]{
  def iterT(interp: F[M[A]] => M[A]): M[A]
  def interpretS[G[_]](st: ~>[F,G]): FreeT[G,M,A]
}
```

`iterT` takes an interpretation of “commands” and translates a “program” of type `FreeT[F,M,A]` to that of `M[A]`. `interpretS` takes a natural transformation from the functor `F` to another functor `G` and translates a “program” of type `FreeT[F,M,A]` into that of type `FreeT[G,M,A]`. The question mark `?` in a type parameter means that a surrounding expression is a type-level anonymous function, e.g., `M[A,?]` takes one type parameter and `M[A,?,?]` takes two.⁴

The trait `FreeF` takes three types `F`, `A`, and `B` and has two constructions `Pure` and `Free`. `F` is the functor that defines “commands.” `Pure` lifts a pure value of type `A` to the “program” represented by the free monad transformer. `Free` lifts a “command” followed by a computation of type `B` to the “program.”

```
trait FreeF[F[_],A,B]
case class Pure[F[_],A,B](a:A) extends FreeF[F,A,B]
case class Free[F[_],A,B](fb: F[B]) extends FreeF[F,A,B]
```

4.2 ContextWorkflow Monad

The `ContextWorkflow` monad `CW` is a free monad transformer defined as:⁵

³ Here we borrow `iterT` from the `free` package of Haskell. Although `iterT` can be defined in Scala, it is not good in practice. We will visit the problem in Sec 4.6.

⁴ This feature is enabled by `kind-projector` (<https://github.com/non/kind-projector>).

⁵ Again, the definition is simplified from the actual definition just for avoiding unnecessary complexity of implicit conversions.

```

case class CW[E,M[_],S,A] (
  run: FreeT[CWT[M,S,?], EitherT[ReaderT[M,Sig,?], InSubL[EV[M[E],S]],?], A])
extends Monad[CW[E,M,S,?]] { /* map, point and flatMap */ }

```

The type parameter E is for the exception type; M is for the monad that represents effects in the atomic actions; S is for the suspended workflow type (explained later); and A is for the successful result value type. Sig is the type of the context, which is just an alias of `Stream[Context]`. A `Context` is either `Continue`, `Abort`, `PAbort` or `Suspend`, which are objects that extend `Context`. `EV` is the type of exceptional values that consists of the compensation actions to be executed and the suspended workflow. `InSubL` keeps track of the depth of the sub-block to skip compensation actions. We call `EitherT[ReaderT[M,Sig,?], InSubL[EV[M[E],S]], ?]` the underlying monad of `CW[E,M,S,A]` in the rest of the paper.

`CWT` represents the “commands” of `ContextWorkflow`. Concrete commands and `CWT` are defined as follows.

```

trait CWT[M[_],S,A] extends Functor[CWT[M,S,?]] { /* map */ }
case class Comp[M[_],S,A] (comp:M[Unit], a:A) extends CWT[M,S,A]
case class SubB[M[_],S,A] (a:A) extends CWT[M,S,A]
case class SubE[M[_],S,A] (a:A) extends CWT[M,S,A]
case class Cp[M[_],S,A] (a:A) extends CWT[M,S,A]
case class Cpn[M[_],S,A] (s:S,a:A) extends CWT[M,S,A]
case class Check[M[_],S,A] (a:A) extends CWT[M,S,A]

```

M is a monad for atomic actions; S is the type of a suspended workflow that corresponds to the evaluation contexts in the calculus. `Comp` is for specifying compensation action. `SubB` and `SubE` are the beginning and end marks of a sub-block, respectively. `Cp` and `Cpn` are checkpoints that correspond to `cp` and `cp#E` in the calculus, respectively. `Cpn` has a suspended workflow, which corresponds to the fact that `cp#E` has an evaluation context E . `Check` corresponds to `check` in the calculus.

One may wonder why we do not have a command for normal actions while we have one for compensation actions. This is because the normal actions of type $M[A]$ are handled by the underlying monad `EitherT[ReaderT[M,...],...]` of the free monad transformer.

The exception type `EV` consists of three constructors as follows:

```

sealed trait EV[ME,S]
case class Aborting[ME,S] (e:ME) extends EV[ME,S]
case class Suspending[ME,S] (s:S) extends EV[ME,S]
case class PAborting[ME,S] (s:Option[S],e:ME) extends EV[ME,S]

```

The type parameter ME is for the type of compensation actions. `Aborting` represents that the workflow is aborted. The field `e` keeps the compensations to be executed. `Suspending` represents that the workflow is suspended. The field `s` keeps the suspended workflow of type S . `PAborting` represents that the workflow is partially aborted. The suspended workflow `s` is optional because a workflow may not have a checkpoint and in that case, there is no suspended workflow.

`InSubL` represents whether the workflow execution is in the sub-block or not.

```

sealed trait InSubL[A]
case class InSub[A] (n:InSubL[A]) extends InSubL[A]
case class NonSub[A] (a:A) extends InSubL[A]

```

InSub and NonSub represent that the workflow execution is in a sub-workflow and not, respectively. Notice that only executions of compensation actions are changed by sub-workflows and programmable compensations. It is therefore sufficient to wrap only the exceptional values propagated backwards with InSubL.

Readers may wonder what CW[A] that appeared in Section 2 is. This abbreviates CW[Unit,IO,Nothing,A]; see Appendix A.1 for further details.

4.3 Auxiliary Definitions

This section gives the auxiliary functions and macros that correspond to the syntax for the users of ContextWorkflow. For readability and simplicity, we omit the type and implicit arguments of method invocations necessary to compile if they are clear from the context.

The functions `check` and `checkpoint` correspond to `check` and `cp` in the calculus, respectively.

```
def check[E,M[_],S]: CW[E,M,S,Unit] = CW(liftF(Check(())))
def checkpoint[E,M[_],S]: CW[E,M,S,Unit] = CW(liftF(Cp(())))
```

`liftF` lifts the objects of type `F[A]` for any functor `F` and type `A` to a free monad transformer `FreeT[F,M,A]` for any monad `M`.

The primitive workflow `A/C` in the calculus is written as `compL(A,C)` where `compL` is an auxiliary function defined as follows:

```
def compL[E,M[_],S,A](na:M[A])(ca:A => M[Unit]): CW[E,M,S,A] = CW{
  na.liftM.liftM.liftM.flatMap(x => liftF(Comp(ca(x),x))) }
```

`liftM` lifts the monadic values of type `G[A]` to another monadic value of type `H[G,A]` where `G` and `H` are a monad and a monad transformer, respectively. We also define another auxiliary function `/+` that corresponds to `check;A/C`⁶.

```
def /+[E,M[_],S,A](na:M[A])(ca:A => M[Unit]): CW[E,M,S,A] =
  check.flatMap(_ => compL(na)(ca))
```

For the programmable compensations and sub-workflows, we define the two auxiliary functions `subC` and `sub`, respectively. `subC` takes a workflow and a compensation and `sub` takes only a workflow. `sub` concatenates the beginning mark of the block, the given workflow, and the end mark of the block. `subC` additionally concatenates the sub-workflow created from the given workflow and the given compensation action.

```
def sub[E,M[_],S,A](cw :CW[E,M,S,A]): CW[E,M,S,A] = CW{ for{
  _ <- liftF(SubB())
  r <- cw.run
  _ <- liftF(SubE())
} yield r }
def subC[E,M[_],S,A](cw :CW[E,M,S,A])(ca :A => M[Unit]): CW[E,M,S,A] = CW{
  sub(cw).flatMap(r => liftF(Comp(ca(r),r))) }
```

We also define two macros `!!` and `workflow` using the Monadless [2] library. The macro `!!` takes a workflow and escapes it from the program transformation. The macro `workflow` works as a block that specifies the target area of the program transformation. Assignments

⁶ Though omitted here, to regard `/+` as an infix operator, we have to define it using implicit conversions in Scala.

and sequential compositions in `workflow` are transformed into a chain of monadic binds. For example,

```
workflow { val x = !(w1); val y = !(w2); x + y }
```

is transformed into

```
w1.flatMap(x => w2.map(y => x + y))
```

4.4 Types of Suspended Workflows

Before showing the monadic interpreter for the `CW` monad, we need to fix the type of the suspended workflows. Clearly, it must be equal to the type of the workflow to be executed, i.e., S in $CW[E, M, S, A]$ must be again $CW[E, M, S, A]$. This means that S is a fixpoint of the functor $CW[E, M, ?, A]$ [23, 45]. The data type `Fix` is parameterized over functors

```
case class Fix[F[_]](out: F[Fix[F]])
```

and the type of suspended workflows is represented as `Fix[CW[E, M, ?, A]]`.

4.5 Monadic interpreter

Our monadic interpreter of the `CW` language is the function `runCWT` from, for any monad M and type A , $CW[Unit, M, Fix[CW[Unit, M, ?, A]], A]$, which is equal to $Fix[CW[Unit, M, ?, A]]$, to $MM[A]$ where MM is the underlying monad defined as follows.

```
def runCWT[M[_], A](s: Fix[CW[Unit, M, ?, A]])
: EitherT[ReaderT[M, Sig, ?], InSubL[EV[M[Unit], Fix[CW[Unit, M, ?, A]]], A] = {
  type S = Fix[CW[Unit, M, ?, A]] // the type of suspended workflows
  type R = EV[M[Unit], S] // the type of exceptional results
  type F[X] = CWT[M, S, X] // the term functor
  type MM[X] = EitherT[ReaderT[M, Sig, ?], InSubL[R], X] // the underlying monad

  def runCWT0(c1: F[MM[A]]): MM[A] = c1 match{
    case Comp(c, k) => ...
    ...
  }
  s.out.run.iterT(runCWT0)
}
```

The function `runCWT0` translates each command of the `CW` language defined by `CWT` to the program of the language given by the underlying monad MM . Because the translation proceeds from the last terms to the first terms by `iterT`, each command object has the subsequent translated program. In other words, the result of the rest of the workflow is always available.

The interpretation of `Check` follows `CW-CHECK-*`. It installs a context check to the resulting program. If the context is `Continue`, it returns the result of the subsequent program. It otherwise throws exceptions. Note that the exceptions are just the values of type `EitherT[...]`, that is the underlying monad, and we do not use the exception handling mechanism of Scala.

```
case Check(k) => { // k: EitherT[ReaderT[M, Sig, ?], InSubL[R], A]
  ask.liftM.flatMap{ sig =>
    sig.head match {
      case Abort => raiseException(InSubL.point(Aborting(M.point(()))))
    }
  }
}
```

```

case PAbort => raiseException(InSubL.point(PAborting(None, M.point(()))))
case Suspend => raiseException(InSubL.point(Suspending(
  Fix(CW(FreeT.roll(Check(k.liftM)))))) // creates the suspended workflow
case Continue => local(_.tail)(k)
}}

```

k is the interpretation of the subsequent workflow. The method `ask` gets a value from the environment. In our case, they are the context that is represented by the streams of type `Stream[Context]`. The variable `sig` is bound to a stream. If the head, which represents the current context, is `Abort`, `Aborting` of point of the unit value is thrown. This is because there is no compensation to be executed at this point. If the current context is `PAabort`, `PAborting` of `None` and `point` of the unit value is thrown. If the current context is `Suspend`, we throw the translated program k as the suspended workflow. If the current context is `Continue`, we drop the head of the stream and continue interpreting the workflow.

The interpretation of `Comp` corresponds to `CW-SEQ-INT-*`, `CW-PROGRAM-*`, `CW-COMP-ACTION` and `CW-COMP-SEQ-*`. The parameters `comp` and k are the compensation action and the interpretation of the rest of the workflow, respectively.

```

case Comp(comp, k) => EitherT {
  k.run.map{ ev => ev match {
    case \-(_) => ev // successful execution
    case -\/(err) =>
      extendSuspending(liftF(Comp(c, ())))(err) match { // at compensation
        case NonSub(p) => p match { // binding compensation
          case Aborting(cp) => \.left(NonSub(Aborting(cp.flatMap(res => comp.flatMap(_ => M
            .point(res))))))
          case PAborting(None, cp) => \.left(NonSub(PAborting(None, cp.flatMap(res => comp.
            flatMap(_ => M.point(res))))))
          case Suspending(sp) => \.left(NonSub(Suspending(sp)))
        }
      }
    case x => \.left(x) // skipping compensation of a complete sub-workflow
  }}}

```

If the result of the subsequent workflow is an exception, the interpreter adds the compensation command `Comp(c, ())` at the head of the suspended workflow in `err` by `extendSuspending`. Following the operational semantics, we skip the compensation actions that (1) are in sub-workflows and (2) are followed by a checkpoint that is not in any sub-workflow and the execution is partially aborted after executing the checkpoint. The first condition is represented by `InSubL`. The last condition is represented by `Option`.

Following `CW-CHECKPOINT` and `CW-COMP-CCP`, the interpretation of `Cp` (1) puts the command represented by `Cpn` at the head of the suspended workflow and (2) puts a suspended workflow to the exception if it is of type `PAborting`. The suspended workflow that corresponds to E of `cp#E` and `ccp#E` is just the argument of `Cp`.

```

case Cp(k) => EitherT {
  k.run.map{r => r match {
    case \-(_) => r
    case -\/(err) => {
      val s = Fix(CW(k.liftM))
      val kp = liftF(Cpn(s, ())) // creates Cpn that is substituted for the Cp
      \.left(setPAabort(s)(extendSuspending(kp)(err))) // set pabort with suspension
    }
  }}}

```

`s` is the suspended workflow. The function `setPAbort` merely replaces the first parameter of `PAborting` with `s` if it is `None`. The interpretation of `Cpn` is similar.

The interpretations of `SubB` and `SubE` just remove and add `InSub` layers in the exceptional values, respectively.

4.6 Stack Safety

Implementations of free monad transformers in eager languages usually need some care to avoid stack overflow (so-called stack safety) and do not provide `iterT`. Instead, they provide a “foldl variant” of `iterT` [21], namely `runFreeT` in Purescript and `runM` in scalaz, which takes a function from `F[FreeT[F,M,A]]` to `M[FreeT[F,M,A]]` and returns a value of type `M[A]` for any functor `F`, monad `M` and type `A`.

It is necessary to know whether the subsequent workflow is interrupted or not to perform compensation actions. We use continuation monads to achieve this as the compensation monad [47]. We wrap the underlying monad of `CW` with a continuation monad transformer `ContT`.⁷

```
case class CW[E,M[_],S,R,A](
  run: FreeT[CWT[M,S,?],
  ContT[EitherT[ReaderT[M,Sig,?], InSubL[EV[M[E],S]],?], R, ?],
  A)
extends Monad[CW[E,M,S,R,?]] { /* map, point and flatMap */ }
```

The function `runCWTO` for `runM` takes a command followed by an uninterpreted workflow and returns a continuation monad transformer followed by the workflow left uninterpreted.

```
def runCWT[M[_],R,A](s: Fix[CW[Unit,M,?,R,A]]) = {
  type S = Fix[CW[Unit, M, ?, R, A]]
  type F[X] = CWT[M, S, X]
  type MM[X] = ContT[EitherT[ReaderT[M, Sig, ?], InSubL[EV[M[Unit], S]], X], R, X]
  def runCWTO[M[_],R,A](c1: F[FreeT[F, MM, A]]): MM[FreeT[F, MM, A]]
  = ...
}
```

The change on the definition of `runCWTO` is straightforward. All we need to do is just wrap the exception monad transformer with the continuation monad transformer. For example, the interpretation of the command `CompL` is defined as follows.

```
case Comp(comp, k) = ContT{knt =>
  EitherT{
    knt(k).run.map{ev => ev match {
      ... /* the same to the previous definition */
    }}}}
```

4.7 Atomicity

In this section, we extend `CWT` and the `CW` monad to support the `atomic` and `nonatomic` blocks.

⁷ The continuation monad transformer must be stack safe. Unfortunately, neither scalaz nor cats (another library similar to scalaz) provides it. Our Scala implementation employs a workaround that relies on `Trampoline` [8] in the `IO` monad. In other words, we always use the `IO` monad as the underlying user monad of the `CW` monad.

We add a command `CheckA` for active context checking and `CheckI` for inactive context checking, whose definitions are similar to that of `Check`.

```
case class CheckA[M[_], S, A](a: A) extends CWT[M, S, A]
case class CheckI[M[_], S, A](a: A) extends CWT[M, S, A]
```

The interpretation of `CheckA` is similar to that of `Check` and that of `CheckI` is just continuing the evaluation of the subsequent workflow without checking the context.

The two blocks are implemented as two functions, similarly to how `sub` sub-workflows are implemented. The functions `atomic` and `nonatomic` replace `Check` with `CheckI` and `CheckA`, respectively, as follows.

```
def atomic[E, M[_], S, A](cw: CW[E, M, S, A]) : CW[E, M, S, A] = CW {
  cw.run.interpretS[CWT[M, S, ?]](new (~>[CWT[M, S, ?]], CWT[M, S, ?])) {
    def apply[A](c: CWT[M, S, A]): CWT[M, S, A] = c match {
      case Check(a) => CheckI(a)
      case _ => c
    }
  }
}
```

4.8 Abnormal Termination and Exceptions in Scala

We have already mentioned abnormal termination in Section 3. In our implementation in Scala, abnormal termination is realized by exceptions of the language. Basically, if an exception is thrown in an atomic action, the whole execution stops. However, we sometimes want to convert an exception in normal action to context, and it can be done using a new form of primitive workflow (*normal /~ compensation*). This is mostly the same as `/+`, but absorbs some particular exceptions `AbortE` and `PAbortE` in the normal action, and raises the interruption `Abort` or `PAbsort`. For example:

```
trait CWException extends Exception
class AbortE extends CWException
class PAbortE extends CWException
val cw0 = {if(...) "success" else throw e} /~ comp
```

When running `cw0`, if the exception `e` is `AbortE` or `PAbortE`, it will abort or partially abort; otherwise, the exception is raised as usual. In both cases, it does not do the corresponding compensation `comp`.

`/~` is defined as follows.

```
def /~[E, M[_], S, A](na: M[Try[A]])(comp: A => M[Unit]): CW[E, M, S, A] = for {
  tried <- compl(na)(_ match {
    case Success(a) => comp(a) // same as /+
    case Failure(e) => M.point(()) // skip the compensation comp
  })
  a <- tried match {
    case Failure(AbortE) => throwCWException(Abort) // raise abort
    case Failure(RestartE) => throwCWException(PAbort) // raise pabort
    case Success(a) => compl(M.point(a))(_ => M.point(())) // same as /+
    case Failure(e) => compl(M.point[A]{throw e})(_ => M.point(())) // rethrowing e
  } yield a
```

The argument `na` is of the type `M[Try[A]]`. `Try[T]` is a Scala's class that represents a computation that may either result in an exception (`Failure[T]`) or return a successfully computed value (`Success[T]`). What `/~` does is first binding the result of `compl` to `tried` of

the type `Try[A]` and then carry out one of the following: (1) raising `Abort` or `PAAbort` inside `ContextWorkflow`, (2) successfully committing `na`, or (3) throwing the exception `e` of Scala.

Readers may wonder that the type of `/~` (and also `/+`) is different from that of actual use in examples so far. To omit the explicit type constructors of `M` and `Try`, we use implicit conversions. For further details, see Appendix A.1.

5 Related Work

This work is the direct descendant of our previous work [32]. The main differences between the two are the monadic interpreter, a formalization of semantics, the realization of suspension and checkpoint, and advanced implementation.

Context-Oriented Programming. The literature on context-oriented programming [30], which advocates the use of layers to modularize context-dependent behavior, includes several reports on behavioral change in response to asynchronous context changes [33, 57, 7]. Among them, the closest to the present work is Flute [7] in that it supports interruptible context-dependent execution. Interruptions occur when the context changes, and the context is represented as a reactive value. If the execution of the program is interrupted, it is suspended and another execution that reflects the new context starts. The main difference from `ContextWorkflow` is that `ContextWorkflow` provides a wider variety of reactions to interruptions, using compensations, sub-workflows, and checkpoints, while Flute emphasizes changing program behavior according to context change.

Termination and Suspension. Rudys and Wallach [50] argue that in language run-time systems such as JVM that execute mobile code, it is important to be able to terminate such code for security reasons. For example, it can be critical to stop executing potentially buggy or untrusted mobile code. They propose a concept called *soft termination* to ensure that mobile code is properly terminated. For example, it makes a program with potentially infinite loops interruptible. Unlike our approach, theirs automatically transforms mobile code using code rewriting.

Several languages provide features to easily realize suspensions, such as first-class continuations [29, 15], which are supported in languages such as Scheme [55] and Scala [49], and *coroutines* [13]. Coroutines are a generalization of subroutines in the sense that they do not exit but call another coroutine as the caller coroutine suspends, and are supported in languages such as Lua [16]. We expect that these facilities are also useful for implementing `ContextWorkflow`.

Asynchronous Exception. Asynchronous exception, found in, e.g., Haskell [40], Ruby and OCaml [18], is also used to realize interruption. Java and Scala threads take a so-called semi-asynchronous approach [40], where asynchronous exceptions are thrown in the thread if the thread is blocked by `sleep()`, `wait()`, or `join()`; otherwise, an interrupted flag is turned on and the thread has to manually check the flag. The design of `ContextWorkflow` is closer to the former languages in the sense that such a flag to denote interruption is completely implicit.

Workflow. Workflow is a broadly used notion [22, 12] and is provided in several languages such as Windows Workflow Foundation [42] in .NET and Windows PowerShell [41]. PowerShell also supports checkpointing for fault tolerance. There are many studies for the

formalization of workflow [9, 10, 38]. Among them, our core ContextWorkflow is based on Bruni et al.'s formalization [9].

In a scientific workflow [39], which is an adaptation of the workflow to scientific computations, a series of heavy computations are executed. In a scientific workflow, checkpoints are also useful to avoid wasteful recomputation [14]. We suppose that ContextWorkflow can be used to develop these applications.

Software Transactional Memory. The software transactional memory (STM) [53], provided, e.g., by Scala [52] and Haskell [26], is a language-level approach to concurrency control, which is similar to a database transaction. STM provides the *atomic block* for atomic execution of all of the loads and stores of a critical section. If multiple atomic blocks are executed on multiple threads and inconsistency is found by interleaving execution, all the atomic blocks will be automatically rolled back. Checkpoints and continuations are also introduced in STM to realize partial aborts without using nested atomic block and gain efficiency [37]. STM is similar to our ContextWorkflow in the sense that they are automatically rolled back when some inconsistency occurs, although inconsistency is caused by rather different events (racy access to memory and context change).

Interruption in Functional Reactive Programming. The ideas of interruption and roll-back are also found in the context of FRP, such as P-FRP [34]. P-FRP is an FRP language for real-time systems, based on E-FRP [59]. In E-FRP, discrete events trigger executions of event handlers, which update reactive values. While E-FRP requires that each event handler execute atomically, P-FRP introduces priorities between events and allows event handlers to be interrupted when an urgent event occurs. To realize such an interruption, P-FRP adopts roll-back mechanisms like STM.

A difference from ours appears in what is rolled back and what kind of effect is removed. While P-FRP rolls back each event handlers and prevents reactive values from being updated incorrectly, ours rolls back the entire execution of a workflow and may remove any computational effects.

Compensation and Asynchronous Exception Monads. Ramalingam et al. showed that workflows with compensating actions can be represented by the compensation monad [47]. Besides the compensation monad, we also got the idea that computations with asynchronous exceptions can be represented by using the resumption monads [27, 28], which are structurally equal to the free monad [46].

Modular Exception Handling. Modularization of exception-handling code has been a significant concern in aspect-oriented programming [35, 11] because the separation of exception-handling code from normal code enhances the re-usability of each module. The compensation approach [60], which we adopt here, regards a pair of a normal code and a compensation as a unit of reuse instead, and also is modular.

Reversible Programming. Compensation actions can be seen as weak manual inversions of normal actions. In reversible programming languages [61], programs run forward and backward, and it is ensured that each direction is the exact inverse of the other. In other words, if programmers write a normal action in reversible programming languages, its compensation action is automatically defined. Therefore, integrating reversible programming to ContextWorkflow will be interesting because it can release programmers from the burden of manually

specifying compensation actions. Programming compensations is often cumbersome, but has an advantage that we may be able to avoid redundant compensation – such as visiting unnecessary nodes to go back to the start node as we saw in the maze search example in Section 2.

6 Conclusions

In this work, we have proposed ContextWorkflow for developing interruptible context-aware applications. ContextWorkflow basically combines the ideas of workflow and FRP and supports compensations, asynchronous interruption, checkpointing, nested-workflow and suspension. We also formalized the core idea of our language by developing a big-step operational semantics. Further, we proposed a method to embed our ContextWorkflow in existing languages such as Scala and Haskell, mainly using free monads; and the embedded DSL empowers host languages to treat the above features.

One important direction of future work is to support parallelism as many other workflow languages do, that is, atomic actions are executed in parallel on several threads. With parallelism, we expect the semantics of suspension, checkpoints, and sub-workflows to be changed drastically. A question is, for example, if only one sub-workflow of several concurrently running sub-workflows has a checkpoint, how does the whole workflow partially abort? In addition, in a parallel setting, an abort of a sub-workflow need not result in the abort of the parent workflow.

Another direction of future work is efficient implementation. Currently, since we use monad transformers naively, our implementation is not efficient; at least, we should unroll the monad transformer stack as is the standard practice in Haskell programming. It would also be valuable to develop ContextWorkflow with other implementation techniques such as first-class continuations and extensible effects [36], which are also introduced in Scala, and compare different implementations.

One tediousness in ContextWorkflow is that we have to write compensations manually, while we do not need to do so in database transaction and STM. Therefore, it would be interesting to develop a method to construct compensation actions from normal actions. Existing studies such as reversible computing would be helpful to achieve this.

In the current design, programmers can write as long atomic actions as they wish. Since we suppose that one application of ContextWorkflow is battery-aware software, it is interesting to automatically estimate how much execution time an atomic action will consume; then we can perform a kind of verification, e.g., by estimating that 10% of battery level would be enough to complete any compensations of the workflow. We expect that we can rely on existing studies about complexity estimation such as Gulwani et al. [25].

References

- 1 scalaz. URL: <https://github.com/scalaz/scalaz>.
- 2 Monadless. URL: <http://monadless.io/>.
- 3 Gregory Abowd, Anind Dey, Peter Brown, Nigel Davies, Mark Smith, and Pete Steggle. Towards a better understanding of context and context-awareness. In *Handheld and ubiquitous computing*, volume 1707 of *Springer LNCS*, pages 304–307, 1999.
- 4 Liliana Ardissono, Roberto Furnari, Anna Goy, Giovanna Petrone, and Marino Segnan. Context-aware workflow management. In *International Conference on Web Engineering*, volume 4607 of *Springer LNCS*, pages 47–52, 2007.
- 5 Steve Awodey. *Category Theory*. Oxford University Press, Inc., 2nd edition, 2010.

- 6 Engineer Bainomugisha, Andoni Lombide Carreton, Tom Van Cutsem, Stijn Mostinckx, and Wolfgang De Meuter. A survey on reactive programming. *ACM Computing Surveys (CSUR)*, 45(4):52, 2013.
- 7 Engineer Bainomugisha, Jorge Vallejos, Coen De Roover, Andoni Lombide Carreton, and Wolfgang De Meuter. Interruptible context-dependent executions: a fresh look at programming context-aware applications. In *Proc. of ACM Onward! 2012*, pages 67–84. ACM, 2012.
- 8 Rúnar Óli Bjarnarson. Stackless scala with free monads. *Scala Days*, 2012.
- 9 Roberto Bruni, Hernán Melgratti, and Ugo Montanari. Theoretical foundations for compensations in flow composition languages. In *Proc. of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages Pages (POPL 2005)*, pages 209–220. ACM, 2005.
- 10 Michael Butler, Tony Hoare, and Carla Ferreira. A trace semantics for long-running transactions. In *Communicating Sequential Processes. The First 25 Years*, volume 3525 of *Springer LNCS*, pages 133–150. Springer, 2005.
- 11 Nelio Cacho, Fernando Castor Filho, Alessandro Garcia, and Eduardo Figueiredo. EJFlow: Taming exceptional control flows in aspect-oriented programming. In *Proc. of AOSD'08*, pages 72–83, New York, NY, USA, 2008. ACM.
- 12 Christian Colombo and Gordon J. Pace. Recovery within long-running transactions. *ACM Comput. Surv.*, 45(3):28:1–28:35, 2013.
- 13 Melvin E Conway. Design of a separable transition-diagram compiler. *Communications of the ACM*, 6(7):396–408, 1963.
- 14 Daniel Crawl and Ilkay Altintas. A provenance-based fault tolerance mechanism for scientific workflows. In *Proc. of Provenance and Annotation of Data and Processes*, volume 5272 of *Springer LNCS*, pages 152–159, 2008.
- 15 Olivier Danvy and Andrzej Filinski. Abstracting control. In *Proc. of Lisp and Functional Programming*, pages 151–160, 1990.
- 16 Ana Lúcia de Moura, Noemi Rodriguez, and Roberto Ierusalimsky. Coroutines in Lua. *Journal of Universal Computer Science*, 10(7):910–925, 2004.
- 17 William R. Dieter and James E. Lump. A user-level checkpointing library for POSIX threads programs. In *Fault-Tolerant Computing, 1999. Digest of Papers. Twenty-Ninth Annual International Symposium on*, pages 224–227. IEEE, 1999.
- 18 Stephen Dolan, Spiros Eliopoulos, Daniel Hillerström, Anil Madhavapeddy, KC Sivaramakrishnan, and Leo White. Concurrent system programming with effect handlers. In *Proceedings of the Symposium on Trends in Functional Programming, TFP*, 2017.
- 19 Conal Elliott and Paul Hudak. Functional reactive animation. In *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming*, pages 263–273. ACM, 1997.
- 20 Marc Feeley. Polling efficiently on stock hardware. In *Proceedings of the conference on Functional programming languages and computer architecture*, pages 179–187. ACM, 1993.
- 21 Phil Freeman. Stack safety for free. URL: <http://functorial.com/stack-safety-for-free/index.pdf>.
- 22 Hector Garcia-Molina and Kenneth Salem. Sagas. In *Proc. of ACM SIGMOD*, pages 249–259, New York, NY, USA, 1987. ACM.
- 23 Jeremy Gibbons. Datatype-generic programming. In *Datatype-Generic Programming*, pages 1–71. Springer, 2007.
- 24 Jim Gray. The transaction concept: Virtues and limitations. In *Proceedings of the Seventh International Conference on Very Large Data Bases*, pages 144–154, 1981.

- 25 Sumit Gulwani, Krishna K. Mehra, and Trishul Chilimbi. Speed: Precise and efficient static estimation of program computational complexity. In *Proc. of ACM POPL*, pages 127–139, New York, NY, USA, 2009. ACM.
- 26 Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60. ACM, 2005.
- 27 William L. Harrison. The essence of multitasking. In *International Conference on Algebraic Methodology and Software Technology*, volume 4019 of *Springer LNCS*, pages 158–172. Springer, 2006.
- 28 William L. Harrison, Gerard Allwein, Andy Gill, and Adam Procter. Asynchronous exceptions as an effect. In *Proceedings of the 9th international conference on Mathematics of Program Construction*, volume 5133 of *Springer LNCS*, pages 153–176. Springer-Verlag, 2008.
- 29 Christopher T. Haynes, Daniel P. Friedman, and Mitchell Wand. Continuations and coroutines. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 293–298. ACM, 1984.
- 30 Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3):125–151, 2008.
- 31 Paul Hudak. Building domain-specific embedded languages. *ACM Computing Surveys (CSUR)*, 28(4es):196, 1996.
- 32 Hiroaki Inoue, Tomoyuki Aotani, and Atsushi Igarashi. A DSL for compensable and interruptible executions. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems*, REBLS 2017, pages 8–14, New York, NY, USA, 2017. ACM.
- 33 Hiroaki Inoue and Atsushi Igarashi. A library-based approach to context-dependent computation with reactive values: Suppressing reactions of context-dependent functions using dynamic binding. In *Companion Proc. of the 15th Intl. Conf. on Modularity*, pages 50–54, New York, NY, USA, 2016. ACM.
- 34 Roumen Kaiabachev, Walid Taha, and Angela Zhu. E-FRP with priorities. In *Proceedings of the 7th ACM & IEEE international conference on Embedded software*, pages 221–230. ACM, 2007.
- 35 Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proc. of ECOOP*, volume 1241 of *Springer LNCS*, pages 220–242. Springer, 1997.
- 36 Oleg Kiselyov, Amr Sabry, and Cameron Swords. Extensible effects: An alternative to monad transformers. In *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell, Haskell '13*, pages 59–70, New York, NY, USA, 2013. ACM.
- 37 Eric Koskinen and Maurice Herlihy. Checkpoints and continuations instead of nested transactions. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pages 160–168. ACM, 2008.
- 38 Jing Li, Huibiao Zhu, Geguang Pu, and Jifeng He. Looking into compensable transactions. In *Software Engineering Workshop, 2007. SEW 2007. 31st IEEE*, pages 154–166. IEEE, 2007.
- 39 Bertram Ludäscher, Ilkay Altintas, Chad Berkley, Dan Higgins, Efrat Jaeger, Matthew Jones, Edward A Lee, Jing Tao, and Yang Zhao. Scientific workflow management and the Kepler system. *Concurrency and Computation: Practice and Experience*, 18(10):1039–1065, 2006.

- 40 Simon Marlow, Simon Peyton Jones, Andrew Moran, and John Reppy. Asynchronous exceptions in Haskell. In *Proc. of ACM PLDI*, pages 274–285, New York, NY, USA, 2001. ACM.
- 41 Microsoft. Powershell documentation. URL: <https://docs.microsoft.com/powershell/>.
- 42 Microsoft. Windows workflow foundation. URL: <https://docs.microsoft.com/en-us/dotnet/framework/windows-workflow-foundation/>.
- 43 Eugenio Moggi. Computational lambda-calculus and monads. In *Logic in Computer Science, 1989. LICS'89, Proceedings., Fourth Annual Symposium on*, pages 14–23. IEEE, 1989.
- 44 N.C. Narendra and S Gundugola. Automated context-aware adaptation of web service executions. In *Proceedings of the IEEE International Conference on Computer Systems and Applications*, pages 179–187. IEEE Computer Society, 2006.
- 45 Bruno C. d. S. Oliveira and Jeremy Gibbons. Scala for generic programmers: comparing Haskell and Scala support for generic programming. *Journal of functional programming*, 20(3-4):303–352, 2010.
- 46 Maciej Piróg and Jeremy Gibbons. The coinductive resumption monad. In *Mathematical Foundations of Programming Semantics Thirtieth Annual Conference*, page 273, 2014.
- 47 Ganesan Ramalingam and Kapil Vaswani. Fault tolerance via idempotence. In *Proc. of ACM POPL*, POPL '13, pages 249–262, New York, NY, USA, 2013. ACM.
- 48 Brian Randell, Peter Lee, and Philip C. Treleaven. Reliability issues in computing system design. *ACM Computing Surveys (CSUR)*, 10(2):123–165, 1978.
- 49 Tiark Rompf, Ingo Maier, and Martin Odersky. Implementing first-class polymorphic delimited continuations by a type-directed selective CPS-transform. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, ICFP '09, pages 317–328. ACM, 2009.
- 50 Algis Rudys and Dan S. Wallach. Termination in language-based systems. *ACM Transactions on Information and System Security (TISSEC)*, 5(2):138–168, 2002.
- 51 Guido Salvaneschi, Gerold Hintz, and Mira Mezini. REScala: Bridging between object-oriented and functional style in reactive applications. In *Proc. of Intl. Conf. on Modularity*, pages 25–36. ACM, 2014.
- 52 STM Scala. Expert group. scalastm. web, 2011. URL: <https://nbronson.github.io/scala-stm/>.
- 53 Nir Shavit and Dan Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997.
- 54 Sucha Smanchat, Sea Ling, and Maria Indrawan. A survey on context-aware workflow adaptations. In *Proceedings of the 6th International Conference on Advances in Mobile Computing and Multimedia*, pages 414–417. ACM, 2008.
- 55 Michael Sperber, R. Kent Dybvig, Matthew Flatt, Anton Van Straaten, Robby Findler, and Jacob Matthews. Revised⁶ report on the algorithmic language Scheme. *Journal of Functional Programming*, 19(S1):1–301, 2009.
- 56 Janis Voigtländer. Asymptotic improvement of computations over free monads. In *Proceedings of the 9th International Conference on Mathematics of Program Construction*, pages 388–403. Springer-Verlag, 2008.
- 57 Martin von Löwis, Marcus Denker, and Oscar Nierstrasz. Context-oriented programming: Beyond layers. In *Proc. of Intl. Conf. on Dynamic Languages*, pages 143–156, New York, NY, USA, 2007. ACM.
- 58 Philip Wadler. Monads for functional programming. In *International School on Advanced Functional Programming*, pages 24–52. Springer, 1995.
- 59 Zhanyong Wan, Walid Taha, and Paul Hudak. Event-driven FRP. In *International Symposium on Practical Aspects of Declarative Languages*, pages 155–172. Springer, 2002.

- 60 Westley Weimer. Exception-handling bugs in Java and a language extension to avoid them. In *Advanced Topics in Exception Handling Techniques*, volume 4119 of *Springer LNCS*, pages 22–41, 2006.
- 61 Tetsuo Yokoyama and Robert Glück. A reversible programming language and its invertible self-interpreter. In *Proc. ACM PEPM*, pages 144–153, New York, NY, USA, 2007. ACM.
- 62 Lukasz Ziarek and Suresh Jagannathan. Lightweight checkpointing for concurrent ML. *Journal of Functional Programming*, 20(2):137–173, 2010.

A Appendix

A.1 Hiding Type Parameters for Simplicity

The type `CW[A]` in Section 2 is in fact an abbreviation of `CW[Unit,IO,Nothing,A]`. The important point is to fix `M` to `IO` and `S` to `Nothing`. In Scala, `Nothing` is a subtype of every other type.

The monad `IO` is the standard way to treat effectful code in monadic programming, but explicit use of the `IO` monad constructor is redundant and not kind to many programmers. Therefore, we hide the explicit appearance of `IO` using implicit conversions of Scala. For example, the way `a /+ c` is converted to the corresponding monadic value is that (1) `a` of the type `A` is converted to a special object of the type `CWOps` that contains a field of the type `IO[A]` by implicit conversions, and then (2) the method `/+` of the special object is invoked. It takes an argument of the type `A => Unit` and returns a value of the type `CW[A]`. Here is the definition of the implicit conversion and the class `CWOps`:

```
implicit def toCWOps[A](proc: => A): CWOps[A] = new CWOps[A](IO(proc))
class CWOps[A](t: IO[A]) {
  def /+ (comp: => A => Unit): CW[A] = /+(t)(a => IO(comp(a)))
}
```

`toCWOps` is the definition for the implicit conversion. `IO(a)` is the `IO` monad constructor. We define the method `/+` in class `CWOps` using the function `/+` that appeared in Section 4.

The reason for using `Nothing` as the suspended workflow type is that, to treat `CW` as a monad, type parameters except for `A` must be fixed or parameterized. Although the latter approach appears good, it would become redundant in Scala. For example, let `CWS[S,A]` be `CW[Unit,IO,S,A]`, and let us combine two `CWS`:

```
def testU[S]: CWS[S,Unit] = ...; def testI[S]: CWS[S,Int] = ...
def testUI[S]: CWS[S,Int] = testU[S].flatMap(_ => testI[S])
```

We would have to use `def` and then type parameter `S` would appear everywhere, since Scala's value is not polymorphic. While such definitions can be treated well in Haskell, we would have to manually parameterize it one by one in Scala. Instead, we fix `S` to `Nothing` and cast `Nothing` to a proper suspended workflow type `Fix[CW[Unit,IO,?,A]]` at run time.

A.2 Derivation Example

Let $t_k = A_k/C_k$ for $k = a, b, 1, 2, \dots$

$$\begin{array}{c}
 \boxed{\text{subgoal 1: } \langle \text{sub} \{ \text{sub} \{ t_1; \text{cp}; \text{sub}(t_2)/C_a; \text{check} \} / C_b; t_3 \}; t_4, \square, \bullet \rangle \uparrow_P^{A_1, A_2} \langle C_a; \text{ccp}_0; C_1; \text{sub}; \text{sub}, \square \rangle} \\
 \hline
 \frac{\frac{\frac{\frac{\frac{\frac{\langle t_1; \text{cp}; \text{sub}(t_2)/C_a; \text{check}, E_0, \text{sub}; \text{sub} \rangle \uparrow_P^{A_1, A_2} \langle \bar{c}_0, \square \rangle}{\langle \text{sub}(t_1; \text{cp}; \text{sub}(t_2)/C_a; \text{check} \} / C_b, \text{sub}(\square); t_3, \text{sub} \rangle \uparrow_P^{A_1, A_2} \langle \bar{c}_0, \square \rangle}{\langle \text{sub}(t_1; \text{cp}; \text{sub}(t_2)/C_a; \text{check} \} / C_b; t_3, \text{sub}(\square); t_4, \text{sub} \rangle \uparrow_P^{A_1, A_2} \langle \bar{c}_0, \square \rangle}{\langle \text{sub}(t_1; \text{cp}; \text{sub}(t_2)/C_a; \text{check} \} / C_b; t_3, \square; t_4, \bullet \rangle \uparrow_P^{A_1, A_2} \langle \bar{c}_0, \square \rangle}{\langle \text{sub}(\text{sub}(t_1; \text{cp}; \text{sub}(t_2)/C_a; \text{check} \} / C_b; t_3), \square; t_4, \bullet \rangle \uparrow_P^{A_1, A_2} \langle \bar{c}_0, \square \rangle}{\langle \text{sub}(\text{sub}(t_1; \text{cp}; \text{sub}(t_2)/C_a; \text{check} \} / C_b; t_3); t_4, \square, \bullet \rangle \uparrow_P^{A_1, A_2} \langle \bar{c}_0, \square \rangle} \text{Seq-Int1}}}{\langle t_1; \text{cp}; \text{sub}(t_2)/C_a; \text{check}, E_0, \text{sub}; \text{sub} \rangle \uparrow_P^{A_1, A_2} \langle \bar{c}_0, \square \rangle} \text{Sub-Int}}}{\langle \text{sub}(t_1; \text{cp}; \text{sub}(t_2)/C_a; \text{check} \} / C_b, \text{sub}(\square); t_3, \text{sub} \rangle \uparrow_P^{A_1, A_2} \langle \bar{c}_0, \square \rangle} \text{Seq-Int1}}}{\langle \text{sub}(t_1; \text{cp}; \text{sub}(t_2)/C_a; \text{check} \} / C_b; t_3, \text{sub}(\square); t_4, \text{sub} \rangle \uparrow_P^{A_1, A_2} \langle \bar{c}_0, \square \rangle} \text{Seq-Int1}}}{\langle \text{sub}(t_1; \text{cp}; \text{sub}(t_2)/C_a; \text{check} \} / C_b; t_3, \square; t_4, \bullet \rangle \uparrow_P^{A_1, A_2} \langle \bar{c}_0, \square \rangle} \text{Sub-Int}}}{\langle \text{sub}(t_1; \text{cp}; \text{sub}(t_2)/C_a; \text{check} \} / C_b; t_3, \square; t_4, \bullet \rangle \uparrow_P^{A_1, A_2} \langle \bar{c}_0, \square \rangle} \text{Sub-Int}}}{\langle \text{sub}(\text{sub}(t_1; \text{cp}; \text{sub}(t_2)/C_a; \text{check} \} / C_b; t_3), \square; t_4, \bullet \rangle \uparrow_P^{A_1, A_2} \langle \bar{c}_0, \square \rangle} \text{Seq-Int1}}}{\langle \text{sub}(\text{sub}(t_1; \text{cp}; \text{sub}(t_2)/C_a; \text{check} \} / C_b; t_3); t_4, \square, \bullet \rangle \uparrow_P^{A_1, A_2} \langle \bar{c}_0, \square \rangle} \text{Seq-Int1}}}{\langle A_1/C_1, E_0[\square]; \text{cp}; \text{sub}(t_2)/C_a; \text{check} \rangle, \text{sub}; \text{sub} \rangle \downarrow_P^{A_1} \langle C_1; \text{sub}; \text{sub} \rangle} \text{PW}}{\langle \text{cp}, E_1, C_1; \text{sub}; \text{sub} \rangle \downarrow_P^\epsilon \langle \text{ccp}_0; C_1; \text{sub}; \text{sub} \rangle} \text{CP}}{\langle A_1/C_1, E_0[\square]; \text{cp}; \text{sub}(t_2)/C_a; \text{check} \rangle, \text{sub}; \text{sub} \rangle \downarrow_P^{A_1} \langle C_1; \text{sub}; \text{sub} \rangle} \text{Seq-Int2}}{\langle \text{cp}; \text{sub}(t_2)/C_a; \text{check}, E_0, C_1; \text{sub}; \text{sub} \rangle \uparrow_P^{A_2} \langle \bar{c}_0, \square \rangle} \text{Seq-Int2}}{\langle t_1; \text{cp}; \text{sub}(t_2)/C_a; \text{check}, E_0, \text{sub}; \text{sub} \rangle \uparrow_P^{A_1, A_2} \langle \bar{c}_0, \square \rangle} \text{Sub-Int}}{\langle \text{sub}(t_1; \text{cp}; \text{sub}(t_2)/C_a; \text{check} \} / C_b, \text{sub}(\square); t_3, \text{sub} \rangle \uparrow_P^{A_1, A_2} \langle \bar{c}_0, \square \rangle} \text{Seq-Int1}}{\langle \text{sub}(t_1; \text{cp}; \text{sub}(t_2)/C_a; \text{check} \} / C_b; t_3, \text{sub}(\square); t_4, \text{sub} \rangle \uparrow_P^{A_1, A_2} \langle \bar{c}_0, \square \rangle} \text{Seq-Int1}}{\langle \text{sub}(t_1; \text{cp}; \text{sub}(t_2)/C_a; \text{check} \} / C_b; t_3, \square; t_4, \bullet \rangle \uparrow_P^{A_1, A_2} \langle \bar{c}_0, \square \rangle} \text{Sub-Int}}{\langle \text{sub}(\text{sub}(t_1; \text{cp}; \text{sub}(t_2)/C_a; \text{check} \} / C_b; t_3), \square; t_4, \bullet \rangle \uparrow_P^{A_1, A_2} \langle \bar{c}_0, \square \rangle} \text{Seq-Int1}}{\langle \text{sub}(\text{sub}(t_1; \text{cp}; \text{sub}(t_2)/C_a; \text{check} \} / C_b; t_3); t_4, \square, \bullet \rangle \uparrow_P^{A_1, A_2} \langle \bar{c}_0, \square \rangle} \text{Seq-Int1}}{\langle \text{sub}(t_2)/C_a; \text{check}, E_0, \text{ccp}_0; C_1; \text{sub}; \text{sub} \rangle \uparrow_P^{A_2} \langle C_a; \text{ccp}_0; C_1; \text{sub}; \text{sub}, \square \rangle} \text{Subgoal 2}} \\
 \hline
 \frac{\frac{\frac{\langle A_2/C_2, E_2, \text{sub}; \text{ccp}_0; C_1; \text{sub}; \text{sub} \rangle \downarrow_P^{A_2} \langle C_2; \text{sub}; \text{ccp}_0; C_1; \text{sub}; \text{sub} \rangle}{\langle \text{sub}(t_2)/C_a, E_0[\square]; \text{check} \rangle, \text{ccp}_0; C_1; \text{sub}; \text{sub} \rangle \downarrow_P^{A_2} \langle \bar{c}_0 \rangle} \text{PW}}{\langle \text{sub}(t_2)/C_a; \text{check}, E_0, \text{ccp}_0; C_1; \text{sub}; \text{sub} \rangle \uparrow_P^{A_2} \langle C_a; \text{ccp}_0; C_1; \text{sub}; \text{sub}, \square \rangle} \text{Sub}}{\langle \text{check}, E_0, C_a; \text{ccp}_0; C_1; \text{sub}; \text{sub} \rangle \uparrow_P^\epsilon \langle \bar{c}_0, \square \rangle} \text{Check-PAAbort}}{\langle \text{sub}(t_2)/C_a; \text{check}, E_0, \text{ccp}_0; C_1; \text{sub}; \text{sub} \rangle \uparrow_P^{A_2} \langle C_a; \text{ccp}_0; C_1; \text{sub}; \text{sub}, \square \rangle} \text{Seq-Int2}}{\langle \text{sub}\{ \text{sub}\{ t_1; \text{cp}; \text{sub}(t_2)/C_a; \text{check} \} / C_b; t_3 \}; t_4 \rangle \downarrow_P^{A_1, A_2, C_a} \langle C_1; \text{sub}; \text{sub}, \text{sub}(\text{sub}(\square; \text{cp}\#E_1; \text{sub}(t_2)/C_a; \text{check})/C_b; t_3); t_4 \rangle} \text{Goal}} \\
 \hline
 \frac{\frac{\frac{\frac{\langle C_a \rangle \downarrow_P^{C_a} \langle \bullet, \square \rangle}{\langle \text{cp}\#E_1 \rangle[\square]; \text{cp}\#E_1]} \downarrow_P^\epsilon \langle C_1; \text{sub}; \text{sub}, E_1[\square]; \text{cp}\#E_1 \rangle} \text{Comp-Ccp}}{\langle \text{ccp}_0; C_1; \text{sub}; \text{sub} \rangle \downarrow_P^\epsilon \langle C_1; \text{sub}; \text{sub}, E_1[\square]; \text{cp}\#E_1 \rangle} \text{Comp-Seq-PAAbort1}}{\langle \text{sub}\{ \text{sub}\{ t_1; \text{cp}; \text{sub}(t_2)/C_a; \text{check} \} / C_b; t_3 \}; t_4 \rangle \downarrow_P^{A_1, A_2, C_a} \langle C_1; \text{sub}; \text{sub}, E \rangle} \text{Comp-Seq-PAAbort2}}{\langle \bar{c}_0 \rangle \downarrow_P^{C_a} \langle C_1; \text{sub}; \text{sub}, E_1[\square]; \text{cp}\#E_1 \rangle} \text{Program-PAAbort}}{\langle \text{sub}\{ \text{sub}\{ t_1; \text{cp}; \text{sub}(t_2)/C_a; \text{check} \} / C_b; t_3 \}; t_4 \rangle \downarrow_P^{A_1, A_2, C_a} \langle C_1; \text{sub}; \text{sub}, E \rangle} \text{Subgoal1}}{\langle \text{sub}\{ \text{sub}\{ t_1; \text{cp}; \text{sub}(t_2)/C_a; \text{check} \} / C_b; t_3 \}; t_4 \rangle \downarrow_P^{A_1, A_2, C_a} \langle C_1; \text{sub}; \text{sub}, E \rangle} \text{Program-PAAbort}}
 \end{array}$$

where

$$\begin{array}{ll}
 E_0 & = \text{sub}(\text{sub}(\square)/C_b; t_3); t_4 \\
 E_1 & = E_0[\square]; \text{sub}(t_2)/C_a; \text{check} = \text{sub}(\text{sub}(\square); \text{sub}(t_2)/C_a; \text{check})/C_b; t_3); t_4 \\
 E_2 & = E_0[\text{sub}(\square)/C_a; \text{check}] = \text{sub}(\text{sub}(\text{sub}(\square)/C_a; \text{check})/C_b; t_3); t_4 \\
 \text{ccp}_0 & = \text{ccp}\#E_1[\square]; \text{cp}\#E_1 \\
 \bar{c}_0 & = C_a; \text{ccp}_0; C_1; \text{sub}; \text{sub} \\
 E & = E_1[\square]; \text{cp}\#E_1 = \text{sub}(\text{sub}(\square); \text{cp}\#E_1; \text{sub}(t_2)/C_a; \text{check})/C_b; t_3); t_4
 \end{array}$$

■ **Figure 6** A derivation of an execution of $\text{sub}\{\text{sub}\{t_1; \text{cp}; \text{sub}(t_2)/C_a; \text{check}\}/C_b; t_3\}; t_4$ with **abort** at **check**.

A.3 Proofs of Properties

In the following theorems, let $p_k = A_k/C_k$ for some k , and we define the functions as follows.

- $b(t)$ be a workflow that is obtained by removing **sub**, **check**, **cp** and **cp#E** from t .
- $\text{includes}(t, m, n)$ iff $b(t) = p_m; \dots; p_n$ and $m \leq n$; or t has no primitive workflows.
- $\text{includes}(E, m, n)$ iff $\text{includes}(E[\text{check}], m, n)$.
- $\text{includes}(\bar{c}, m, n)$ iff $\bar{c} \setminus \{\text{sub}, \text{ccp}\#E\} = C_m; \dots; C_n$ and $m \geq n$; or \bar{c} has no atomic actions C_* .
- $\text{nosub}(t, m, n)$ iff $\text{includes}(t, m, n)$ and t has no **sub**-workflows.

► **Lemma 1 (Commit)**. *If $\text{includes}(t, m, n)$ and $\langle t, E, \bar{c} \rangle \downarrow_{\vec{A}} \langle \bar{c}' \rangle$, then $\vec{A} = A_m, \dots, A_n$ (if $m \leq n$) or $\vec{A} = \epsilon$ (otherwise).*

Proof. By straightforward induction on the derivation. ◀

► **Lemma 2 (Abort)**. *If $\text{nosub}(t, m, n)$ and $\langle t, E, \bar{c} \rangle \uparrow_{A|P}^{\vec{A}} \langle \bar{c}', \square \rangle$, then $\vec{A} = A_m, \dots, A_i$ and $\text{includes}(\bar{c}', i, m)$ for some i such that $m \leq i \leq n$ (when $m \leq n$), or $\vec{A} = \epsilon \wedge \text{includes}(\bar{c}', 0, 1)$ (otherwise).*

Proof. By straightforward induction on the derivation. \blacktriangleleft

► **Lemma 3** (Compensation). *If $\bar{c} = C_m, \dots, C_n$ and $\langle \bar{c} \rangle \Downarrow^{\bar{c}} \langle \bullet, [] \rangle$, then $\vec{C} = C_m, \dots, C_n$.*

Proof. By straightforward induction on the derivation. \blacktriangleleft

► **Lemma 4** (Checkpoint). *Suppose $\text{nosub}(t, m, k)$ and t has no $\text{cp}\#E_*$ and $\langle t, E, \bar{c} \rangle \Downarrow^{\vec{A}} \langle \bar{c}', [] \rangle$ and $\text{includes}(E, k+1, n)$ and $\text{includes}(\bar{c}, m-1, l)$ and $l \leq m$ and $\text{ccp}\#E_s \notin \bar{c}$ and $\text{ccp}\#E_s \in \bar{c}'$ and $\text{ccp}\#E_s$ comes just after C_j (or just before C_{j+1} , so \bar{c}' usually becomes $C_k, \dots, C_{j+1}, \dots, \text{ccp}\#E_s, \dots, C_j, \dots, C_m$) and $m-1 \leq j \leq k$.*

1. *If $m-1 \leq k \leq n \wedge m \leq n$, then $\text{includes}(E_s, j+1, n)$.*

2. *If $n \leq k < m$, then $\text{includes}(E_s, m, k)$.*

Proof. Proof by induction on the derivation of $\langle t, E, \bar{c} \rangle \Downarrow^{\vec{A}} \langle \bar{c}', [] \rangle$. We show only main cases for the first item.

Case CW-CHECKPOINT: $E_s = E[[]; \text{cp}\#E]$ $j = m-1$

It is the case that $k = m-1$, and so $\text{includes}(E, m, n)$. Clearly, $\text{includes}(E_s, m, n)$, finishing the case.

Case CW-SEQ: $t = t_1; t_2$ $\langle t_1, E[[]; t_2], \bar{c} \rangle \Downarrow^{\vec{A}_1} \langle \bar{c}'' \rangle$
 $\langle t_2, E, \bar{c}'' \rangle \Downarrow^{\vec{A}_2} \langle \bar{c}' \rangle$

We get $\text{includes}(t_1, m, i)$ and $\text{includes}(t_2, i+1, k)$ for some i s.t. $m-1 \leq i \leq k$. The induction hypothesis finishes the case. \blacktriangleleft

► **Lemma 5** (Partial Abort). *Suppose $\text{nosub}(t, m, n_0)$ and t has no $\text{cp}\#E_*$ and $\langle t, [], \bullet \rangle \Uparrow_P^{\vec{A}} \langle \bar{c}', [] \rangle$ and $\vec{A} = A_m, \dots, A_n$ and $\text{includes}(\bar{c}', n, m)$ and $\langle \bar{c}' \rangle \Downarrow^{\bar{c}} \langle \bar{c}'', E_s \rangle$.*

■ *If $m \leq n$, then $\vec{C} = \epsilon$ and $\text{includes}(E_s, m, n)$ and $\text{includes}(\bar{c}'', n, m)$, or $\vec{C} = C_n, \dots, C_{k+1}$ and $\text{includes}(E_s, k+1, n)$ and $\text{includes}(\bar{c}'', k, m)$ for some k s.t. $m-1 \leq k < n$.*

■ *If $m > n$, then $\vec{C} = \epsilon$ and $\text{includes}(E_s, m, n)$ and $\text{includes}(\bar{c}'', n, m)$.*

Proof. Proof by induction on the derivation of $\langle \bar{c} \rangle \Downarrow^{\bar{c}} \langle \bar{c}', E_s \rangle$, using Lemma 4. \blacktriangleleft

► **Lemma 6** (Suspend). *Suppose $\text{includes}(t, m, k)$ and $\langle t, E, \bar{c} \rangle \Uparrow_S^{\vec{A}} \langle \bar{c}', E_s \rangle$ and $\text{includes}(E, k+1, n)$.*

1. *If $m-1 \leq k \leq n \wedge m \leq n$, then $\vec{A} = A_m, \dots, A_i$ for some i such that $m \leq i \leq k$ and $\text{includes}(E_s, i+1, n)$, or $\vec{A} = \epsilon$ and $\text{includes}(E_s, m, n)$.*

2. *If $n \leq k < m$, then $\text{includes}(E_s, m, k)$.*

Proof. Proof by induction on the derivation. We show only main cases for the first item.

Case CW-CHECK-SUSPEND:

It is the case that $k = m-1$, and so $\text{includes}(E, m, n)$, finishing the case.

Case CW-SUB-INT: $t = \text{sub}(t')/c$

We can get $\text{includes}(t', m, k)$ and $\text{includes}(E[(\text{sub } [])/c], k+1, n)$. Then, the induction hypothesis finishes the case.

Case CW-SEQ-INT1: $t = t_1; t_2$

We get $\text{includes}(t_1, m, j)$ for some j s.t., $m-1 \leq j \leq k$. We also get $\text{includes}(E[[]; t_2], j+1, n)$. Then, the induction hypothesis finishes the case.

Case CW-SEQ-INT2: $\mathfrak{t} = \mathfrak{t}_1; \mathfrak{t}_2 \quad \langle \mathfrak{t}_1, \mathbf{E}[\square; \mathfrak{t}_2], \bar{\mathfrak{c}} \rangle \Downarrow_{\vec{A}_1} \langle \bar{\mathfrak{c}}' \rangle$
 $\langle \mathfrak{t}_2, \mathbf{E}, \bar{\mathfrak{c}}' \rangle \Uparrow_S^{\vec{A}_2} \langle \bar{\mathfrak{c}}', \mathbf{E}_s \rangle$

We get $\text{includes}(\mathfrak{t}_1, m, j)$ for some j s.t., $m - 1 \leq j \leq k$. By Lemma 1, $\vec{A}_1 = A_m, \dots, A_{j-1}$ (when $m \leq j$), or $\vec{A}_1 = \epsilon$ (when $j = m - 1$). We also get $\text{includes}(\mathfrak{t}_2, j + 1, k)$ from $\text{includes}(\mathfrak{t}, m, k)$ and $\text{includes}(\mathfrak{t}_1, m, j)$. We still have $\text{includes}(\mathbf{E}, k, n)$.

Then, by the induction hypothesis, $\vec{A}_2 = A_j, \dots, A_i$ for some i such that $j \leq i \leq k$ and $\text{includes}(\mathbf{E}_s, i + 1, n)$, or $\vec{A}_2 = \epsilon$ and $\text{includes}(\mathbf{E}_s, m, n)$.

Finally, we can finish the case concatenating \vec{A}_1 and \vec{A}_2 . \blacktriangleleft

► **Theorem 1** (Workflow commits). *If $\text{includes}(\mathfrak{t}, m, n)$ and $\langle \mathfrak{t}, \bar{\mathfrak{c}} \rangle \Downarrow_{\vec{A}} \langle \rangle$ and $m \leq n$, then $\vec{A} = A_m, \dots, A_n$.*

Proof. By Lemma 1 and CW-PROGRAM-COMMIT. \blacktriangleleft

► **Theorem 2** (Workflow aborts (Successful Compensation)). *If $\text{nosub}(\mathfrak{t}, m, n)$ and $\langle \mathfrak{t}, \bar{\mathfrak{c}} \rangle \Downarrow_A^{\vec{A}} \langle \rangle$ and $m \leq n$ and $\bar{\mathfrak{c}} = C_k, \dots, C_l$, then $\vec{A} = A_m, \dots, A_i, C_i, \dots, C_m, C_k, \dots, C_l$ for some i s.t. $m \leq i \leq n$.*

Proof. By Lemmas 2 and 3 and CW-PROGRAM-ABORT. \blacktriangleleft

► **Theorem 3** (Restarted suspended workflow commits). *If $\langle \mathfrak{t}, \bullet \rangle \Downarrow_{\vec{A}} \langle \rangle$ and $\langle \mathfrak{t}, \bullet \rangle \Downarrow_S^{\vec{C}} \langle \bar{\mathfrak{c}}, \mathbf{E} \rangle$ and $\langle \mathbf{E}[\text{check}], \bar{\mathfrak{c}} \rangle \Downarrow_{\vec{C}'} \langle \rangle$, then $\vec{A} = \vec{C}, \vec{C}'$.*

Proof. By Theorem 1, Lemma 6 and CW-PROGRAM-SUSPEND. \blacktriangleleft

► **Theorem 4** (Workflow partially aborts). *If $\text{nosub}(\mathfrak{t}, m, n)$ and $\langle \mathfrak{t}, \bullet \rangle \Downarrow_P^{\vec{A}} \langle \bar{\mathfrak{c}}, \mathbf{E} \rangle$ and $m \leq n$, then either of the followings hold.*

■ $\vec{A} = A_m, \dots, A_i, C_i, C_{i-1}, \dots, C_j$ and $\text{includes}(\mathbf{E}, j, n)$ and $\text{includes}(\bar{\mathfrak{c}}, j - 1, m)$ for some i and j ($m \leq j \leq i \leq n$).

■ $\vec{A} = A_m, \dots, A_n$ and $\text{includes}(\mathbf{E}, 1, 0)$ and $\text{includes}(\bar{\mathfrak{c}}, n, m)$.

Moreover, the followings hold.

1. (Suspended workflow commits) *If $\langle \mathbf{E}[\text{check}], \bar{\mathfrak{c}} \rangle \Downarrow_{\vec{A}'} \langle \rangle$, then $\vec{A}' = A_j, \dots, A_n$, or $\vec{A}' = \epsilon$ (if $\text{includes}(\mathbf{E}, 1, 0)$).*
2. (Suspended workflow aborts) *If $\langle \mathbf{E}[\text{check}], \bar{\mathfrak{c}} \rangle \Downarrow_A^{\vec{A}'} \langle \rangle$, then $\vec{A}' = \epsilon$ (if $j = m$), or $\vec{A}' = C_{j-1}, \dots, C_m$.*

Proof. By Lemma 2, Lemma 5 and CW-PROGRAM-PABORT.

1. By Theorem 1.
2. By Theorem 2. \blacktriangleleft

► **Theorem 5** (Partial abort, checkpoint and nested workflow). *Suppose that $\text{includes}(\mathfrak{t}, 1, n)$ and $\mathfrak{t} \setminus \text{check} =$*

$p_1; \dots; c_p; p_k; \dots; p_m; \text{sub}(p_{m+1}; \dots; c_p; p_j; \dots; p_l) / C_a; p_{l+1}; \dots; p_n$ and $\langle \mathfrak{t}, \bullet \rangle \Downarrow_P^{\vec{A}} \langle \bar{\mathfrak{c}}, \mathbf{E} \rangle$.

1. (Partial abort skips compensations of complete sub-workflow) *If $A_{l+1} \in \{\vec{A}\}$, then $\vec{A} = A_1, \dots, A_i, C_i, \dots, C_{l+1}, C_a, C_m, \dots, C_k$ for some $i > l$.*
2. (A suspended workflow remembers checkpoints in a sub-workflow) *If $A_{l+1} \in \{\vec{A}\}$ and $\langle \mathbf{E}[\text{check}], \bar{\mathfrak{c}} \rangle \Downarrow_P^{\vec{A}'} \langle \bar{\mathfrak{c}}', \mathbf{E}' \rangle$ and $A_j \in \{\vec{A}'\} \wedge A_l \notin \{\vec{A}'\}$, then $\vec{A}' = A_k, \dots, A_i, C_i, \dots, C_j$ for some i s.t. $j \leq i \leq l$.*
3. (A suspended workflow remembers checkpoints before a sub-workflow) *If $C_j \in \{\vec{A}\}$ and $\langle \mathbf{E}[\text{check}], \bar{\mathfrak{c}} \rangle \Downarrow_P^{\vec{A}'} \langle \bar{\mathfrak{c}}', \mathbf{E}' \rangle$ and $A_{l+1} \in \{\vec{A}'\}$, then $\vec{A}' = A_j, \dots, A_i, C_i, \dots, C_{l+1}, C_a, C_m, \dots, C_k$ for some $i > l$.*

Proof. Let $E_0 = [] ; \text{cp}\#E_0 ; p_k ; \dots ; \text{sub}(\dots ; \text{cp} ; \dots) / C_a ; \dots ; p_n$ and $E_1 = \text{cp}\#E_0 ; \text{sub}([] ; \text{cp}\#E_1 ; \dots) / C_a ; \dots ; p_n$.


1. Straightforwardly from the derivation, using Lemma 1 and Lemma 2. Notice that the CW-SUB deletes the `cp` inside the `sub` and installs the other compensation C_a .
2. We can get $E = E_0$ from the derivation tree. Then, the conclusion follows straightforwardly from the derivation of $\langle E[\text{check}], \bar{c} \rangle \Downarrow_P^{\vec{A}} \langle \bar{c}'', E' \rangle$ using Lemma 1 and Lemma 2.
3. We can get $E = E_1$ from the derivation tree. Then, the conclusion follows straightforwardly from the derivation of $\langle E[\text{check}], \bar{c} \rangle \Downarrow_P^{\vec{A}} \langle \bar{c}'', E' \rangle$ using Lemma 1 and Lemma 2. ◀

Theory and Practice of Coroutines with Snapshots

Aleksandar Prokopec

Oracle Labs, Zürich, Switzerland


aleksandar.prokopec@gmail.com

 <https://orcid.org/0000-0003-0260-2729>

Fengyun Liu

École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland

fengyun.liu@epfl.ch

 <https://orcid.org/0000-0001-7949-4303>

Abstract

While event-driven programming is a widespread model for asynchronous computing, its inherent control flow fragmentation makes event-driven programs notoriously difficult to understand and maintain. *Coroutines* are a general control flow construct that can eliminate control flow fragmentation. However, coroutines are still missing in many popular languages. This gap is partly caused by the difficulties of supporting suspendable computations in the language runtime.

We introduce first-class, type-safe, stackful coroutines with snapshots, which unify many variants of suspendable computing. Our design relies solely on the static metaprogramming support of the host language, without modifying the language implementation or the runtime. We also develop a formal model for type-safe, stackful and delimited coroutines, and we prove the respective safety properties. We show that the model is sufficiently general to express iterators, single-assignment variables, `async-await`, actors, event streams, backtracking, symmetric coroutines and continuations. Performance evaluations reveal that the proposed metaprogramming-based approach has a decent performance, with workload-dependent overheads of $1.03 - 2.11\times$ compared to equivalent manually written code, and improvements of up to $6\times$ compared to other approaches.

2012 ACM Subject Classification Software and its engineering → Coroutines, Software and its engineering → Control structures

Keywords and phrases coroutines, continuations, coroutine snapshots, asynchronous programming, inversion of control, event-driven programming

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2018.3

Related Version <https://arxiv.org/abs/1806.01405>

1 Introduction

Asynchronous programming is becoming increasingly important, with applications ranging from actor systems [1, 21], futures and network programming [10, 22], user interfaces [30], to functional stream processing [32]. Traditionally, these programming models were realized either by blocking execution threads (which can be detrimental to performance [4]), or callback-style APIs [10, 22, 26], or with monads [67]. However, these approaches often feel unnatural, and the resulting programs can be hard to understand and maintain. *Coroutines* [11] overcome the need for blocking threads, callbacks and monads by allowing parts of the execution to pause at arbitrary points, and resuming that execution later.

There are generally two approaches to implement control flow constructs like coroutines: *call stack manipulation* and *program transformation*. In the first approach, the runtime is



© Aleksandar Prokopec and Fengyun Liu;

licensed under Creative Commons License CC-BY

32nd European Conference on Object-Oriented Programming (ECOOP 2018).

Editor: Todd Millstein; Article No. 3; pp. 3:1–3:32

Leibniz International Proceedings in Informatics



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

augmented with call stack introspection or the ability to swap call stacks during the execution of the program. We are aware of several such attempts in the context of the JVM runtime [14, 62], which did not become official due to the considerable changes required by the language runtime. In the second approach, the compiler transforms the program that uses coroutines into an equivalent program without coroutines. In Scheme, a similar control flow construct `call/cc` was supported by transforming the program into the continuation-passing style (CPS) [64]. The CPS transform can also be selectively applied to delimited parts of the program [3, 16, 17, 35, 56]. Mainstream languages like Python, C#, JavaScript, Dart and Scala offer suspension primitives such as generators, enumerators and `async-await`, which often target specific domains.

Coroutines based on metaprogramming. We explore a new transformation approach for coroutines that relies on the static metaprogramming support of the host language (in our case Scala), and assumes no call stack introspection or call stack manipulation support in the runtime (in our case JVM). The metaprogramming-based solution has several benefits:

- (1) The language runtime and the compiler do not need to be modified. This puts less pressure on the language and the runtime maintainers.
- (2) Since the metaprogramming API is typically standardized, the coroutine implementation is unaffected by the changes in the runtime or in the compiler.
- (3) The implementation does not need to be replicated for each supported backend. Our own implementation works with both the JVM runtime, and the Scala.JS browser backend.
- (4) Coroutines can be encapsulated as a standalone library. Our implementation in Scala is distributed independently from the standard Scala distribution.

We note that our approach is not strictly limited to metaprogramming – it can also be implemented inside the compiler. However, to the best of our knowledge, we are the first ones to implement and evaluate coroutines using a metaprogramming API.

Summary. Our coroutine model is statically typed, stackful, and delimited. Static typing improves program safety, stackfulness allows better composition, and delimitedness allows applying coroutines to selected parts of the program (this is explained further in Section 2). Regions of the program can be selectively marked as suspendable, without modifying or recompiling existing libraries. These regions represent first-class coroutines that behave similar to first-class function values. We show that the model generalizes many existing suspendable and asynchronous programming models. We extend this model with snapshots, and show that the extension allows expressing backtracking and continuations. Finally, we show that the metaprogramming approach has a reasonable performance (at most $2.11\times$ overheads compared to equivalent manually optimized code) by comparing it against alternative frameworks. We also formalize coroutines with snapshots and prove standard safety properties.

Contributions. The main novelties in this work are as follows:

- We show that *AST-level static metaprogramming support of the host language* is sufficient to implement first-class, typed, stackful, delimited coroutines that are reasonably efficient.
- We propose a new form of coroutines, namely *coroutine with snapshots*, which increases the power of coroutines. For example, it allows emulating backtracking and continuations.
- We formalize stackful coroutines with snapshots in λ_{\sim} by extending the simply typed lambda calculus, and we prove soundness of the calculus.

Some of the features of the proposed model, such as the typed coroutines and first-class composability, have been explored before in various forms [57, 34, 17, 16]. We build on earlier work, and we do not claim novelty for those features. However, to the best of our knowledge, the precise model that we describe is new, as we argue in Section 7 on related work.

Structure and organization. This paper is organized as follows:

- Section 2 describes the core primitives of the proposed programming model – coroutine definitions, coroutine instances, yielding, resuming, coroutine calls, and snapshots.
- Section 3 presents use-cases such as Erlang-style actors [66], `async-await` [23], Oz-style variables [65], event streams [20, 32], backtracking [36], and delimited continuations [56].
- In Section 4, we formalize coroutines with snapshots in $\lambda_{\rightsquigarrow}$ and prove its soundness.
- In Section 5, we describe the AST-level coroutine transformation implemented in Scala.
- In Section 6, we experimentally compare the performance of our implementation against Scala Delimited Continuations, Scala Async, iterators, and lazy streams.

Syntax. Our examples throughout this paper are in the Scala programming language [37]. We took care to make the paper accessible to a wide audience by using a minimal set of Scala features. The `def` keyword declares a method, while `var` and `val` declare mutable and final variables, respectively. Lambdas are declared with the right-arrow symbol `=>`. Type annotations go after the variable name, colon-delimited (`:`). Type parameters are put into square brackets `[]`. Parenthesis can be omitted from nullary and unary method calls.

2 Programming Model

In this section, we describe the proposed programming model through a series of examples.

Coroutine definitions. A subroutine is a sequence of statements that carry out a task. The same subroutine may execute many times during execution. When a program calls a subroutine, execution suspends at that callsite and continues in the subroutine. Execution at the callsite resumes only after the subroutine completes. For example, the program in Listing 1 declares a subroutine that doubles an integer, and then calls it with the argument 7.

■ **Listing 1** Subroutine example.

```
1 val dup = (x:Int) => { x + x }
2 dup(7)
```

■ **Listing 2** Coroutine example.

```
1 val dup =
2   coroutine { (x:Int) => x + x }
```

Upon calling `dup`, the subroutine does an addition, returns the result and terminates. When the execution resumes from the callsite, the subroutine invocation no longer exists.

Coroutines generalize subroutines by being able to suspend during their execution, so that their execution can be resumed later. In our implementation, a coroutine is defined inside the `coroutine` block. We show the coroutine equivalent of `dup` in Listing 2.

Yielding and resuming. Once started, the `dup` coroutine from Listing 2 runs to completion without suspending. However, a typical coroutine will suspend at least once. When it does, it is useful that it *yields* a value to the caller, explaining why it is suspended.

Consider the coroutine `rep` in Listing 3, which takes one argument `x`. The `rep` coroutine invokes the `yieldval` primitive to yield the argument `x` back to its caller, twice in a row.

3:4 Theory and Practice of Coroutines with Snapshots

■ Listing 3 Yielding example.

```
1 val rep = coroutine { (x:Int) =>
2   yieldval(x)
3   yieldval(x)
4 }
```

■ Listing 4 Execution states.

```
1 { ↑yieldval(7); yieldval(7) } =>
2 { yieldval(7); ↑yieldval(7) } =>
3 { yieldval(7); yieldval(7)↑ } =>
4 { yieldval(7); yieldval(7) }
```

Listing 4 show the states that the coroutine undergoes during its execution for $x=7$. The upward arrow (\uparrow) denotes the program counter. After getting invoked, the coroutine is paused in line 1 before the first `yieldval`. The caller resumes the coroutine (resuming is explained shortly), which then executes the next `yieldval` and yields the value 7 in line 2. The caller resumes the coroutine again, and the coroutine executes the last `yieldval` in line 3. The caller then resumes the coroutine the last time, and the coroutine terminates in line 4. The termination of a coroutine is similar to a termination of a subroutine – once the control flow reaches the end, the invocation of the corresponding coroutine no longer exists.

Delimited coroutines. As stated in the introduction, the proposed coroutine model is *delimited*. This means that the `yieldval` keyword can only occur inside a scope that is lexically enclosed with the `coroutine` keyword – a free `yieldval` results in a compiler error.

By itself, this restriction could hinder composability. Consider a hash table with closed addressing, which consists of an array whose entries are lists of elements (buckets). We would like a coroutine that traverses the elements of the hash table. Given a separately implemented `bucket` coroutine from Listing 5 that yields from a list, it is handy if a hash table coroutine can reuse this existing functionality by passing buckets from which to yield.

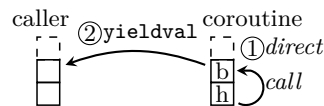
■ Listing 5 List coroutine.

```
1 val bucket = coroutine {
2   (b: List[Int]) =>
3   while (b != Nil) {
4     yieldval(b.head)
5     b = b.tail
6   }
7 }
```

■ Listing 6 Hash table coroutine.

```
1 val hashtable = coroutine {
2   (t: Array[List[Int]]) =>
3   var i = 0
4   while (i < t.length) {
5     bucket(t(i)); i += 1
6   }
7 }
```

Stackful coroutines. To allow composing separately written coroutines, it must be possible for one coroutine to call into another coroutine, but retain the same yielding context. The `hashtable` coroutine in Listing 6 traverses the array entries, and calls `bucket` for each entry. The two coroutines yield values together, as if they were a single coroutine.



Similar to ordinary subroutine calls, when the `hashtable` coroutine calls `bucket`, it must store its local variables and state. One way to achieve this is to use a call stack. When the program resumes the `hashtable` coroutine, it switches from the normal program call stack to a separate call stack that belongs to the resumed coroutine instance. The `hashtable` coroutine then pushes its state to the stack, and passes the control flow to the `bucket` coroutine (① in the figure below). The `bucket` coroutine stores its state to the stack and yields to the same caller that originally resumed the `hashtable` coroutine (②).

By saying that our coroutine model is *stackful*, we mean that coroutines are able to call each other, and yield back to the same resume-site [34]. In our implementation, this is enabled with an artificial call stack, as explained in Section 5.

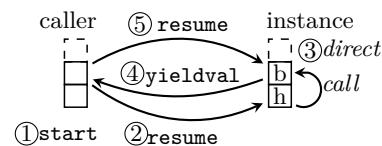
Importantly, a coroutine call can only occur in a scope that is lexically enclosed with the `coroutine` keyword (akin to `yieldval`). Only a coroutine body can call another coroutine – a free call is a compiler error. A natural question follows: how does a normal program create a new coroutine instance?

Coroutine instances. Similar to how invoking a *subroutine* creates a running instance of that subroutine, starting a *coroutine* creates a new *coroutine instance*. A subroutine's execution is not a program value – it cannot be observed or controlled. However, after creating a coroutine instance, the caller must interact with it by reading the yielded values and resuming. Therefore, it is necessary to treat the coroutine instance as a program value.

To distinguish between normal coroutine calls and creating a coroutine instance, we introduce a separate `start` keyword, as a design choice. The instance encapsulates a new call stack. After `start` returns a new coroutine instance, the caller can invoke `resume` and `value` on the instance. By returning `true`, the `resume` indicates that the instance yielded, and did not yet terminate. In this case, invoking `value` returns the last yielded value.

■ **Listing 7** Starting and resuming.

```
1 val i = hashtable.start(array)
2 var sum: Int = 0
3 while (i.resume) sum += i.value
```



Listing 7 shows how to use the hash table coroutine to compute the sum of the hash table values. A new coroutine instance is started in line 1 using the `hashtable` coroutine. In the `while` loop in line 3, the values are extracted from the instance until `resume` returns `false` (subsequent calls to `resume` result in a runtime error in our implementation). The figure on the right shows the creation of the instance, followed by two `resume` calls.

Typed coroutines. In the example in Listing 7, the `sum` variable in line 2 is integer-typed. Therefore, the right hand of the assignment in line 3 must also have the integer type. This illustrates that it is useful to reason about the type of the yielded values.

A coroutine type encodes the types P_i of its parameters, the return type R , and the *yield type* Y of the yielded values, also called the *output type* [57]. Its general form is $(P_1, \dots, P_N) \rightsquigarrow (Y, R)$ ¹. In the following, we annotate the `dup` coroutine from Listing 2:

```
val dup: Int~>(Nothing, Int) = coroutine { (x: Int) => x + x }
```

Since this coroutine does not yield any values, its yield type is `Nothing`, the bottom type in Scala. A coroutine `once`, which yields its argument once, has the yield type `Int`:

```
val once: Int~>(Int, Unit) = coroutine { (x: Int) => yieldval(x) }
```

The `bucket` and `hashtable` coroutines from Listings 5 and 6 have the following types:

```
val bucket: List[Int]~>(Int, Unit) = ...
val hashtable: Array[List[Int]]~>(Int, Unit) = ...
```

Coroutine instances have a separate type $Y \leftarrow \rightsquigarrow R$, where Y is the yield type, and R is the return type. For example, starting `once` produces an instance with the type `Int<~>Unit`:

```
val i: Int<~>Unit = once.start(7)
```

Instances of the `bucket` and `hashtable` coroutines also have the type `Int<~>Unit`.

¹ We note that Scala allows operator syntax, such as `~>`, when declaring custom data types.

Snapshots. We extend standard coroutines with the `snapshot` operation, which takes a coroutine instance and returns a duplicated instance with exactly the same execution state.

```
val i1: Int<~>Unit = once.start(7)
val i2: Int<~>Unit = i1.snapshot
```

In the previous example, the coroutine instance `i1` is duplicated to `i2`. Subsequent coroutine operations can be called independently on the two coroutine instances.

3 Use Cases

Having explained what type-safe, delimited and stackful means, we motivate our design choices with observations and concrete examples. The goal of this section is to show how specific primitives help express other kinds of suspendable computations.

► **Observation 1.** Stackful coroutines allow composing suspendable software modules.

The hash table example from Listings 5 and 6 demonstrated why composing different modules is useful, and the remaining examples in this section reinforce this.

► **Observation 2.** Stackful coroutines simplify the interaction with recursive data types.

Iterators. Data structure iterators are one of the earliest applications of coroutines [29]. We can implement an iterator coroutine `it` for binary trees as shown in Listing 8:

■ **Listing 8** Tree iterator implementation.

```
1 val it = coroutine { (t:Tree) =>
2   if (t.fst != null) it(t.fst)
3   yieldval(t.element)
4   if (t.snd != null) it(t.snd) }
```

■ **Listing 9** Symmetric coroutine.

```
1 type Sym[R] = Sym[R] <~> R
2 def run(i: Sym[R]): R =
3   if (i.resume) run(i.value)
4   else i.result
```

► **Observation 3.** Asymmetric coroutines can be used to express symmetric coroutines.

Symmetric coroutines. In a programming model with symmetric coroutines, there is no `resume` primitive. Instead, a symmetric coroutine always yields the next coroutine instance from which to continue. As shown in Listing 9, a symmetric coroutine instance can be expressed with the recursive type `Sym[R]`. This was observed before [34].

► **Observation 4.** Coroutine calls and coroutine return values, together with the `yieldval` primitive, allow encoding alternative forms of suspension primitives.

The core idea is to express a suspension primitive as a special coroutine. This coroutine yields a value that allows the resume-site to communicate back. Once resumed, the coroutine *returns another value* to its caller coroutine. We show several examples of this pattern.

Async-await. A future is an entity that initially does not hold a value, but may asynchronously attain it at a later point. Future APIs are usually callback-based – the code that handles the future value is provided as a function. In Scala, type `Future` exposes the `onSuccess` method, which takes a callback and calls it once the value is available, and `value`, which can be used to access the future value if available. `Promise` is the future’s writing end, and it exposes `success`, which sets the value of the corresponding future at most once [22].

■ **Listing 10** Callback-style API.

```

1 val token: Future[Token] =
2   authenticate()
3 token.onSuccess { t =>
4   val session: Future[Session] =
5     sessionFor(t)
6   session.onSuccess(useSession)
7 }

```

■ **Listing 11** Async-await-style API.

```

1 async {
2   val token: Token =
3     await { authenticate() }
4   val session: Session =
5     await { sessionFor(token) }
6   useSession(session)
7 }

```

In Listing 10, the `authenticate` method returns a future with a `Token` value. Once the token arrives, the callback function (passed to the `onSuccess` method) calls `sessionFor` to obtain a session future. The program continues in the `useSession` callback. The direct-style async-await version, shown in Listing 11, relies on the `async` statement, which starts asynchronous computations, and the `await`, which suspends until a value is available.

We use coroutines to replicate Scala’s Async framework [23]. The `await` method emulates the suspension primitive – it creates a coroutine that yields a future and returns its value. It assumes that the resume-site invokes `resume` only *after* the future is completed.

■ **Listing 12** The `await` primitive.

```

1 def await[T]: Future[T] ~> (Future[T], T) =
2   coroutine { (f: Future[T]) => yieldval(f); f.value }

```

The `async` method interacts with `await` from the resume-site. For a given computation `b`, it starts an instance, and executes it asynchronously in a `Future`. Whenever the computation `b` yields a future, a callback is recursively installed. If `resume` returns `false`, the resulting future is completed with the evaluation result of `b` (`async` itself must return a future).

■ **Listing 13** The `async` primitive.

```

1 def async[R](b: () ~> (Future[Any], R)): Future[R] = {
2   val i = b.start()
3   val p = new Promise[R]
4   @tailrec def loop(): Unit =
5     if (i.resume) i.value.onSuccess(loop) else p.success(i.result)
6   Future { loop() }
7   p.future
8 }

```

Erlang-style actors. Actor frameworks for the JVM are unable to implement exact Erlang-style semantics, in which the `receive` statement can be called anywhere in the actor [21]. For example, Akka exposes a top-level `receive` method [1], which returns a partial function used to handle messages. This function can be swapped with the `become` statement.

Listing 14 shows a minimal Akka actor example that implements a server. The server starts its execution in the `receive` method, which awaits a password from the user. If the password is correct, the top-level event-handling loop becomes the `loggedIn` method, which accepts GET requests. When the `Logout` message arrives, the actor stops. Listing 15 shows an equivalent Erlang-style actor, in which the control flow is more apparent. The `receive` method has the role of a suspension primitive – it pauses the actor until a message arrives.

■ **Listing 14** Akka-style actor.

```

1 class Server extends Actor {
2   def receive = {
3     case Login(pass) =>
4       assert(isCorrect(pass))

```

```

5     become(loggedIn) }
6   def loggedIn = {
7     case Get(url) => serve(url)
8     case Logout() => stop() } }

```

■ **Listing 15** Erlang-style actor.

```
1 def server() = {
2   val Login(pass) = receive()
3   assert(isCorrect(pass))
4   while (true) receive() match {
```

```
5     case Get(url) => serve(url)
6     case Logout() => stop()
7   }
8 }
```

As shown in the appendix, the `receive` coroutine follows a similar pattern as `async-await` – `receive` yields an object into which the top-level loop can insert the message.

Event streams. First-class Rx-style event streams [32] expose a set of declarative transformation combinators. As an example, consider how to collect a sequence of points when dragging the mouse. The mouse events are represented as an event stream value. Dragging starts when the mouse is pressed down, and ends when released. In Listing 16, the `after` combinator removes a prefix of events, and `until` removes a suffix. The first drag event’s `onEvent` callback creates a `Curve` object, and the last event saves the curve.

■ **Listing 16** Rx-style streams.

```
1 val drag = mouse.after(_ .isDown)
2   .until(_ .isUp)
3 drag.first.onEvent { e =>
4   val c = new Curve(e.x, e.y)
5   drag.onEvent(
6     e => c.add(e.x, e.y))
7 drag.last.onEvent(
8   e => saveCurve(c) }
```

■ **Listing 17** Direct-style streams.

```
1 var e = mouse.get
2 while (!e.isDown) e = mouse.get
3 val c = new Curve(e.x, e.y)
4 while (e.isDown) {
5   e = mouse.get
6   c.add(e.x, e.y)
7 }
8 saveCurve(c)
```

The equivalent direct-style program in Listing 17 uses the `get` coroutine to suspend the program until an event arrives. We show the implementation of `get` in the appendix.

Oz-style single-assignment variables. A variable in the Oz language [65] can be assigned only once. Reading a single-assignment variable suspends execution until some other thread assigns a value. Assigning a value to the variable more than once is an error.

■ **Listing 18** Oz-style variable read.

```
1 @volatile var state: AnyRef =
2   List.empty()
3
4 val get = coroutine { () =>
5   if (READ(state).is[List])
6     yieldval(this)
7   READ(state).as[ElemType]
8 }
```

■ **Listing 19** Oz-style variable write.

```
1 @tailrec def set(y: ElemType) {
2   val x = READ(state)
3   if (x.is[List]) {
4     if (CAS(state, x, y))
5       for (i <- x.as[List])
6         schedule(i)
7     else set(y)
8   } else throw Reassigned }
```

Internally, a single-assignment variable has some `state`, which is either the list of suspended threads or a value. When `get` from the Listing 18 is called, `state` is atomically read in line 6. If the state is a list, coroutine is yielded to the scheduler, which atomically adds the new suspended thread to the list (not shown). The coroutine is resumed when the value becomes available – method `set` in Listing 19 tries to atomically replace the list with the value of type `ElemType`. If successful, it schedules the suspended threads for execution.

► **Observation 5.** Snapshots enable backtracking and allow emulating full continuations.

Backtracking. Testing frameworks such as ScalaCheck [36] rely on backtracking to systematically explore the test parameter space. ScalaCheck uses a monadic-style API to compose the parameter generators. Listing 20 shows a ScalaCheck-style test that first creates a generator for number pairs in a Scala `for`-comprehension, and then uses the generator in a commutativity test. The `pairs` generator is created from the intrinsic `ints` generator.

In the direct-style variant in Listing 21, the `ints` generator is a coroutine that yields and captures the execution snapshot. Therefore, it can be called from anywhere inside the test.

■ **Listing 20** Monadic ScalaCheck test.

```

1 val pairs =
2   for {
3     x <- ints(0 until MAX_INT)
4     y <- ints(0 until MAX_INT)
5   } yield (x, y)
6
7 forAll(pairs) { pair =>
8   val (a, b) = pair
9   assert(a * b == b * a)
10 }
11
```

■ **Listing 21** Direct-style ScalaCheck test.

```

1 test {
2   val a = ints(0 until MAX_INT)
3   val b = ints(0 until MAX_INT)
4   assert(a * b == b * a)
5 }
```

■ **Listing 22** Positive-definite matrix test.

```

1 val pd = coroutine { (m:Mat) =>
2   val x = nonZeroVector(m.size)
3   assert(x.t * m * x > 0)
4 }
```

Moreover, generator late-binding allows modularizing the properties. Listing 22 shows a modular *positive-definite* matrix property `pd`: given a matrix M , value $x^T M x$ is positive for any non-zero vector x . The `pd` coroutine can be called from within other tests. Importantly, note that the vector x is generated *from within* the test. This is hard to achieve in the standard ScalaCheck tests, since their generators require prior knowledge about the vector x .

Consider implementing the `test` and the `ints` primitives from Listing 21. The key idea is as follows: each time a test calls a generator, it suspends and yields a list of environment setters. Each environment setter is a function that prepares a value to return from the generator. The resume-site runs each environment setter, creates a snapshot and resumes it.

■ **Listing 23** Direct-style ScalaCheck.

```

1 type Test =
2   () -> (List[() => Unit], Unit)
3
4 type Instance =
5   List[() => Unit] <-> Unit
6
7 def backtrack(i: Instance) = {
8   if (i.resume)
9     for (setEnv <- i.value) {
10      setEnv()
11      backtrack(i.snapshot)
12    }
13 }
```

■ **Listing 24** Direct-style ScalaCheck, cont.

```

1 val ints = coroutine {
2   (xs: List[Int]) =>
3     var env: Int = _
4     val setEnvs =
5       xs.map(x => () => env = x)
6     yieldval(setEnvs)
7     env
8 }
9
10 def test(t: Test) = {
11   val instance = t.start()
12   backtrack(instance)
13 }
```

In Listing 23, we first declare two type aliases `Test` and `Instance`, which represent a test coroutine and a running test instance. Their yield type is a list of environment setters `List[() => Unit]`. The `backtrack` subroutine takes a running test instance, and resumes it. If the instance yields, then the test must have called a generator, so `backtrack` traverses the environment setters and recursively resumes a snapshot for each of them. Thus, each recursive call to `backtrack` represents a node in the respective backtracking tree.

The `ints` generator in Listing 24 is a coroutine that takes a list of integers `xs` to choose from. It starts by creating a local variable `env`, and a list of functions that set `env` (one for each integer in `xs`). The generator then yields this list. Each time `ints` gets resumed, the `env` variable is set to a proper value, so it is returned to the test that called it.

$t ::=$	terms:	$T ::=$	types:
$(x:T) \Rightarrow t$	abstraction	$T \Rightarrow T$	function type
$t(t)$	application	$T \overset{T}{\rightsquigarrow} T$	coroutine type
x	variable	$T \longleftrightarrow T$	instance type
$()$	unit value	Unit	unit type
$(x:T) \overset{T}{\rightsquigarrow} t$	coroutine	\perp	bottom type
$\text{yield}(t)$	yielding	$r ::=$	runtime terms:
$\text{start}(t, t)$	starting	i	instance
$\text{resume}(t, t, t, t)$	resuming	$\langle t, v, v, v \rangle_i$	resumption
$\text{snapshot}(t)$	snapshot	$\llbracket t \rrbracket_v$	suspension
$\text{fix}(t)$	recursion	\emptyset	empty term
i	instance	$v ::=$	values:
$\langle t, v, v, v \rangle_i$	resumption	$(x:T) \Rightarrow t$	abstraction
$\llbracket t \rrbracket_v$	suspension	$()$	unit value
\emptyset	empty term	$(x:T) \overset{T}{\rightsquigarrow} t$	coroutine
		i	instance
		\emptyset	empty term

■ **Figure 1** Syntax and types of the $\lambda_{\rightsquigarrow}$ calculus.

Continuations. Shift-reset-style delimited continuations use the `reset` operator to delimit program regions for the CPS-transform [12]. The `shift` operator, which takes a function whose input is the continuation, can be used inside these regions. We sketch the implementation of `shift` and `reset` similar to those in Scala delimited continuations [56].

```

1 type Shift = (() => Unit) => Unit
2 def reset(b: () ~> (Shift, Unit)): Unit = {
3   def continue(i: Shift <~> Unit) =
4     if (i.resume) i.value(() => continue(i.snapshot))
5     continue(b.start())
6 }
7 def shift: Shift ~> (Shift, Unit) =
8   coroutine { (b: Shift) => yieldval(b) }
```

The type alias `Shift` represents continuation handlers – functions that take continuations of the current program. The `reset` operator takes a coroutine that can yield a `Shift` value. It starts a new coroutine instance and resumes it. If this instance calls `shift` with a continuation handler, the handler is yielded back to `reset`, which creates an instance snapshot and uses it to create a continuation. The continuation is passed to the continuation handler. The use of `snapshot` is required, as the continuation can be invoked multiple times.

4 Formal Semantics

This section presents the $\lambda_{\rightsquigarrow}$ (pron. *lambda-squiggly*) calculus that captures the core of the programming model from Section 2. This calculus is an extension of the simply-typed lambda-calculus. The complete formalization, along with the proofs of the progress and preservation theorems, is given in the corresponding tech report [51].

Syntax. Figure 1 shows the syntax. The abstraction, application and variable terms are standard. The *coroutine* term represents coroutine declarations. The *yield* term corresponds

$$\begin{array}{c}
\frac{\Sigma|\Gamma, x:T_1 \vdash t_2:T_2|\perp}{\Sigma|\Gamma \vdash (x:T_1) \Rightarrow t_2 : T_1 \Rightarrow T_2|\perp} \quad (T\text{-ABS}) \qquad \frac{\Sigma|\Gamma \vdash t_1:T_2 \Rightarrow T_1|T_y}{\Sigma|\Gamma \vdash t_1(t_2) : T_1|T_y} \quad (T\text{-APP}) \qquad \frac{x:T \in \Gamma}{\Sigma|\Gamma \vdash x:T|\perp} \quad (T\text{-VAR}) \qquad \frac{\Sigma|\Gamma \vdash t:T|\perp}{\Sigma|\Gamma \vdash t:T|T_y} \quad (T\text{-CTX}) \\
\\
\frac{\Sigma|\Gamma \vdash () : \mathbf{Unit}|\perp}{(T\text{-UNIT})} \qquad \frac{\Sigma|\Gamma, x:T_1 \vdash t_2:T_2|T_y}{\Sigma|\Gamma \vdash (x:T_1) \overset{T_y}{\rightsquigarrow} t_2 : T_1 \overset{T_y}{\rightsquigarrow} T_2|\perp} \quad (T\text{-COROUTINE}) \qquad \frac{\Sigma|\Gamma \vdash t_1:T_1 \overset{T_y}{\rightsquigarrow} T_2|T_w}{\Sigma|\Gamma \vdash t_2:T_1|T_w} \quad (T\text{-START}) \\
\\
\frac{\Sigma|\Gamma \vdash t:T|T}{\Sigma|\Gamma \vdash \mathbf{yield}(t) : \mathbf{Unit}|T} \quad (T\text{-YIELD}) \qquad \frac{\Sigma|\Gamma \vdash t:T_y \rightsquigarrow T_2|T_w}{\Sigma|\Gamma \vdash \mathbf{snapshot}(t) : T_y \rightsquigarrow T_2|T_w} \quad (T\text{-SNAPSHOT}) \qquad \frac{\Sigma|\Gamma \vdash t:T \Rightarrow T|\perp}{\Sigma|\Gamma \vdash \mathbf{fix}(t) : T|\perp} \quad (T\text{-FIX}) \\
\\
\frac{\Sigma|\Gamma \vdash t_1:T_y \rightsquigarrow T_2|T_w \quad \Sigma|\Gamma \vdash t_2:T_2 \overset{T_w}{\rightsquigarrow} T_R|T_w}{\Sigma|\Gamma \vdash t_3:T_y \overset{T_w}{\rightsquigarrow} T_R|T_w} \quad \frac{\Sigma|\Gamma \vdash t_4:\mathbf{Unit} \overset{T_w}{\rightsquigarrow} T_R|T_w}{\Sigma|\Gamma \vdash \mathbf{resume}(t_1, t_2, t_3, t_4) : T_R|T_w} \quad (T\text{-RESUME}) \qquad \frac{\Sigma|\Gamma \vdash t_1:T_2 \overset{T_y}{\rightsquigarrow} T_1|T_y \quad \Sigma|\Gamma \vdash t_2:T_2|T_y}{\Sigma|\Gamma \vdash t_1(t_2) : T_1|T_y} \quad (T\text{-APPCOR}) \\
\\
\frac{\Sigma|\Gamma \vdash t:T|T_y \quad \Sigma|\Gamma \vdash v:T_y|\perp}{\Sigma|\Gamma \vdash \llbracket t \rrbracket_v : T|T_y} \quad (T\text{-SUSPENSION}) \qquad \frac{\Sigma(i) = T_y \rightsquigarrow T_2 \quad \Sigma|\Gamma \vdash t_1:T_2|T_y \quad \Sigma|\Gamma \vdash v_2:T_2 \overset{T_w}{\rightsquigarrow} T_R|\perp}{\Sigma|\Gamma \vdash v_3:T_y \overset{T_w}{\rightsquigarrow} T_R|\perp} \quad \frac{\Sigma|\Gamma \vdash v_4:\mathbf{Unit} \overset{T_w}{\rightsquigarrow} T_R|\perp}{\Sigma|\Gamma \vdash \langle t_1, v_2, v_3, v_4 \rangle_i : T_R|T_w} \quad (T\text{-RESUMPTION}) \\
\\
\frac{\Sigma(i) = T_y \rightsquigarrow T_2}{\Sigma|\Gamma \vdash i:T_y \rightsquigarrow T_2|\perp} \quad (T\text{-INSTANCE}) \qquad \Sigma|\Gamma \vdash \emptyset : T|\perp \quad (T\text{-EMPTY})
\end{array}$$

■ **Figure 2** Typing rules for the $\lambda_{\rightsquigarrow}$ calculus.

to the `yieldval` statement shown earlier. The `start` term is as before, but uses prefix syntax. The `resume` term encodes both the `resume` and the `value` from Section 2. This is because resuming an instance can complete that instance (in our implementation, `resume` returns `false`), it can result in a yield (previously, `true`), or fail because an instance had already completed earlier (in our implementation, an exception is thrown). In $\lambda_{\rightsquigarrow}$, the `resume` term therefore accepts four arguments: the coroutine instance, the result handler, the yield handler, and the handler for the already-completed case. The `fix` term supports recursion [39].

The calculus differentiates between *user terms*, which appear in user programs, and *runtime terms*, which only appear during the program evaluation. A label i is used to represent a coroutine instance. Each coroutine instance has an evaluation state, which changes over the lifetime of the instance. A resumed instance i is represented by the *resumption term* $\langle t, v, v, v \rangle_i$. A term that yielded a value v , and is about to be suspended, is represented by the *suspension term* $\llbracket t \rrbracket_v$. Finally, the *empty term* \emptyset is used in the store μ (shown shortly) when encoding terminated coroutines. The abstraction term $(x:T) \Rightarrow t$, the unit term $()$, the coroutine definition $(x:T) \overset{T_y}{\rightsquigarrow} t$, the instance label i and the empty term \emptyset are considered values.

Note that the calculus distinguishes between standard function types $T_1 \Rightarrow T_2$ and coroutine types of the form $T_1 \overset{T_y}{\rightsquigarrow} T_2$, where T_y is the type of values that the coroutine may yield. Coroutine instances have the $T_y \rightsquigarrow T_2$ type, and a unit value has the type `Unit`. The bottom type \perp is used to represent the fact that a term does not yield.

Example. Recall the `rep` coroutine from Listing 3. We can encode it in $\lambda_{\rightsquigarrow}$ as follows:

$$(x:\mathbf{Int}) \overset{\mathbf{Int}}{\rightsquigarrow} ((u:\mathbf{Unit}) \overset{\mathbf{Int}}{\rightsquigarrow} \mathbf{yield}(x))(\mathbf{yield}(x))$$

The encoding uses a standard trick to sequence two statements [39], but instead of a regular lambda, relies on a coroutine to ignore the result of the first statement. Starting this coroutine creates an instance i , whose current term is saved in the store:

$$\mathbf{start}((x:\mathbf{Int}) \overset{\mathbf{Int}}{\rightsquigarrow} ((u:\mathbf{Unit}) \overset{\mathbf{Int}}{\rightsquigarrow} \mathbf{yield}(x))(\mathbf{yield}(x)), 7) \rightarrow i$$

Assume that we now resume this instance once. We provide three handlers to **resume**. We show the complete yield handler (here, identity), and name the other two c_2 and c_4 :

$$\begin{aligned} & \mathbf{resume}(i, c_2, (x:\mathbf{Int}) \overset{\perp}{\rightsquigarrow} x, c_4) \rightarrow \\ & \langle ((u:\mathbf{Unit}) \overset{\mathbf{Int}}{\rightsquigarrow} \mathbf{yield}(7))(\mathbf{yield}(7)), c_2, (x:\mathbf{Int}) \overset{\perp}{\rightsquigarrow} x, c_4 \rangle \rightarrow \\ & \langle ((u:\mathbf{Unit}) \overset{\mathbf{Int}}{\rightsquigarrow} \mathbf{yield}(7))(\llbracket () \rrbracket_7), c_2, (x:\mathbf{Int}) \overset{\perp}{\rightsquigarrow} x, c_4 \rangle \rightarrow \\ & \langle \llbracket ((u:\mathbf{Unit}) \overset{\mathbf{Int}}{\rightsquigarrow} \mathbf{yield}(7))(\llbracket () \rrbracket) \rrbracket_7, c_2, (x:\mathbf{Int}) \overset{\perp}{\rightsquigarrow} x, c_4 \rangle \rightarrow ((x:\mathbf{Int}) \overset{\perp}{\rightsquigarrow} x)(7) \rightarrow 7 \end{aligned}$$

Typing. Before showing the typing rules, we introduce the *instance typing* Σ , which tracks coroutine instance types, and is used alongside the standard typing context Γ .

► **Definition 6** (Instance typing). The *instance typing* Σ is a sequence of coroutine instance labels and their types $i:T$, where comma $(,)$ extends a typing with a new binding.

► **Definition 7** (Typing relation). The *typing relation* $\Sigma|\Gamma \vdash \mathfrak{t}:T|\mathbf{T}_y$ in Fig. 2 is a relation between the instance typing Σ , the typing context Γ , the term \mathfrak{t} , its type T , and the *yield type* \mathbf{T}_y , where \mathbf{T}_y is the type of values that may be yielded when evaluating \mathfrak{t} .

We inspect the most important rules here, and refer the reader to the tech report [51] for a complete discussion. The T-ABS rule is the modification of the standard abstraction typing rule. Note that the yield type of the function body **must be** \perp . This is because $\lambda_{\rightsquigarrow}$ models *delimited coroutines* – if a **yield** expression occurs, it must instead be lexically enclosed within a coroutine term (which corresponds to the **coroutine** statement). We emphasize that $\lambda_{\rightsquigarrow}$ nevertheless models stackfulness – a **yield** can still cross coroutine boundaries at runtime if a coroutine calls another coroutine, as illustrated in Section 3, and explained shortly.

Note further, that the T-APP rule permits a non- \perp type on the subterms, since the reduction of the function and its arguments is itself allowed to yield. A non-yielding term can be assumed to yield any type by the T-CTX rule. Given a term whose type is T and yield type is also T , the T-YIELD rule types a **yield** expression as **Unit** with the yield type T .

The T-COROUTINE rule allows the body to yield a value of the type \mathbf{T}_y , which must correspond to the yield type of the coroutine. The coroutine itself gets a \perp yield type (the coroutine definition effectively swallows the yield type). Consider now the T-APPCOR rule, which is similar to the standard T-APP rule for functions. To directly call another coroutine \mathfrak{t}_1 , its yield type \mathbf{T}_y must correspond to the yield type at the callsite.

Last, we examine the runtime term typing rules. The T-SUSPENSION rule requires that the yielded value v has the type \mathbf{T}_y , and that the suspension has the same type and yield type as the underlying suspended term \mathfrak{t} . The T-INSTANCE rule requires that the instance typing Σ contains a corresponding type for i . Finally, the T-RESUMPTION term has the type \mathbf{T}_R that corresponds to the return types of the handler coroutines \mathfrak{t}_2 , \mathfrak{t}_3 and \mathfrak{t}_4 . The corresponding yield type is \mathbf{T}_w , which is generally different from the yield type \mathbf{T}_y that the coroutine instance evaluation \mathfrak{t}_1 can yield. The empty term can be assigned any type.

► **Definition 8** (Well-typed program). A term \mathfrak{t} is *well-typed* if and only if $\exists T, \mathbf{T}_y, \Sigma$ such that $\Sigma|\emptyset \vdash \mathfrak{t}:T|\mathbf{T}_y$. A term \mathfrak{t} is a *well-typed user program* if \mathfrak{t} is well-typed and $\mathbf{T}_y = \perp$.

$$\begin{array}{c}
\frac{i \notin \text{dom}(\mu)}{\text{start}((x:T_1) \xrightarrow{T_y} t, v) | \mu \rightarrow i | \mu, i \triangleright [x \mapsto v]t} \quad \frac{i_2 \notin \text{dom}(\mu) \quad i_1 \neq i_2}{\text{snapshot}(i_1) | \mu, i_1 \triangleright t \rightarrow i_2 | \mu, i_1 \triangleright t, i_2 \triangleright t} \\
\text{(E-START)} \qquad \qquad \qquad \text{(E-SNAPSHOT)} \\
\text{yield}(v) | \mu \rightarrow [()]_v | \mu \qquad \langle [t_1]_v, v_2, v_3, v_4 \rangle_i | \mu, i \triangleright [t_0]_{v'} \rightarrow v_3(v) | \mu, i \triangleright t_1 \\
\text{(E-YIELD)} \qquad \qquad \qquad \text{(E-CAPTURE)} \\
\frac{t \neq [t_0]_{\emptyset}}{\text{resume}(i, v_2, v_3, v_4) | \mu, i \triangleright t \rightarrow \langle t, v_2, v_3, v_4 \rangle_i | \mu, i \triangleright [t]_{\emptyset}} \text{(E-RESUME1)} \\
\text{resume}(i, v_2, v_3, v_4) | \mu, i \triangleright [t_0]_{\emptyset} \rightarrow v_4(()) | \mu, i \triangleright [t_0]_{\emptyset} \text{(E-RESUME2)} \\
\langle v, v_2, v_3, v_4 \rangle_i | \mu, i \triangleright [t_0]_{v'} \rightarrow v_2(v) | \mu, i \triangleright [v]_{\emptyset} \text{(E-TERMINATE)}
\end{array}$$

■ **Figure 3** A subset of evaluation rules in the $\lambda_{\rightsquigarrow}$ calculus.

Semantics. Before showing the operational semantics, we introduce the concept of a coroutine store μ , which is used to track the evaluation state of the coroutine instances.

► **Definition 9** (Coroutine store). A *coroutine store* μ is a sequence of coroutine instance labels i and their respective evaluation terms t , where the comma operator $(,)$ extends the coroutine store with a new binding $i \triangleright t$.

We only show a subset with the most important evaluation rules in Fig. 3, and present the complete set of rules in the tech report [51]. The E-START rule takes a coroutine value and an argument, and uses them to create a new coroutine instance i , where i is a fresh label. The coroutine store μ is modified to include a binding from i to the coroutine body t after substitution. Such a coroutine instance i can then be resumed by the E-RESUME1 rule, which reduces a `resume` expression into an resumption term $\langle t, v_2, v_3, v_4 \rangle_i$. Note the convention that an executing or a terminated coroutine has a suspension term $[t_0]_{\emptyset}$ in the coroutine store μ . The E-RESUME1 rule applies if and only if $t \neq [t_0]_{\emptyset}$. If the instance is terminated, the E-RESUME2 rule applies instead, which just invokes the ‘callback’ v_4 to handle that case.

Consider now what happens when a resumption term yields. By E-YIELD, the expression `yield(v)` reduces to a suspended unit value that is yielding the value v . A suspension term then spreads across the surrounding terms. The following two example reductions spread the suspension across an application. There is one such rule for each non-value syntax form.

$$[[t_1]_v(t_2)] | \mu \rightarrow [[t_1(t_2)]_v] | \mu$$

$$v_1([t_2]_v) | \mu \rightarrow [v_1(t_2)]_v | \mu$$

Once the suspension reaches the coroutine resumption term, the E-CAPTURE reduces it to a call to the yield handler v_3 , and puts the term in the suspension into the store μ .

Safety. We now state the safety properties of $\lambda_{\rightsquigarrow}$. Complete proofs are given in the corresponding tech report article [51].

► **Definition 10** (Well-typed coroutine store). A coroutine store μ is well-typed with respect to the instance typing Σ , denoted $\Sigma \vdash \mu$, if and only if it is true that $\forall i \in \text{dom}(\mu)$, $\Sigma(i) = T_y \rightsquigarrow T_2 \Leftrightarrow \Sigma | \emptyset \vdash \mu(i) : T_2 | T_y$, and $\text{dom}(\Sigma) = \text{dom}(\mu)$.

► **Theorem 11 (Progress).** *Suppose that t is a closed, well-typed term for some T and Σ . Then, either t is a value, or t is a suspension term $\llbracket t \rrbracket_v$, or, for any store μ such that $\Sigma \vdash \mu$, there is some term t' and store μ' such that $t|\mu \rightarrow t'|\mu'$.*

► **Theorem 12 (Preservation).** *If a term and the coroutine store are well-typed, that is, $\Sigma|\Gamma \vdash t:T|T_y$, and $\Sigma|\Gamma \vdash \mu$, and if $t|\mu \rightarrow t'|\mu'$, then there exists $\Sigma' \supseteq \Sigma$ such that $\Sigma'|\Gamma \vdash t':T|T_y$ and $\Sigma'|\Gamma \vdash \mu'$.*

► **Corollary 1 (Yield safety).** *If a user program \mathfrak{t}_u is well-typed, then it does not evaluate to a suspension term of the form $\llbracket \mathfrak{t} \rrbracket_v$.*

5 Implementation

This section describes our metaprogramming-based coroutine implementation in Scala, which consists of a runtime library and an AST-level transformation.

5.1 Preliminaries

Our implementation relies on Scala Macros [9], the metaprogramming API in Scala². The following are the key metaprogramming features of Scala Macros that our transformation scheme relies on. First, it must be possible to declare *macro definitions* that take ASTs as input arguments and return transformed ASTs. Second, it must be possible to invoke such macro definitions from user programs, which makes the compiler execute the macro with its argument expressions and replace the macro invocation with the resulting ASTs. Third, it must be possible to decompose and compose ASTs inside the macro definition. Finally, it must be possible to inspect the types of the expressions passed to the macro, and reason about symbol identity. The following is an example of a macro definition:

```
1 def log(msg: String): Unit = macro log_impl
2 def log_impl(c: Context)(msg: c.Tree): c.Tree =
3   q"""if (loggingEnabled) info(currentTime() + ": " + $msg)"""
```

In the above, we declared a `log` macro with the macro implementation `log_impl`, which takes the corresponding `Tree` of `msg` as argument. The `log_impl` method uses the quasiquote notation `q"""` to build an AST [61], which in turn checks if logging is enabled before constructing the command line output from the `msg` string. Values are interpolated into the AST using the `$` notation, as is the case with the expression `$msg` above.

Scala macro definitions can be packaged into libraries. Our coroutine implementation is therefore a macro library, which the user program can depend on. User coroutines implemented with our coroutine library can be similarly packaged into third party libraries.

5.2 Runtime Model

When yielding, the coroutine instance must store the state of the local variables. When resuming, that state must be retrieved into the local variables. Since coroutines are stackful, i.e. they support nested coroutine calls, it is necessary to store the entire coroutine call stack.

² We have an ongoing second implementation that relies on a newer Scala Macros API, but we are using the original Scala Macros for the evaluation purposes in this paper.

Call stacks. Our implementation uses arrays to represent coroutine call stacks. A call stack is divided into a sequence of coroutine frames, which are similar to method invocation frames. Each frame stores the following:

- (1) pointer to the coroutine object (i.e. `Coroutine` block),
- (2) the position in the coroutine where the last yield occurred,
- (3) the local variables and the return values.

A coroutine call stack can be implemented as a single contiguous memory area. Scala is constrained by the JVM platform, on which arrays contain either object references or primitive values, but not both. Hence, our implementation separates coroutine descriptors, program counters and local variables into reference and value stacks.

It would be costly to create a large call stack whenever a coroutine instance starts. Many coroutines only need a single or several frames, so we set the initial stack size to 4 entries. When pushing, a stack overflow check potentially reallocates the stack, doubling its size. In the worst case, the reallocation overhead is $2\times$ compared to an optimally sized stack.

Coroutine instance class. A coroutine instance is represented by the `Instance` class shown in Listing 25. A new instance is created by the `start` method. In addition to the call stack, a coroutine instance tracks if the instance is live (i.e. non-terminated), if a nested call is in progress, what the last yield value was, the result value (if the instance terminated), and the exception that was thrown (if any). The coroutine instance exposes the `value`, `result` and `exception` user-facing methods, which either return the value of the respective field, or throw an error if the field is not set. The instance also exposes the `resume` method, which is described shortly.

■ **Listing 25** Coroutine instance class.

```

1 class Instance[Y, R] {
2   var _live = true
3   var _call = false
4   var _value: Y = null
5   var _result: R = null
6   var _exception:
7     Exception = null
8   /* Call stack arrays */
9 }

```

Coroutine class. For each `Coroutine` declaration in the program, the transformation macro generates a new anonymous subclass of the `Coroutine` class shown in Listing 26. Each concrete `Coroutine` subclass defines several entry point methods, and implements the `_enter` method of the `Coroutine` base class. An *entry point method* is a replica of the `Coroutine` block such that it starts from either:

- (1) the beginning of the method, or
- (2) a `yieldval` statement, or
- (3) a call to another coroutine.

The `_enter` method is called when a coroutine instance resumes. It reads the current position from the coroutine instance, and dispatches to the proper entry point method with a `switch` statement. A `goto` primitive is unavailable in Scala, so the `_enter` method emulates the `goto` semantics.

■ **Listing 26** Coroutine definition base classes.

```

1 class Coroutine[Y, R] {
2   def _enter(i: Instance[Y, R]): Unit

```

```

3 }
4 class Coroutine1[T0, Y, R]
5 extends Coroutine[Y, R] {
6   def _call(
7     i: Instance[Y, R], a0: T0): Unit
8 }
9 /* One class for each arity */

```

Listing 26 also shows the abstract `Coroutine1` subclass. The `Coroutine1` subclass declares the `_call` method which stores the resume-site or callsite arguments into the proper locations in the call stack. This method is invoked by `start` and by coroutine calls. Neither JVM nor Scala support variadic templates, so we include 4 different arity classes (the same approach is used for `Function` classes in Scala, and functional interface classes in Java).

Trampolining. An entry point method ends at a position at which the original `coroutine` block has a `yieldval`, a coroutine call, or a return. Therefore, each entry point method in the `Coroutine` class is tail-recursive. Consequently, nested coroutine calls can be invoked from a trampoline-style loop.

The `resume` method in Listing 27 implements a trampoline in lines 8-10. After resetting the yield-related fields, `resume` repetitively reads the topmost coroutine from the coroutine stack `_cstack`, and invokes `_enter`. If an entry point calls another coroutine, or returns from a coroutine call, the `_call` field is set to `true`. Otherwise, if the instance yields or terminates, the `_call` field is set to `false`, and the loop ends.

■ **Listing 27** The resume trampoline.

```

1 def resume[Y, R]
2   (i: Instance[Y, R]) = {
3   if (!i._live)
4     throw sys.error()
5   i._hasValue = false
6   i._value = null
7   do {
8     i._cstack(i._ctop)
9     ._enter(i)
10  } while (i._call)
11  i._live }

```

5.3 Transformation

The transformation is performed by the following `coroutine` macro, which takes a Scala AST, typed `c.Tree`. The `coroutine` macro checks that the AST is statically a function type, and reports a compiler error otherwise. The macro returns an AST that holds a definition of an anonymous `CoroutineN` subclass (for a specific arity `N`), and a new instance of that class.

```

1 def coroutine[T, R](f: Any): Any = macro coroutine_impl[T, R]
2 def coroutine_impl[T, R](c: Context)(f: c.Tree): c.Tree = ...

```

The transformation consists of four compilation phases. First, the input AST is converted into a normal form. Second, the normalized AST is converted into a control flow graph. Third, the control flow graph is cut into segments at the points where the coroutine yields or calls other coroutines. Finally, control flow graph segments are converted back to ASTs, which represent the coroutine's entry points and are used to generate the anonymous class.

AST normalization. This phase converts the input AST with arbitrary phrases into a normalized AST. The phrases in the normalized AST are restricted to:

- (1) single operator expressions on constants and identifiers,
- (2) assignments and declarations whose right-hand side is a constant or an identifier³,
- (3) method calls on constants and identifiers,
- (4) if-statements and while-loops whose condition is a constant or an identifier and whose body is normalized,
- (5) basic blocks whose statements are normalized.

The benefit of normalization is that the subsequent phases have fewer cases to consider.

■ **Listing 28** Canonicalized list coroutine.

```

1 val bucket = coroutine {
2   (b: List[Int]) =>
3   var x_0 = b != Nil
4   var x_1 = x_0
5   while (x_1) {
6     var x_2 = b.head
7     var x_3 = yieldval(x_2)
8     var x_4 = b.tail
9     b = x_4
10    var x_0 = b != Nil
11    x_1 = x_0 }
12   () }

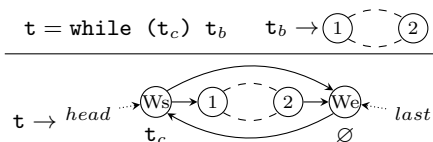
```

Example. Recall the `bucket` coroutine from Listing 5, which yields the elements of a list. After normalization, this coroutine is transformed into the coroutine in Listing 28.

$$\text{var } x=t \rightarrow \text{head} \cdots \text{var } x=t \leftarrow \cdots \text{last}$$

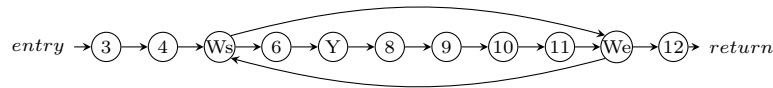
Control flow graph conversion. A normalized AST is converted to a control flow graph. The transformation is implemented as a set of mappings between the input AST and the output CFG nodes. An example rule for a variable declaration is informally shown above, where a declaration is replaced by a single node that records the AST.

The rule for `while`-loops, informally shown on the right, relies on the recursive transformation of the body t_b of the loop. Given that t_b transforms to a CFG that starts with a node 1 and ends with a node 2, a `while`-loop transforms to a pair of W_s and W_e nodes, which are connected with successor links (solid lines) as shown in the figure.



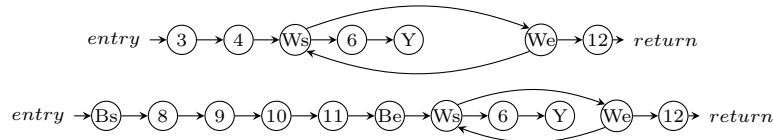
Example. Recall once more the `bucket` coroutine from Listing 5. The resulting control flow graph is shown below. Nodes that do not represent structured control flow or yielding are annotated with the line number from Listing 28, and the Y node represents the yield-site.

³ We sometimes slightly deviate from this in the examples, for better readability.



Control flow graph splitting Resuming a coroutine effectively jumps to the middle of its body. Such a jump is not possible if the target language that supports only structured programming constructs. As explained in Section 5.2, the transformation outputs multiple entry point subroutines, each containing only structured control flow. Therefore, the CFG from the previous phase is split into multiple segments, each corresponding to an entry point.

The splitting starts at the node that corresponds to the coroutine method entry, and traverses the nodes of the graph until reaching a previously unseen yield-site or coroutine call. The search is repeated from each split, taking care not to repeat the search twice. If the graph traversal encounters a control flow node (such as *We*) whose corresponding *Ws* node was not seen as part of the same segment (which can happen if there is a yield inside a while-loop), such a node is converted into a *Be* (block exit) node, followed by a loop again.



Example. The control flow graph of the `bucket` coroutine from Listing 5, produced in the previous phase, is split into the following pair of segments. Note that the second segment starts from the yield-site inside the loop, and that one loop iteration is effectively unrolled.

■ **Listing 29** Entry points of `bucket`.

```

1 def _enter(
2   i: Instance[Int, Unit]
3 ): Unit =
4   i._pstack(i._ptop) match {
5     case 0 => _ep0(i)
6     case 1 => _ep1(i)
7   }
8
9 def _ep0(
10  i: Instance[Int, Unit]
11 ): Unit = {
12   var b = i._rstack(i._rtop + 0)
13   var x_0 = b != Nil
14   var x_1 = x_0
15   while (x_1) {
16     var x_2 = b.head
17     i.value = x_2
18     i._pstack(i._ptop + 0) = 1
19     return
20   }
21   i._result = ()
22   i._cstack(i._ctop) = null
23   i._ctop -= 1
24   i._ptop -= 1 }

```

■ **Listing 30** Entry points of `bucket`, cont.

```

1 def _ep1(
2   i: Instance[Int, Unit]
3 ): Unit = {
4   var b = i._rstack(i._rtop + 0)
5   var x_1 = false
6   {
7     var x_3 = ()
8     var x_4 = b.tail
9     b = x_4
10    var x_0 = b != Nil
11    x_1 = x_0
12  }
13
14  while (x_1) {
15    var x_2 = b.head
16    i.value = x_2
17    i._rstack(i._rtop + 0) = b
18    i._pstack(i._ptop + 0) = 1
19    return
20  }
21  i._result = ()
22  i._cstack(i._ctop) = null
23  i._ctop -= 1
24  i._ptop -= 1 }

```

AST generation. This phase transforms the CFG segments back into the ASTs for the entry point methods. Each entry point starts by restoring the local variables from the call stack. At each yield-site and each coroutine call (commonly, the `exit`), the entry point method stores the local variables back to the stack. It then either stores the yield value into the

coroutine, or stores the result value. The entry point methods are placed into a `Coroutine` subclass, and the `_enter` method dispatches to the proper entry points.

Example. Listings 29 and 30 show the implementation of the entry points of the `bucket` coroutine. Note that each method starts by restoring the local variable `b` from the reference stack of the coroutine instance `i`. Each entry point ends by either storing the yield value into the `value` field of the coroutine instance, or the result value into the `result` field. Local variables are then stored onto the stack (there are two stacks – `_vstack` for primitive values and `_rstack` for references), and the program counter is stored to the `_pstack`.

■ **Listing 31** Error-handling coroutines.

```

1 val fail =
2   coroutine { (e: Error) =>
3     throw e
4   }
5 val forward =
6   coroutine { () =>
7     fail(new Error)
8   }
9 val main =
10  coroutine { () =>
11    try forward()
12    catch {
13      case e: Error =>
14        println("Failed.")
15    }
16  }
```

■ **Listing 32** Normalized `main` coroutine.

```

1 /* main */
2 var x_0: Exception = null
3 try forward()
4 catch {
5   case e => x_0 = e
6 }
7 var x_1 = x_0 != null
8 if (x_1) {
9   var x_2 =
10    x_0.isInstanceOf[Error]
11    if (x_2) {
12      println("Failed.")
13    } else {
14      throw x_0
15    }
16 }
```

5.4 Exception handling

Code inside the `coroutine` block can throw an exception. In this case, standard exception handling semantics apply – the control flow must continue from the nearest dynamically enclosing `catch` block that handles the respective exception type. To ensure this, the transformation does three things:

- (1) it normalizes the `try-catch` and `throw` ASTs,
- (2) it treats each `throw` statement as a suspension point that writes to the instance's `_exception` field,
- (3) it places an exception handler at the beginning of each entry point.

To explain these steps, we use the example in Listing 31. The `fail` coroutine takes an `Error` argument and throws it. The `forward` coroutine creates an `Error` object, and calls the `fail` coroutine without handling its exceptions. The `main` coroutine calls the `forward` coroutine inside a `try` block, and catches the subset of exceptions with the `Error` type.

Normalization. The normalized coroutine `main` is shown in Listing 32. The `catch` handler is transformed so that, once caught, the exception is immediately stored into `x_0`. The variable `x_0` is matched against the `Error` type in the subsequent `if`-statements.

Exception throws. The transformation of the `throw` statement from the `fail` coroutine is in Listing 33. The parameter is loaded into the variable `e`, and immediately stored into the `exception` field. The coroutine stack `_cstack` is then popped, and the coroutine returns.

■ **Listing 33** 1st entry point of fail.

```

1 /* fail, _ep0 */
2 var e = i._rstack(i._rtop)
3 i._exception = e
4 i._rstack(i._rtop) = null
5 i._rtop -= 1
6 return

```

■ **Listing 34** 2nd entry point of main.

```

1 /* main, _ep1 */
2 try {
3     try {
4         var x_0 = i._exception
5         if (x_0 != null) throw x_0
6     } catch { case e => x_0 = e }
7     var x_1 = x_0 != null
8     if (x_1) {
9         var x_2 =
10            x_0.isInstanceOf[Error]
11            if (x_2) println("Failed.")
12            else throw x_0
13     }
14     /* Normal exit */
15 } catch { case x_1 =>
16     /* Exceptional exit */
17 }

```

■ **Listing 35** Entry points of forward.

```

1 /* forward, _ep0 */
2 var x_0 = new Error
3 fail._call(i, x_0)
4 i._ctop += 1
5 i._cstack(i._ctop) = null
6 i._ptop += 1
7 i._pstack(i._ptop) = 0
8 i._call = true
9 return
10
11 /* forward, _ep1 */
12 try {
13     var x_0 = i._exception
14     if (x_0 != null) throw x_0
15     i._cstack(i._ctop) = null
16     i._ctop -= 1
17     i._ptop -= 1
18     return
19 } catch { case x_1 =>
20     i._exception = x_1
21     i._cstack(i._ctop) = null
22     i._ctop -= 1
23     i._ptop -= 1
24     return
25 }
26

```

Stack unwinding. The final rule is to wrap every entry point that starts at a return from a coroutine call into an *unwinding exception handler*. In addition, if the previous entry point ended inside a *user exception handler*, then a replica of that handler is added.

The *forward* coroutine does not have an exception handler, so its second entry point `_ep1` contains only the *unwinding* handler, as shown in Listing 35. On the other hand, *main*'s first entry point `_ep0` ends with a coroutine call. Listing 34 shows that the second entry point `_ep1` therefore has both the *unwinding* handler and the *user* handler. If the user handler cannot handle the exception, then the exception is rethrown.

5.5 Optimizations

An entry point method does not need to load all the local variables at the beginning, nor store all of them to the stack. For example, the entry points of the *bucket* coroutine in Listings 29 and 30 only store a subset of all the variables in the scope. In particular, `_ep0` does not store the variables `b`, `x_0`, `x_1` and `x_2`, while `_ep1` only stores `b`. In this section, we explain the optimization rules used to avoid the unnecessary loads and stores.

Scope rule. A variable does not need to be loaded or stored if it is not in scope after the exit point. This rule applies to, for example, the variables `x_3` and `x_4` from Listing 30.

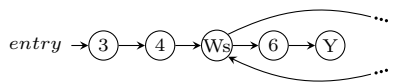
► **Definition 13.** A control flow graph node d dominates a node n if every control flow path from the begin node to n must go through d .

Must-load rule. A local variable v must be loaded from the stack if the respective entry point contains at least one read of v that is not dominated by a write of v .

Example. Consider the variable `b` of the entry point `entry` \rightarrow $\textcircled{\text{Bs}}$ \rightarrow $\textcircled{8}$ \rightarrow $\textcircled{9}$ \rightarrow $\textcircled{10}$ \rightarrow \dots `_ep1` of the `bucket` coroutine in Listing 30. In the corresponding control flow graph, the read in the node 8 precedes the write in the node 9. Consequently, there exists a read that is not dominated by any write, and `b` must be loaded.

► **Definition 14.** A control flow path is a connected sequence of CFG nodes. A control flow path is *v-live* if the variable `v` is in scope in all the nodes of that control flow path.

Was-changed rule. A local variable `v` must not be stored to the stack if there is no *v-live* control flow path that starts with a write to `v` and ends at the respective exit node.



Example. Consider the variable `b` of the entry point `_ep0` of the `bucket` coroutine in Listing 29. In the corresponding CFG, the variable `b` is read in nodes 3 and 6. However, there is no write to `v` that connects to the exit node `Y`. Therefore, at `Y`, `b` did not change its value since the begin node, so it does not have to be stored.

► **Definition 15.** We say that an exit node `x` *resumes* at an entry point `e`, if the exit node corresponds to the begin node of `e` in the original control flow graph.

► **Definition 16.** Relation $needed(x, v, e)$ between an exit node `x`, the variable `v` and an entry point method `e` holds if and only if either:

- (1) `x` resumes at `e`, and the must-load rule applies to `v` and `e`, or
- (2) `x` resumes at `e'`, and there is a *v-live* control flow path between the begin node of `e'` and some exit node `x'` of `e'`, such that $needed(x', v, e)$.

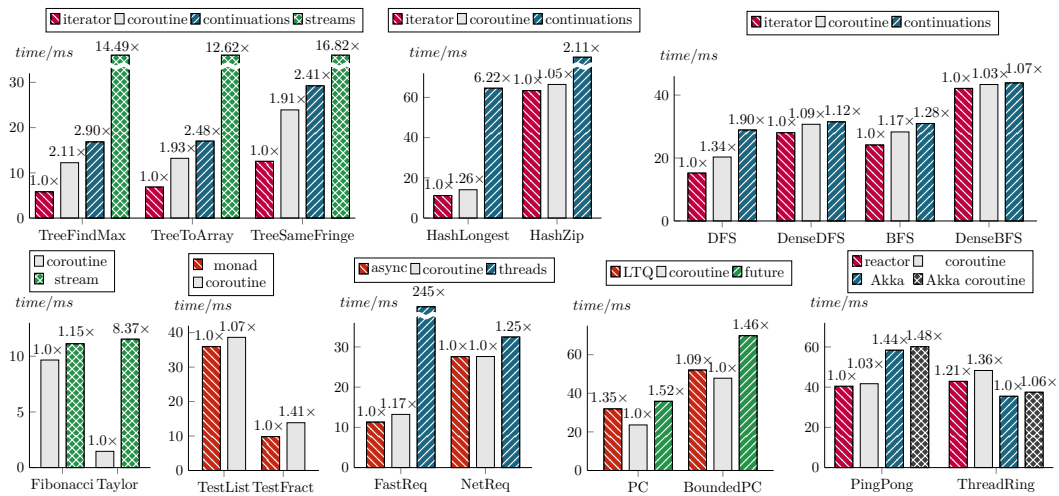
Is-needed rule. A local variable `v` must not be stored to the stack at an exit node `x` if there is no entry point method `e` such that $needed(x, v, e)$.

Example. Intuitively, this rule applies when it becomes impossible to reach (without `v` going out of scope) an entry point that would need to load `v`. This rule applies to the variable `x_2` in the entry point `_ep1` in Listing 30. Variable `x_2` is in scope at the exit point, however, it does not need to be loaded when `_ep1` is reentered, and it goes out of scope before it is needed again.

6 Performance Evaluation

The goal of the evaluation is to assess coroutine performance on a range of different use cases, most notably those from Section 3. The source code of the benchmarks is available online [44]. Evaluation was done in accordance with established JVM benchmarking methodologies [19]. We used the ScalaMeter framework [42] to repeat each benchmark 30 times, across 6 different JVM process instances, and we report the average values. We used a quad-core 2.8 GHz Intel i7-4900MQ processor with 32 GB of RAM, and results are shown in Figure 4.

Iterators. We test tree iterators from Listing 8 against a manually implemented tree iterator. The first benchmark, `TreeFindMax`, traverses a single tree and finds the largest integer, while `TreeToArray` copies the integers to an array. `TreeSameFringe` compares the corresponding integers in two trees with a different layout [18]. These benchmarks heavily modify the stack, so a coroutine is 1.9 – 2.1× slower than an iterator. A CPS-based iterator, built using Scala



■ **Figure 4** Performance of coroutines and alternative frameworks (lower is better).

delimited continuations [56], is 2.4 – 2.9 \times slower. For comparison, a lazy functional stream is 12 – 17 \times slower.

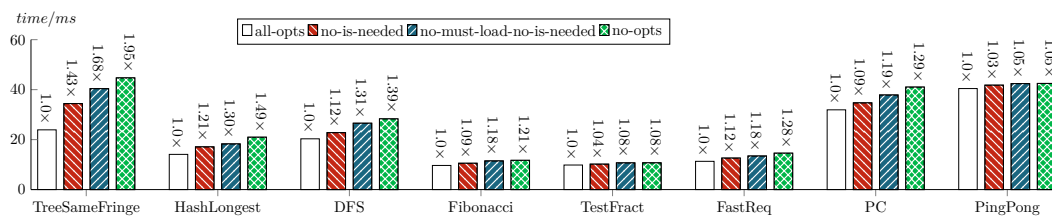
The *HashLongest* benchmark traverses a hash table to find the longest string (strings contain size information, so checks are cheap). Since most of the time is spent in the loop and not in coroutine calls, performance overhead compared to iterators is only 26%. This benchmark reveals a downside of CPS-based continuations⁴. The relative overhead of allocating continuation closures is considerable for hash table iterators, so a continuation-based iterator is 6.22 \times slower. *HashZip* simultaneously traverses two hash tables, picks an element from each pair, and inserts it into a third hash table, effectively implementing a zip operation on the hash tables. Since zipping does additional work of implementing the resulting hash table, coroutines have only a 5% overhead compared to a manually implemented iterator, while continuations are 2.1 \times slower.

DFS and *BFS* benchmarks traverse a sparse graph (degree 3) in depth-first and breadth-first order. Coroutines have an overhead of 34% and 17% compared to manually implemented iterators, and continuations 90% and 28%, respectively. Making the graphs denser (degree 16) amortizes the overhead of suspensions, resulting in overheads of 9% and 3% for coroutines.

Lazy streams. Functional lazy streams, or lazy lists, are a neat abstraction for recursively defining number series. Coroutines are also a good fit for this use-case, but are considerably faster, since a stream needs to create a node object for each number in the series.

The *Fibonacci* benchmarks generates Fibonacci numbers, and uses big integer arithmetic to do so. The overhead of a lazy-stream-based solution compared to a coroutine-based one is only 15%. The *Taylor* benchmark generates a Taylor series, using floating-point arithmetic. The work involved in a floating-point computation is much lower compared to big-integer arithmetic. Since the relative overhead of lazy streams is much more pronounced in this case, the stream-based solution is 8.4 \times slower.

⁴ In some cases, the Scala compiler can eliminate tail-calls, but typically not when invoking lambda objects that encode the continuations produced by Scala’s delimited continuations plugin.



■ **Figure 5** Impact of optimizations on performance (lower is better).

ScalaCheck. The *TestList* and *TestFract* benchmarks compare regular ScalaCheck generator-based testing [36] with backtracking from Section 3. The *TestList* benchmark checks properties of list objects, and is computation-heavy – relative backtracking overhead due to creating snapshots is only 7%. The *TestFract* benchmark checks properties of fractions and does only simple arithmetic – in this case, backtracking overhead is 41%.

Async-Await. Here, network communication is the primary use-case. *FastReq* creates an immediately completed request, and awaits it. In this case, coroutine-based implementation from Section 3 has a 17% overhead. Just for comparison, starting a new thread for each request is 245× slower. In practice, the network introduces a delay between requests and responses. *NetReq* uses a 1 ms delay, in which case coroutines have no observable overhead.

Single-assignment variables. In this benchmark, Oz-style single-assignment variables from Listing 18 are used to implement dataflow streams – a variant of cons-lists with single-assignment tails. This allows a straightforward encoding of the producer-consumer pattern [65]. The *PC* benchmark compares dataflow streams based on Scala Futures [22] with coroutine-based streams. The direct-style coroutine API has an interesting performance impact. Futures are 52% slower because every tail-read must allocate a closure and install a callback even if the value is already present, whereas a coroutine can be directly resumed when a value is available. The *LinkedTransferQueue* from the JDK blocks the thread when waiting for a value, and is 35% slower. *Bounded PC* adds an additional backpressure dataflow stream between the producer and the consumer, and has similar performance ratios.

Actors and event streams. We compare callback-style and direct-style Akka actors [1] and reactors [54, 47, 43, 45] on two benchmarks from the Savina actor benchmark suite [24]. Direct-style programs are encoded by hot-swapping the event loop, as explained in Section 3. The callback allocation in *receive* and *get* calls causes a 3% slowdown for reactors and 2.8% for actors in *PingPong*. In *ThreadRing*, slowdown is 12% for reactors and 6% for actors.

Optimization Breakdown. We show a breakdown of different optimizations from Section 5.5. We pick eight benchmarks from Figure 4 and run them after disabling different optimization combinations. We observe the highest impact on *TreeSameFringe*, *HashLongest*, *DFS* and *PC*. In Figure 5, *all-opts* shows performance with all optimizations enabled, *no-is-needed* disables the is-needed rule, *no-must-load-no-is-needed* additionally disables the must-load rule, and *no-opts* disables all optimizations. Results show that optimizations have the highest impact on *TreeSameFringe*, where disabling them causes a total slowdown of almost 2×. Here, 50% of the performance comes from the is-needed rule. In other benchmarks shown in Figure 5, total improvement from optimizations ranges from 5% to 50%.

■ **Table 1** Comparison of Suspension Primitives in Different Languages.

Name	Type-safe	First-class	Stackful	Allocation-free	Scope	Snapshots
Enumerators (C#)	✓	✗	✗	∅	delimited	✗
Generators (Python)	✗	✓	✓	✗	delimited	✗
Async (Scala, C#)	✓	✓	✓	✗	delimited	✗
Spawn-sync (Cilk)	✓	✗	✓	✓	whole program	✗
Boost (C++)	✓	✓	✓	✓	delimited	✗
CO2 (C++)	✓	✓	✗	∅	delimited	✗
Coroutines (Lua)	✗	✓	✓	✓	just-in-time	✗
Coroutines (Kotlin)	✓	✓	✓	✗	delimited	✗
Coroutines (Scala)	✓	✓	✓	✓	delimited	✓

7 Related Work

We organize the related work on coroutines into several categories. We start with the origins and previous formalization approaches, we then contrast coroutines to similar domain-specific primitives, and conclude with the related work on continuations. Where appropriate, we contrast our model with alternatives. As stated in the introduction, many features of our model have been studied already. However, our main novelty is that our delimited coroutines rely only on metaprogramming, as well as augmenting coroutines with snapshots.

Coroutine in programming languages. Table 1 is a brief comparative summary of suspension primitives in different languages. We compare type-safety, whether suspendable code blocks are first-class objects, whether coroutines are stackful, and if suspendable blocks can call each other without dynamic allocation. The scope column denotes the scope in which the primitive can be used.

The Kotlin language exposes coroutines with its `suspend` and `yield` keywords [7]. Kotlin’s implementation is delimited and CPS-based, and it translates every call to a coroutine `c` to an allocation of a coroutine class specific to `c`. This coroutine-specific class holds the state of the local variables. Instances of this class are chained and form linked-list-based callstack. Our translation approach is different in that a coroutine call modifies an array-based call stack, and does not require an object allocation. Currently, Kotlin coroutines do not allow snapshots, which makes them equivalent to one-shot continuations [34].

In the C++ community, there are two popular coroutine libraries: Boost coroutines [27] and CO2 [25]. Boost coroutines are stackful, and they expose two separate asymmetric coroutine types: push-based coroutines, where resuming takes an input value, and pull-based coroutines, where resuming returns an output value. They do not support snapshots due to problems with memory safety in copying the stack. CO2 aims to implement fast coroutines, and reports better performance than Boost, but it supports only stackless coroutines.

Origins and formalizations. The idea of coroutines dates back to Erdwinn and Conway’s work on a tape-based Cobol compiler and its separability into modules [11]. Although the original use-case is no longer relevant, other use-cases emerged. Coroutines were investigated on numerous occasions, and initially appeared in languages such as Modula-2 [68], Simula [6], and BCPL [33]. A detailed classification of coroutines is given by Moura and Ierusalimschy [34], along with a formalization of asymmetric coroutines through an operational semantics. Moura and Ierusalimschy observed that asymmetric first-class stackful coroutines have an equal expressive power as one-shot continuations, but did not investigate snapshots, which

make coroutines equivalent to full continuations. Anton and Thiemann showed that it is possible to automatically derive type systems for symmetric and asymmetric coroutines by converting their reduction semantics into equivalent functional implementations, and then applying existing type systems for programs with continuations [2]. James and Sabry identified the input and output types of coroutines [57], where the output type corresponds to the *yield* type described in this paper. The input type ascribes the value passed to the coroutine when it is resumed. As a design tradeoff, we chose not to have explicit input values in our model. First, the input type increases the verbosity of the coroutine type, which may have practical consequences. Second, as shown in examples from Section 3, the input type can be simulated with the return type of another coroutine, which yields a writable location, and returns its value when resumed (e.g. the `await` coroutine from Section 3). Fischer et al. proposed a coroutine-based programming model for the Java programming language, along with the respective formal extension of Featherweight Java [17].

Domain-specific approaches. The need for simpler control flow prompted the introduction of coroutine-inspired primitives that target specific domains. One of the early applications was data structure traversal. Push-style traversal with `foreach` is easy, but the caller must relinquish control, and many applications cannot do this (e.g. the same-fringe benchmark from Section 6). Java-style iterators with `next` and `hasNext` are harder to implement than a `foreach` method, and coroutines bridge this gap.

Iterators in CLU [29] are essentially coroutines – program sections with `yield` statements that are converted into traversal objects. C# inherited this approach – its iterator type `IEnumerator` exposes `Current` and `MoveNext` methods. Since enumerator methods are not first class entities, it is somewhat harder to abstract suspendable code, as in the backtracking example from Section 3. C# enumerators are not stackful, so the closed addressing hash table example from the Listing 6 must be implemented inside a single method. Enumerators can be used for asynchronous programming, but they require exposing `yield` in user code. Therefore, separately from enumerators, C# exposes `async-await` primitives. Some newer languages such as Dart similarly expose an `async-await` pair of primitives.

Async-Await in Scala [23] is implemented using Scala’s metaprogramming facilities. Async-await programs can compose by expressing asynchronous components as first-class `Future` objects. The Async-Await model does not need to be stackful, since separate modules can be expressed as separate futures. However, reliance on futures and concurrency makes it hard to use Async-Await generically. For example, iterators implemented using futures have considerable performance overheads due to synchronization involved in creating future values.

There exist other domain-specific suspension models. For example, Erlang’s `receive` statement effectively captures the program continuation when awaiting for the inbound message [66]. A model similar to Scala Async was devised to generate Rx’s `Observable` values [20, 32], and the event stream composition in the reactor model [46, 49], as well as callbacks usages in asynchronous programming models based on futures and flow-pools [22, 53, 52, 41, 59] can be similarly simplified. Cilk’s `spawn-sync` model [28] is similar to `async-await`, and it is implemented as a full program transformation. The Esterel language defines a `pause` statement that pauses the execution, and continues it in the next event propagation cycle [5]. Behaviour trees [31] are AI algorithms used to simulate agents – they essentially behave as AST interpreters with `yield` statements.

Generators. Dynamic languages often support *generators*, which are essentially untyped asymmetric coroutines. A Python generator instance [58] exposes only the `next` method (an

equivalent of `resume`), which throws a `StopIteration` error when it completes. In practice, Python generators are mostly used for *list comprehensions*, as programmers find it verbose to handle `StopIteration` errors. Newer Python versions allow stackful generators [15] with the `yield from` statement, which is implemented as syntactic sugar around basic generators that chains the `resume` points instead of using call stacks. ECMAScript 6 generators are similar to Python's generators. Lua coroutines bear the most similarity with the coroutine formulation in this paper [13], with several differences. First, Lua coroutines are not statically typed. While this is less safe, it has the advantage of reduced syntactic burden. Second, Lua coroutines are created from function values dynamically. This is convenient, but requires additional JIT optimizations to be efficient.

Transformation-based continuations. Continuations are closely related to coroutines, and with the addition of `snapshot` the two can express the same programs. Scheme supports programming with continuations via the `call/cc` operator, which has a similar role as `shift` in shift-reset delimited continuations [12, 3]. In several different contexts, it was shown that continuations subsume other control constructs such as exception handling, backtracking, and coroutines. Nonetheless, most programming languages do not support continuations today. It is somewhat difficult to provide an efficient implementation of continuations, since the captured continuations must be callable more than once. One approach to continuations is to transform the program to continuation-passing style [64]. Scala's continuations [56] implement delimited shift-reset continuations with a CPS transform. One downside of the continuation-passing style transformation is the risk of stack overflows when the tail-call optimization is absent from the runtime, as is the case of JVM.

Optimizing compilers tend to be tailored to the workloads that appear in practice. For example, it was shown that optimizations such as inlining, escape analysis, loop unrolling and devirtualization make most collection programs run nearly optimally [50, 40, 55, 63, 48]. However, abstraction overheads associated with coroutines are somewhat new, and are not addressed by most compilers. For this reason, compile-time transformations of coroutine-heavy workloads typically produce slower programs compared to their runtime-based counterparts. We postulate that targeted high-level JIT optimizations could significantly narrow this gap.

Runtime-based continuations. There were several attempts to provide runtime continuation support for the JVM, ranging from Ovm implementations [14] based on `call/cc`, to JVM extensions [62], based on the `capture` and `resume` primitives. While runtime continuations are not delimited and can be made very efficient, maintenance pressure and portability requirements prevented these implementations from becoming a part of official JVM releases. An alternative, less demanding approach relies only on stack introspection facilities of the host runtime [38]. There exists a program transformation that relies on exception-handling to capture the stack [60]. Here, before calling the continuation, the saved state is used in method calls to rebuild the stack. This works well for continuations, where the stack must be copied anyway, but may be too costly for coroutine `resume`. Bruggeman et al. observed that many use cases call the continuation only once and can avoid the copying overhead, which lead to *one-shot continuations* [8]. One-shot continuations are akin to coroutines without snapshots.

Other related constructs. Coroutines are sometimes confused with *goroutines*, which are lightweight threads in the Go language. While coroutines can be used to encode goroutines, the converse encoding is not as efficient, as goroutines involve message passing.

8 Conclusion

We described a programming model for first-class typed stackful coroutines with snapshots, along with a formalization. Our implementation relies on metaprogramming facilities of the host language. We identified the critical optimizations that need to accompany the implementation, and showed their performance impact. We identified a range of use cases such as iterators, Async-Await, Oz-style dataflow variables, Erlang-style actors, backtracking, and direct-style event streams, and we showed that they can be expressed in our model. Experimental evaluation shows that our coroutine implementation is almost as efficient as these other primitives, and in some cases has an even better performance.

Our implementation is available online [44], as an independent module that relies on metaprogramming capabilities in Scala, and works with the official language releases. This work may indicate a wider need for metaprogramming support in general purpose languages, which may be easier to provide than continuation support in the runtime. Moreover, runtime support and JIT optimizations [50] could improve the performance of our implementation even further, and we plan to investigate this in the future.

References

- 1 Akka. Akka documentation, 2011. <http://akka.io/docs/>.
- 2 Konrad Anton and Peter Thiemann. *Towards Deriving Type Systems and Implementations for Coroutines*, pages 63–79. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010. doi:10.1007/978-3-642-17164-2_6.
- 3 Kenichi Asai and Chihiro Uehara. Selective cps transformation for shift and reset. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, PEPM '18, pages 40–52, New York, NY, USA, 2018. ACM. doi:10.1145/3162069.
- 4 V. Beltran, D. Carrera, J. Torres, and E. Ayguade. Evaluating the scalability of java event-driven web servers. In *International Conference on Parallel Processing, 2004. ICPP 2004.*, pages 134–142 vol.1, Aug 2004. doi:10.1109/ICPP.2004.1327913.
- 5 Gérard Berry and Georges Gonthier. The ESTEREL Synchronous Programming Language: Design, Semantics, Implementation. *Sci. Comput. Program.*, 19(2):87–152, 1992. doi:10.1016/0167-6423(92)90005-V.
- 6 G.M. Birtwhistle, O.J. Dahl, B. Myhrhaug, and K. Nygaard. *Simula Begin*. Chartwell-Bratt Ltd, 1979.
- 7 Andrey Breslav. Coroutines for kotlin (revision 3.2), 2017. <https://github.com/Kotlin/kotlin-coroutines/blob/master/kotlin-coroutines-informal.md>.
- 8 Carl Bruggeman, Oscar Waddell, and R. Kent Dybvig. Representing control in the presence of one-shot continuations. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*, PLDI '96, pages 99–107, New York, NY, USA, 1996. ACM. doi:10.1145/231379.231395.
- 9 Eugene Burmako. Scala Macros: Let Our Powers Combine!: On How Rich Syntax and Static Types Work with Metaprogramming. In *Proceedings of the 4th Workshop on Scala*, SCALA '13, pages 3:1–3:10, New York, NY, USA, 2013. ACM. doi:10.1145/2489837.2489840.
- 10 Mike Cantelon, Marc Harter, TJ Holowaychuk, and Nathan Rajlich. *Node.js in Action*. Manning Publications Co., Greenwich, CT, USA, 1st edition, 2013.
- 11 Melvin E. Conway. Design of a Separable Transition-Diagram Compiler. *Commun. ACM*, 6(7):396–408, 1963. doi:10.1145/366663.366704.

- 12 Olivier Danvy and Andrzej Filinski. Abstracting Control. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, LFP '90, pages 151–160, New York, NY, USA, 1990. ACM. doi:10.1145/91556.91622.
- 13 A. Lúcia de Moura, N. Rodriguez, and R. Ierusalimsky. Coroutines in Lua. *Journal of Universal Computer Science*, 10(7):910–925, July 2004.
- 14 Iulian Dragos, Antonio Cuneo, and Jan Vitek. Continuations in the Java Virtual Machine. In *Second ECOOP Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS'2007)*, Berlin, 2007. Technische Universität Berlin.
- 15 Gregory Ewing. PEP 380 - Syntax for Delegating to a Subgenerator, 2009. <https://www.python.org/dev/peps/pep-0380/>.
- 16 Matthias Felleisen. The theory and practice of first-class prompts. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pages 180–190, New York, NY, USA, 1988. ACM. doi:10.1145/73560.73576.
- 17 Jeffrey Fischer, Rupak Majumdar, and Todd Millstein. Tasks: Language support for event-driven programming. In *Proceedings of the 2007 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, PEPM '07, pages 134–143, New York, NY, USA, 2007. ACM. doi:10.1145/1244381.1244403.
- 18 Richard P. Gabriel. The Design of Parallel Programming Languages. In Vladimir Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation*, pages 91–108. Academic Press Professional, Inc., San Diego, CA, USA, 1991. URL: <http://dl.acm.org/citation.cfm?id=132218.132225>.
- 19 Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous java performance evaluation. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, pages 57–76, New York, NY, USA, 2007. ACM. doi:10.1145/1297027.1297033.
- 20 Philipp Haller and Heather Miller. RAY: Integrating Rx and Async for Direct-Style Reactive Streams. In *Workshop on Reactivity, Events and Modularity*, 2013.
- 21 Philipp Haller and Martin Odersky. Scala actors: Unifying thread-based and event-based programming. *Theor. Comput. Sci.*, 410(2-3):202–220, feb 2009. doi:10.1016/j.tcs.2008.09.019.
- 22 Philipp Haller, Aleksandar Prokopec, Heather Miller, Viktor Klang, Roland Kuhn, and Vojin Jovanovic. Scala improvement proposal: Futures and promises. In *SIP-14*, 2012. URL: <http://docs.scala-lang.org/sips/pending/futures-promises.html>.
- 23 Philipp Haller and Jason Zaugg. Scala Async Repository, 2013. <https://github.com/scala/async>.
- 24 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. ACM. doi:10.1145/2687357.2687368.
- 25 Jamboree. Co2: A c++ await/yield emulation library for stackless coroutine, 2017. URL: <https://github.com/jamboree/co2>.
- 26 Ralph E. Johnson and Brian Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, June/July 1988. URL: <http://www.laputan.org/drc.html>.
- 27 Oliver Kowalke. Coroutine2, 2017. URL: http://www.boost.org/doc/libs/1_66_0/libs/coroutine2.
- 28 Charles E. Leiserson. *Programming irregular parallel applications in Cilk*, pages 61–71. Springer Berlin Heidelberg, Berlin, Heidelberg, 1997. doi:10.1007/3-540-63138-0_6.
- 29 Barbara Liskov, Alan Snyder, Russell Atkinson, and Craig Schaffert. Abstraction Mechanisms in CLU. *Commun. ACM*, 20(8):564–576, aug 1977. doi:10.1145/359763.359789.

- 30 Ingo Maier and Martin Odersky. Deprecating the Observer Pattern with Scala.react. Technical report, EPFL, 2012.
- 31 A. Marzotto, M. Colledanchise, C. Smith, and P. Ögren. Towards a Unified Behavior Trees Framework for Robot Control. In *2014 IEEE International Conference on Robotics and Automation (ICRA)*, pages 5420–5427, May 2014. doi:10.1109/ICRA.2014.6907656.
- 32 Erik Meijer. Your Mouse is a Database. *Commun. ACM*, 55(5):66–73, may 2012. doi:10.1145/2160718.2160735.
- 33 Ken Moody and Martin Richards. A coroutine mechanism for bcpl. *Softw., Pract. Exper.*, 10(10):765–771, 1980. doi:10.1002/spe.4380101002.
- 34 Ana Lúcia De Moura and Roberto Ierusalimsky. Revisiting Coroutines. *ACM Trans. Program. Lang. Syst.*, 31(2):6:1–6:31, 2009. doi:10.1145/1462166.1462167.
- 35 Lasse R. Nielsen and BRICS. A selective cps transformation. *Electronic Notes in Theoretical Computer Science*, 45:311–331, 2001. MFPS 2001, Seventeenth Conference on the Mathematical Foundations of Programming Semantics. doi:10.1016/S1571-0661(04)80969-1.
- 36 Rickard Nilsson. ScalaCheck Website, 2010. <https://www.scalacheck.org/>.
- 37 Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An Overview of the Scala Programming Language. Technical report, EPFL, 2004.
- 38 Greg Pettyjohn, John Clements, Joe Marshall, Shriram Krishnamurthi, and Matthias Felleisen. Continuations from Generalized Stack Inspection. *SIGPLAN Not.*, 40(9):216–227, sep 2005. doi:10.1145/1090189.1086393.
- 39 Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, MA, USA, 2002.
- 40 A. Prokopec, D. Petrashko, and M. Odersky. Efficient lock-free work-stealing iterators for data-parallel collections. In *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 248–252, March 2015. doi:10.1109/PDP.2015.65.
- 41 Aleksandar Prokopec. *Data Structures and Algorithms for Data-Parallel Computing in a Managed Runtime*. PhD thesis, IC, Lausanne, 2014. doi:10.5075/epfl-thesis-6264.
- 42 Aleksandar Prokopec. Scalometer website, 2014. URL: <http://scalometer.github.io>.
- 43 Aleksandar Prokopec. Pluggable scheduling for the reactor programming model. In *Proceedings of the 6th International Workshop on Programming Based on Actors, Agents, and Decentralized Control*, AGERE 2016, pages 41–50, New York, NY, USA, 2016. ACM. doi:10.1145/3001886.3001891.
- 44 Aleksandar Prokopec. Scala Coroutines Website, 2016. <https://storm-enroute/coroutines>.
- 45 Aleksandar Prokopec. Accelerating by idling: How speculative delays improve performance of message-oriented systems. In Francisco F. Rivera, Tomás F. Pena, and José C. Cabaleiro, editors, *Euro-Par 2017: Parallel Processing*, pages 177–191, Cham, 2017. Springer International Publishing.
- 46 Aleksandar Prokopec. Encoding the building blocks of communication. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2017, pages 104–118, New York, NY, USA, 2017. ACM. doi:10.1145/3133850.3133865.
- 47 Aleksandar Prokopec. Reactors.io website, 2018. URL: <http://reactors.io>.
- 48 Aleksandar Prokopec, Phil Bagwell, Tiark Rompf, and Martin Odersky. A generic parallel collection framework. In *Proceedings of the 17th international conference on Parallel processing - Volume Part II*, Euro-Par’11, pages 136–147, Berlin, Heidelberg, 2011. Springer-Verlag. URL: <http://dl.acm.org/citation.cfm?id=2033408.2033425>.
- 49 Aleksandar Prokopec, Philipp Haller, and Martin Odersky. Containers and aggregates, mutators and isolates for reactive programming. In *Proceedings of the Fifth Annual Scala*

- Workshop*, SCALA '14, pages 51–61, New York, NY, USA, 2014. ACM. doi:10.1145/2637647.2637656.
- 50 Aleksandar Prokopec, David Leopoldseder, Gilles Duboscq, and Thomas Würthinger. Making collection operations optimal with aggressive JIT compilation. In *Proceedings of the 8th ACM SIGPLAN International Symposium on Scala*, SCALA 2017, pages 29–40, New York, NY, USA, 2017. ACM. doi:10.1145/3136000.3136002.
 - 51 Aleksandar Prokopec and Fengyun Liu. On the soundness of coroutines with snapshots. *CoRR*, abs/1806.01405, 2018. arXiv:1806.01405.
 - 52 Aleksandar Prokopec, Heather Miller, Philipp Haller, Tobias Schlatter, and Martin Odersky. FlowPools: A Lock-Free Deterministic Concurrent Dataflow Abstraction, Proofs. Technical report, EPFL, 2012.
 - 53 Aleksandar Prokopec, Heather Miller, Tobias Schlatter, Philipp Haller, and Martin Odersky. Flowpools: A lock-free deterministic concurrent dataflow abstraction. In *LCPC*, pages 158–173, 2012. doi:10.1007/978-3-642-37658-0_11.
 - 54 Aleksandar Prokopec and Martin Odersky. Isolates, Channels, and Event Streams for Composable Distributed Programming. In *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, Onward! 2015, pages 171–182, New York, NY, USA, 2015. ACM. doi:10.1145/2814228.2814245.
 - 55 Aleksandar Prokopec, Dmitry Petrashko, and Martin Odersky. On lock-free work-stealing iterators for parallel data structures. Technical report, EPFL, 2014.
 - 56 Tiark Rompf, Ingo Maier, and Martin Odersky. Implementing First-Class Polymorphic Delimited Continuations by a Type-Directed Selective CPS-Transform. *SIGPLAN Not.*, 44(9):317–328, 2009. doi:10.1145/1631687.1596596.
 - 57 P. James Roshan and Amr Sabry. Yield: Mainstream delimited continuations, 2011.
 - 58 Neil Schemenauer, Tim Peters, and Magnus Hetland. PEP 255 - Simple Generators, 2001. <https://www.python.org/dev/peps/pep-0255/>.
 - 59 Tobias Schlatter, Aleksandar Prokopec, Heather Miller, Philipp Haller, and Martin Odersky. Multi-lane flowpools: A detailed look. *Tech Report*, 2012.
 - 60 Tatsuro Sekiguchi, Takahiro Sakamoto, and Akinori Yonezawa. Portable implementation of continuation operators in imperative languages by exception handling. In *Advances in Exception Handling Techniques (the Book Grows out of a ECOOP 2000 Workshop)*, pages 217–233, London, UK, UK, 2001. Springer-Verlag. URL: <http://dl.acm.org/citation.cfm?id=647332.722736>.
 - 61 Denys Shabalin, Eugene Burmako, and Martin Odersky. Quasiquotes for Scala. Technical report, EPFL, 2013.
 - 62 Lukas Stadler, Christian Wimmer, Thomas Würthinger, Hanspeter Mössenböck, and John Rose. Lazy continuations for java virtual machines. In *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, PPPJ '09, pages 143–152, New York, NY, USA, 2009. ACM. doi:10.1145/1596655.1596679.
 - 63 Arvind K. Sujeeth, Tiark Rompf, Kevin J. Brown, HyoukJoong Lee, Hassan Chafi, Victoria Popic, Michael Wu, Aleksandar Prokopec, Vojin Jovanovic, Martin Odersky, and Kunle Olukotun. Composition and reuse with compiled domain-specific languages. In *Proceedings of the 27th European Conference on Object-Oriented Programming*, ECOOP'13, pages 52–78, Berlin, Heidelberg, 2013. Springer-Verlag. doi:10.1007/978-3-642-39038-8_3.
 - 64 Gerald Jay Sussman and Guy L. Steele, Jr. Scheme: A interpreter for extended lambda calculus. *Higher Order Symbol. Comput.*, 11(4):405–439, 1998. doi:10.1023/A:1010035624696.
 - 65 Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*. The MIT Press, 1st edition, 2004.

- 66 Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in ER-LANG (2nd Ed.)*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1996.
- 67 Philip Wadler. Monads for Functional Programming. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, pages 24–52, London, UK, UK, 1995. Springer-Verlag. URL: <http://dl.acm.org/citation.cfm?id=647698.734146>.
- 68 N. Wirth. *Programming in Modula-2*. Texts and Monographs in Computer Science. Springer-Verlag, 1985. URL: <https://books.google.ch/books?id=ZVaRXPrD1AoC>.

A Additional coroutine-based implementations

In Section 3, we showed how coroutines simplify actors and Rx streams. In this section, we show the coroutine-based implementations for these use-cases.

Actors. Since JVM does not have continuations, actor frameworks are unable to implement exact Erlang-style semantics, in which the `receive` statement can be called anywhere in the actor. Instead, frameworks like Akka expose a top-level `receive` method [1], which returns a partial function used to handle messages. This function can be swapped during the actor lifetime with the `become` statement. In the example from Listing 15, we used a `receive` statement that suspends in the middle of the actor and awaits a message. We now show its implementation using coroutines.

The core idea is to implement a `recv` coroutine (we call it `recv` to disambiguate from Akka’s `receive` function), which yields a partial function that represents the continuation of the actor. The resume-site can then call `become` to hot-swap Akka’s top-level `receive` handler with the yielded partial function.

In Listing 36, we first define an auxiliary type `Rec`, which describes a partial function that can take `Any` message objects. The method `act` declares an Erlang-style actor – it takes a coroutine that may yield partial functions of the `Rec` type, which describe how to process the next message. The `act` method starts an instance of the input coroutine, and resumes it inside a recursive `loop` function. The instance potentially calls the `recv` method, which yields. When this happens, `act` extends the yielded partial function with the `andThen` combinator which recursively resumes the coroutine instance. This chained partial function is passed to `become`, which tells Akka to run the chained function when a message arrives.

The `recv` is a coroutine that yields the `Rec` function and returns a message of type `Any` – the actor definition must then match this value. The implementation of `recv` declares a local variable `res` in which the incoming message is stored by the yielded partial function. After `recv` gets resumed, `act` will have already called the yielded function, which will have written the message to `res`, so that it can be returned to the actor that invoked `recv`.

Listing 36 Erlang-style actor implementation.

```

1 type Rec = PartialFunction[Any, Unit]
2 def act(c: () ~> (Rec, Unit)) = Actor { self =>
3   val i = c.start()
4   def loop() =
5     if (i.resume) self.become(i.value.andThen(loop))
6     else self.stop()
7   loop()
8 }
9 val recv: () ~> (Rec, Any) = coroutine { () =>
10  var res: Any = _
11  yieldval({ case x => res = x })
12  res
13 }

```

Event streams. Event streams expose the `onEvent` method, similar to `onSuccess` on futures. The `onEvent` method takes a callback that is invoked when the next event arrives. As shown in Listing 17, it is much more convenient to extract an event in the direct-style by invoking a `get` statement, instead of installing a callback.

In Listing 37, we implement the method `get` on the event stream of type `Events[T]`. We declare a type alias `Install` that represents a function that installs a callback to the event stream. When an `Install` function is invoked with a function `f`, the function `f` is passed as a callback to some event stream.

The `react` method is similar to the `act` method for actors – it delimits the suspendable part of the event-driven program. The `react` method starts and resumes the coroutine. When the coroutine yields an `Install` function, the `react` method uses the `Install` function to install a callback that recursively resumes the coroutine.

The `get` method is called by the users to extract the next event out of a reactor’s event stream. Its implementation yields an `Install` function that installs the callback on the event stream by calling `onEvent`. The event stream callback sets the result variable and invokes the continuation function `f`. This technique is similar to the actor use-case, but the difference is that it abstracts over what `become` is.

■ **Listing 37** Direct-style event streams.

```

1 type Install = (() => Unit) => Unit
2 def react(c: () ~> (Install, Unit)) = {
3   val i = c.start()
4   def loop(): Unit = if (i.resume) i.value(loop)
5   loop()
6 }
7 def get[T](e: Events[T]): () ~> (Install, T) = coroutine { () =>
8   var res: T = _
9   val install = (f: () => Unit) => e.onEvent(x => {
10    res = x
11    f()
12  })
13   yieldval(install)
14   res
15 }

```

We note that, in this example, it might have been more natural to yield an event stream directly, instead of yielding `Install` functions. However, the event stream is parametric in the type of events, and the coroutine would always have to yield event streams of the same event type. The `Install` function hides the event type inside the `get` function, and allows a more flexible event stream API.

A Concurrent Specification of POSIX File Systems

Gian Ntzik

Imperial College London & Amadeus, UK
gian.ntzik@amadeus.com

Pedro da Rocha Pinto

Imperial College London, UK
pmd09@doc.ic.ac.uk

Julian Sutherland

Imperial College London, UK
jhs110@doc.ic.ac.uk

Philippa Gardner

Imperial College London, UK
pg@doc.ic.ac.uk

Abstract

POSIX is a standard for operating systems, with a substantial part devoted to specifying file-system operations. File-system operations exhibit complex concurrent behaviour, comprising multiple actions affecting different parts of the state: typically, multiple atomic reads followed by an atomic update. However, the standard's description of concurrent behaviour is unsatisfactory: it is fragmented; contains ambiguities; and is generally under-specified. We provide a formal concurrent specification of POSIX file systems and demonstrate scalable reasoning for clients. Our specification is based on a concurrent specification language, which uses a modern concurrent separation logic for reasoning about abstract atomic operations, and an associated refinement calculus. Our reasoning about clients highlights an important difference between reasoning about modules built over a heap, where the interference on the shared state is restricted to the operations of the module, and modules built over a file system, where the interference cannot be restricted as the file system is a public namespace. We introduce specifications conditional on *context invariants* used to restrict the interference, and apply our reasoning to the example of lock files.

2012 ACM Subject Classification Theory of computation → Program verification

Keywords and phrases POSIX, concurrency, file systems, refinement, separation logic, atomicity

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2018.4

Funding EPSRC Grants EP/H008373/1, EP/K008528/1 and EP/L016796/1

1 Introduction

POSIX [2] is a standard for operating systems, with a substantial part devoted to specifying file-system operations. File-system operations exhibit complex fine-grained concurrent behaviour, in the sense that they comprise multiple actions affecting different parts of the state: typically, multiple atomic¹ reads followed by an atomic update. The standard's description of this complex concurrent behaviour is unsatisfactory: it is fragmented; contains ambiguities; and is generally under-specified. There has been much work on formal, mathematical

¹ Atomic in the sense of *linearisability* [18], where operations appear to take effect at a single discrete point in time.



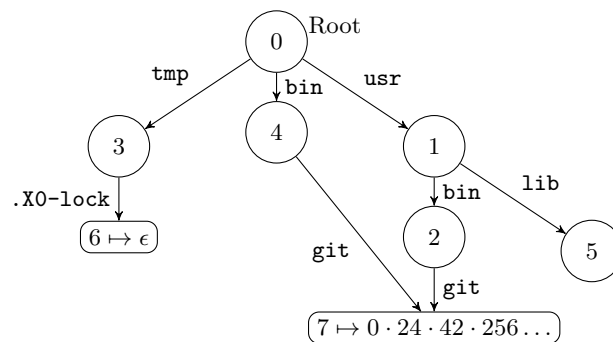
specifications of POSIX file systems, but no formal description of its concurrent behaviour: the work either restricts to sequential fragments (for example [3, 19, 24, 15, 16, 17, 7, 30]); or takes a coarse-grained view of concurrency that does not capture the POSIX behaviour [33].

Although poorly described, there is a consensus between major file-system implementations on what the concurrent behaviour of POSIX file systems should be. File-system operations (such as unlinking files) typically traverse paths to identify the files or directories on which they will act. Path traversal comprises a sequence of multiple atomic reads, each looking up a component of the path within a directory. Other operations (such as renaming files) exhibit the more complex behaviour of resolving multiple paths. Since POSIX does not specify the order in which multiple paths are resolved, the atomic reads of multiple path traversals can be arbitrarily interleaved. After the path resolution, other atomic actions perform the intended update of the file-system operation. In summary, file-system operations are sequential and parallel combinations of atomic actions.

We provide the first formal concurrent specification of POSIX file systems using a specification language based on concurrent separation logic. Such separation logics provide compositional reasoning about fine-grained concurrency and atomic operations: for example, the TaDA program logic [9, 8] uses a first-order approach to atomicity; the Iris framework [23] encodes the TaDA atomicity, using a higher-order approach initially introduced by Jacobs [20]; and the FCSL logic [26] uses histories. With TaDA, we are able to reason directly about atomic operations by introducing abstract atomic triples. However, such an atomic triple only specifies one atomic action for a given program statement. We cannot specify POSIX file-system operations which perform multiple atomic actions using TaDA. With Iris, it is possible to give a higher-order encoding of the TaDA atomic triples, yielding multiple atomic operations for free. We spent a considerable amount of time trying to use Iris to specify POSIX file-system operations, but found that the Hoare-style specifications were getting too complex. The issue is that the multiple atomic actions in POSIX are not simple linear sequences of atomic steps, but exhibit patterns of control flow which are better associated with program statements than logical assertions. The same issue also arises with FCSL [26].

We introduce TaDA-Refine, a specification language for specifying multiple atomic actions using TaDA assertions in the basic atomic statements, and an associated refinement calculus [4] for verifying clients. Our approach is inspired by the work of Turon and Wand [36], which was the first to combine such a specification language with separation-logic reasoning [32]. They introduced a refinement calculus for reasoning about *atomicity abstraction*, where a specification program appears to perform an operation in one atomic step even though its implementation takes many steps. They can verify that operations on simple data structures, such as incrementing a non-blocking counter, can be abstracted to atomic specification statements. They introduce an ownership discipline, formally captured by the notion of fenced refinement, to verify operations on more complex data structures such as a non-blocking stack. In contrast, we are able to reason about complex data structures using assertions and laws inspired from modern concurrent separation logics.

Our specifications of POSIX file-system operations take the form of simple programs from the TaDA-Refine specification language, built from *atomic specification statements*. An atomic specification statement has the form $\forall \vec{x}. \langle P, Q \rangle$, where P and Q are TaDA assertions for describing shared state. It provides an abstract description of operations that, for arbitrary \vec{x} , atomically updates states satisfying precondition P to states satisfying postcondition Q . The associated refinement calculus gives subtle behaviour to these atomic statements. For example, using the stuttering law of refinement, an equivalent specification program is $\forall \vec{x}. \langle P, P \rangle; \forall \vec{x}. \langle P, Q \rangle$. In fact, the combination of stuttering and mumbling laws with the



■ **Figure 1** Example snapshot of a file-system graph.

universal quantification means that the atomic statements are robust to the environment changing the values of the \vec{x} over time. In §5 and the technical report [29], we demonstrate that the presence of these laws means that the TaDA-Refine laws, model and soundness proof are significantly simpler than those of TaDA.

We use TaDA-Refine to verify clients of POSIX file systems, often using derived *hybrid specification statements* to reason about both atomic and non-atomic behaviour within one specification. Our client reasoning is different from the usual reasoning about concurrent heap modules using concurrent separation logics. A heap is a private namespace in the sense that a thread can safely access only what its been given through allocation or ownership transfer. Concurrent modules built over a heap restrict the interference on the shared resource they encapsulate, by only allowing access to the resource via the module operations. In contrast, a file system is a public namespace in the sense that a process or thread by default has access to any part of the file system. File access permissions can only enforce restrictions to sets of processes. It is not possible for a thread to keep part of a file system hidden from the rest of the system to restrict interference. Instead, the interference must be explicitly restricted by the reasoning. We do this by introducing *context invariants* to our specifications. We study lock files which provide a simple example to introduce context invariants. A lock file is a regular file under a path. If the lock file exists, the lock it represents is locked. Otherwise, the lock is unlocked. Our context invariant ensures that the path must remain fixed, and the lock file can only be added and removed using the lock operations, not the file-system operations. Other examples of our client reasoning include named pipes which build on lock files, and an email server to demonstrate the importance of reasoning about the full concurrent behaviour of POSIX file systems.

2 POSIX File-system Primer

Most readers will have a basic understanding of POSIX file systems. They will perhaps have less of an understanding of the concurrent behaviour of the file-system operations. We describe the fragment of POSIX used in this paper, and illustrate why the concurrent behaviour is poorly specified in the standard.

2.1 POSIX File-systems

A file system is an abstraction used to organise data, typically stored in some storage medium such as a disk. In POSIX, this abstraction takes the form of a *directed graph*. In figure 1, we give an example of an instance of such a file-system graph. The nodes in the graph

Basic types:

$\iota_0, \iota, j, \dots \in \text{INODES}$: countable set of inode numbers, ι_0 is the root

$a, b, \dots \in \text{FNAMES}$: countable set of filenames

$\text{BYTES} \triangleq \{n \in \mathbb{N} \mid 0 \leq n < 2^8\}$ $\text{ERRS} \triangleq \{\text{ENOENT}, \text{ENOTDIR}, \text{ENOTEMPTY}, \dots\}$

$\text{PATHS}' \ni p ::= a \mid a/p$ $\text{PATHS} \triangleq \text{PATHS}' \cup \{\emptyset_p\}$

File-system structure:

$FS \in \mathcal{FS} \triangleq \text{INODES} \xrightarrow{\text{fin}} \text{LINKS} \uplus \text{FILEDATA}$ $\text{LINKS} \triangleq \text{FNAMES} \xrightarrow{\text{fin}} \text{INODES}$

$\text{FILEDATA} \triangleq \text{BYTES}^*; (\{\emptyset\}^*; \text{BYTES}^?)^*$ where \emptyset denotes a file gap

Notation:

$\text{isfile}(o) \triangleq o \in \text{FILEDATA}$ $\text{isdir}(o) \triangleq o \in \text{LINKS}$ $\text{iserr}(o) \triangleq o \in \text{ERRS}$

$\iota \in FS \triangleq \iota \in \text{dom}(FS)$ $a \in FS(\iota) \triangleq a \in \text{dom}(FS(\iota))$

■ **Figure 2** File-system structure, basic types and some notation.

are *files*. There are different types of files. For this paper, we are primarily interested in *directories* which are denoted as circles in figure 1, and *regular files* which are denoted as curved rectangles. Each file is uniquely identified by an *inode number*, or henceforth simply an inode. In figure 1, the inodes are integers with 0 denoting the root directory.

Directories store *links*² to other files. Each link has an associated name which is unique to the directory and, thus, the links give the files their names. In figure 1, the links are given by the labelled edges. Regular files contain *file data* which are sequences of *bytes* which need not necessarily be contiguous. In figure 1, the notation $7 \mapsto 0 \cdot 24 \cdot 42 \cdot 256$ describes a regular file with inode 7 and the sequence of bytes $0 \cdot 24 \cdot 42 \cdot 256$. Regular files can be linked more than once, as is the case with the file with inode 7 in figure 1.

In figure 2, we give the basic mathematical definitions for the file-system structure that we use throughout the paper. We give a simplified view of the file-system structure which is enough to introduce our reasoning. In particular, we omit features such as symbolic links, the special filenames “.” and “..”, and file-access permissions. These features are orthogonal to reasoning about the concurrent behaviour of file-system operation and are discussed further in Ntzik’s thesis [27].

The basic types, given by the sets INODES with a distinguished inode ι_0 for the root directory, FNAMES and BYTES, are self-explanatory. The error types, given by the set ERRS, consists of the errors used by POSIX. We describe them as we use them in examples. The paths describe absolute paths starting from the root directory; a model in [27] also uses relative paths which start from a particular inode. The paths, given by the set PATHS, is either the empty path \emptyset_p or a finite sequence of file names written as $a_1/\dots/a_n$.

A *file-system structure* is a finite map from inodes to their contents, given by the set $\text{LINKS} \uplus \text{FILEDATA}$. For a directory which stores links to other files, the content is described formally as a finite partial function from filenames to inodes, given by the set LINKS. A directory is empty if its link function has the empty domain. For a regular file, the context is a sequence of bytes of a regular language, given by the set FILEDATA, where \emptyset denotes a gap in the sequence and ϵ denotes the empty sequence. A file-system structure is well formed if there are no dangling links.

² In POSIX, the terms link, hard link, directory entry, and entry mean the same thing.

A file-system structure is shared across all processes and is inherently concurrent. We have given concurrent specifications for operations of a core fragment of POSIX file systems. The fragment comprises the operations `mkdir`, `rmdir`, `link`, `unlink`, `rename`, `stat`, `open`, `close`, `read`, `write`, `lseek`, `opendir`, `closedir`, `readdir`, `pread` and `pwrite`. The fragment is significant, in that it includes most of the primitive structural commands that manipulate the file-system directory structure and the primitive input-output operations that change the contents of regular files. For this paper, we motivate our specifications by focusing in the structural operations `unlink` and `link`, and the input-output operations `read` and `write`. We also use a number of other operations in our client examples. The full specification of the fragment is available in Ntzik's thesis [27].

2.2 Concurrent Behaviour: the unlink operation

The POSIX file-system standard is a mature English standard with a comparatively clear description of the sequential behaviour of file-system operations. The description of the concurrent behaviour of file-system operations is much less clear. It is fragmented, contains ambiguities and is generally under-specified.

A particular difficulty lies with the POSIX atomicity guarantees for file-system operations. To illustrate this point, let us consider the `unlink` operation. Its sequential behaviour is straightforward. According to the POSIX standard (volume XSH, section 3), `unlink(path)` removes the link identified by the `path` argument. For example, using the file-system graph of figure 1, `unlink(/usr/bin/git)` first resolves the path `/usr/bin`, starting from the root directory, following the links `usr` and `bin` to yield the directory 2. It then removes the link named `git` to the regular file with inode 7 from this directory. If `unlink` is unable to resolve the path because, for example, one of the names in the path does not exist in the appropriate directory, it returns an error. Furthermore, POSIX allows some flexibility with the behaviour, in that it allows implementations to return an error if the path identifies a link to a directory rather than a regular file. In other words, `unlink` is permitted to exhibit non-determinism due to different implementation decisions.

In a different section of the standard (volume XSH, section 2.9.7), `unlink` is specified as *atomic*, which suggests that the whole process of resolving the path and removing the link to the identified file is logically indivisible. However, this is a common misconception. Hidden in the fine print of a section describing the specification rationale for `unlink`, we find the statement:

“... Any part of the path of a file could be changed in parallel to a call to `unlink`, resulting in unspecified behavior ...”.

Here, *unspecified behaviour* means that we cannot predict whether the operation is going to succeed or error, even if we know the file-system's state when `unlink` is invoked. When the POSIX standard describes `unlink` as atomic (volume XSH, section 2.9.7), it means that the removal of the link that the path identifies is atomic. The path resolution itself comprises a sequence of atomic lookups that traverse the file-system graph by following the path. This fragmentation and ambiguity of the description of the `unlink` operation in the standard applies to all the POSIX operations that resolve paths. Such operations are sequences of atomic read operations followed by an atomic update which removes, adds, moves (renames) and looks up individual links in a directory. This behaviour is demonstrated by virtually all major file-system implementations. It has two interesting implications. First, because the file system can be changed arbitrarily by the concurrent environment between the individual atomic steps comprising an operation, it is impossible to determine whether the operation

is going to succeed or error just by examining the file-system state at the invocation point. The result depends on the concurrent environment and the scheduler interleavings. Thus, POSIX operations exhibit non-determinism due to concurrent interleavings. Second, if an operation succeeds, it does not necessarily mean that the path given as argument exists, or even existed at any single point in time. It merely means that the operation was able to resolve the path.

POSIX exhibits this ambiguity only for the operations that resolve paths. It is important to understand what the intentions of the standard are with respect to their behaviour. We suspect that POSIX does intend for these operations to have an atomic *effect*, but with consideration to implementation performance. A truly atomic implementation, where both the path resolution and the effect at the end of the path takes place in a single observable step, would require synchronisation over the entire file-system graph. For most implementations, the performance impact of this coarse-grained behaviour would be unacceptable. Therefore, the wording of the standard allows path resolution to be implemented non-atomically, as a sequence of atomic steps, where each looks up where the next name in the path leads to. The specification of path resolution (volume XBD, section 4.13), is silent on this matter.

Our interpretation of the standard’s intentions is verified in the Austin Group mailing list [1].³ Path resolution itself consists of a sequence of atomic lookups that traverse the file-system graph by following the path. In the case of `unlink`, the effect of removing the resolved link from the file-system graph is atomic. In fact, this is part of a common tenet followed by virtually all major file-system implementations: removing (`unlink`), adding (`open`, `creat`, `link`), moving (`rename`) and looking up individual links in a directory (path resolution steps) are implemented atomically. In other words, when accounting for concurrency, POSIX operations that resolve paths are sequences of atomic operations.

3 TaDA-Refine Specification Examples

We first introduce our TaDA-Refine specifications of file-system operations by example. In particular, we specify operations on links and I/O operations on regular files. To account for the fact that file-system operations perform sequences of atomic operations, our specifications take the form of “programs” in a simple specification programming language.

3.1 Operations on links

In §2.2, we have informally described the behaviour of the `unlink(path)` operation: it performs a sequence of atomic steps, first to resolve the argument *path* and then to remove the link to the file identified by the path. We define the specification of `unlink` using the following TaDA-Refine *specification program*:

```
let unlinkSpec(path)  $\triangleq$  let p = dirname(path);
                           let a = basename(path);
                           let r = resolve(p,  $\iota_0$ );
                           if  $\neg$ iserr(r) then
                               return link_delete(r, a)  $\sqcup$  link_delete_notdir(r, a)
                           else return r fi
```

The specification program initially splits the *path* argument to the path prefix *p* and last name *a*, using `dirname` and `basename` respectively. If *path* is only one name, then `dirname`

³ Thread: “Atomicity of path resolution”, Date: 21 Apr 2015.

returns `null`. The path prefix p is then resolved by calling the function `resolve(p, ι_0)`. The second argument to `resolve` is the inode number of the directory from which to start the path resolution. In figure 1, this would be the directory with inode 0. To simplify the presentation, we define the specifications in this paper in terms of absolute paths, and therefore we start the resolution from the root directory, which has the known fixed inode ι_0 . If the resolution fails with an error code, we return it. If the resolution succeeds, the return value is the inode of the directory containing the link we want to remove. POSIX allows implementations to return an error if the link we want to remove is a link to a directory. This freedom of choice given to implementations introduces *angelic non-determinism*. An implementation is allowed to choose which behaviour to implement. On the other hand, clients must be robust with respect to both behaviours if they wish to be portable. To account for this, we use the non-deterministic *angelic choice* operator (\sqcup) to join the atomic operations `link_delete` and `link_delete_notdir`.

`resolve` is defined as a function that recursively follows $path$, starting from the initial directory with inode ι :

```

letrec resolve( $path, \iota$ )  $\triangleq$  if  $path = \text{null}$  then return  $\iota$  else
    let  $a = \text{head}(path)$ ;
    let  $p = \text{tail}(path)$ ;
    let  $r = \text{link\_lookup}(\iota, a)$ ;
    if  $\text{iserr}(r)$  then return  $r$  else return resolve( $p, r$ ) fi
fi

```

The `head` and `tail` operations return the first name and the path postfix of the path argument. Note that if $path$ is a single name, then `tail` returns `null`. In each step, `resolve` calls the atomic operation `link_lookup(ι, a)`, to get the inode of the file pointed to by the link named a , if that link exists in the directory with inode ι . If the link a does not exist in the ι directory, or if the ι file is not a directory, `link_lookup` returns an error, the resolution stops and the error is immediately returned. The procedure returns the resolved inode when there is no more path to resolve, i.e. the postfix p of the $path$ argument is `null`.

Any implementation of the `unlink` operation must exhibit behaviour given by the specification program `unlinkSpec`. In other words, a correct implementation must be a refinement of our specification program: `unlink($path$) \sqsubseteq unlinkSpec($path$)`.

In §5 and the technical report [29], we formally define our specification language and an associated refinement calculus. The resulting refinement relation, \sqsubseteq , is contextual meaning that, in any context, `unlink` can be replaced by `unlinkSpec` to achieve the same behaviour. Therefore, to reason about a client (a particular context), we can replace an implementation with its specification.

To complete our `unlink` specification, we need to define the primitive atomic operations `link_lookup`, `link_delete` and `link_delete_notdir` that do the actual lookup and deletion of a link. Note that these are not POSIX operations, but abstract operations corresponding to the basic atomic actions that POSIX operations perform. We use *atomic specification statements*, $\forall \vec{x}. \langle P, Q \rangle$, to denote any program that atomically updates a state satisfying the precondition P to a state satisfying the postcondition Q , inspired by Morgan's specification statements [25]. The universal quantifier binds \vec{x} to both the precondition and postcondition, and declares that the operation is atomic for all values of \vec{x} .

We define the atomic operations `link_lookup`, `link_delete` and `link_delete_notdir` as the atomic specification statements given in figure 3. Consider `link_delete` used in the definition of `unlinkSpec` earlier. There are three cases composed with \sqcap , which we will

explain shortly. Consider the first case:

$$\forall FS. (\text{fs}(FS) \wedge \text{isdir}(FS(\iota)), a \in FS(\iota) \Rightarrow \text{fs}(FS[\iota \mapsto FS(\iota) \setminus \{a\}]) * \text{ret} = 0)$$

In the precondition $\text{fs}(FS) \wedge \text{isdir}(FS(\iota))$, the abstract predicate $\text{fs}(FS)$ states that the file-system structure is given by the file-system graph FS , and the pure predicate $\text{isdir}(FS(\iota))$ states that a directory with inode ι must exist in that file-system graph. In the postcondition, we use the notation $f[x \mapsto v]$ to denote the function that maps x to v and all other elements of the domain of f to $f(x)$, and $f \setminus S$ to denote the restriction of f to $\text{dom}(f) \setminus S$. The postcondition states that if, at the point the atomic update takes effect, the link named a exists in the directory with inode ι , then the link is removed and the return variable ret is bound to 0. As a convention, we use ret within a function to bind its return value.

The other two cases specify erroneous behaviour. The first error case, defined by `enoent`, specifies that if a link named a does not exist in the directory with inode ι then the return variable is bound to the POSIX error code `ENOENT`. The second error case specifies that if the inode ι does not identify a directory then the error code `ENOTDIR` is returned. Note that the error cases do not modify the file system.

The three specification cases are composed with the non-deterministic *demonic choice* operator \sqcap . We use demonic choice to account for the non-determinism of a specification due to scheduling behaviour. In the case of `link_delete`, which of the three possible behaviours we observe in a particular execution depends not only on the environment, but also on which of the possible interleavings the scheduler decides to execute. Thus we consider the scheduler to act as a demon and we call such specifications demonic. For example, `link_delete` handles errors by returning the error code to the client. When reasoning about a particular client, if we have information that restricts the environment, for example by requiring some path to always exist, we can elide the cases that are no longer applicable. On the other hand, an implementation of a demonic specification must implement all the cases. For example, an implementation of `link_delete` must implement all three atomic specification statements.

The definition of `link_delete_notdir` is similar, except that it succeeds only when the link being removed does not link a directory, and an extra error case is added for when it does. `link_lookup` has the same error cases as `link_delete`, but does not modify the file system, simply returning the target inode of the link named a , if it exists in the directory with inode ι .

Now, let us consider the `link(source, target)` operation. Informally, it creates a new link identified by the path `target` to the file identified by `source`, if it does not already exist. Formally, we give the following refinement specification:

```

link(source, target)
  ⊑ let ps = dirname(source); let a = basename(source);
    let pt = dirname(target); let b = basename(target);
    let rs, rt = resolve(ps,  $\iota_0$ ) || resolve(pt,  $\iota_0$ );
    if ¬iserr(rs) ∧ ¬iserr(rt) then
      return link_insert(rs, a, rt, b) ⊔ link_insert_notdir(rs, a, rt, b)
    else if iserr(rs) ∧ ¬iserr(rt) then return rs
    else if ¬iserr(rs) ∧ iserr(rt) then return rt
    else if iserr(rs) ∧ iserr(rt) then return rs ⊔ return rt fi

```

Note that the operation has to resolve two paths before the actual linking is attempted. POSIX does not specify the order in which multiple paths are resolved. Therefore, we compose the two `resolve` invocations in parallel, with `||`. This allows implementations to

```

let link_lookup( $\iota, a$ )  $\triangleq$ 
   $\forall FS. \langle fs(FS) \wedge isdir(FS(\iota)), a \in FS(\iota) \Rightarrow fs(FS) * ret = FS(\iota)(a) \rangle$ 
   $\sqcap$  return enoent( $\iota, a$ )  $\sqcap$  return enotdir( $\iota$ )

let link_delete( $\iota, a$ )  $\triangleq$ 
   $\forall FS. \langle fs(FS) \wedge isdir(FS(\iota)), a \in FS(\iota) \Rightarrow fs(FS[\iota \mapsto FS(\iota) \setminus \{a\}]) * ret = 0 \rangle$ 
   $\sqcap$  return enoent( $\iota, a$ )  $\sqcap$  return enotdir( $\iota$ )

let link_delete_notdir( $\iota, a$ )  $\triangleq$ 
   $\forall FS. \langle fs(FS) \wedge isdir(FS(\iota)), a \in FS(\iota) \Rightarrow fs(FS[\iota \mapsto FS(\iota) \setminus \{a\}]) * ret = 0 \rangle$ 
   $\sqcap$  return enoent( $\iota, a$ )  $\sqcap$  return enotdir( $\iota$ )  $\sqcap$  return err_nodir_links( $\iota, a$ )

let link_insert( $\iota, a, j, b$ )  $\triangleq$ 
   $\forall FS. \langle fs(FS) \wedge isdir(FS(\iota)) \wedge isdir(FS(j)),$ 
   $\left. \begin{array}{l} a \in FS(\iota) \wedge b \notin FS(j) \Rightarrow fs(FS[j \mapsto FS(j)[b \mapsto FS(\iota)(a)]) * ret = 0 \end{array} \right\rangle$ 
   $\sqcap$  return enoent( $\iota, a$ )  $\sqcap$  return eexist( $j, b$ )  $\sqcap$  return enotdir( $\iota$ )  $\sqcap$  return enotdir( $j$ )

let link_insert_notdir( $\iota, a, j, b$ )  $\triangleq$ 
   $\forall FS. \langle fs(FS) \wedge isdir(FS(\iota)) \wedge isdir(FS(j)),$ 
   $\left. \begin{array}{l} isfile(FS(\iota)(a)) \wedge b \notin FS(j) \Rightarrow fs(FS[j \mapsto FS(j)[b \mapsto FS(\iota)(a)]) * ret = 0 \end{array} \right\rangle$ 
   $\sqcap$  return enoent( $\iota, a$ )  $\sqcap$  return eexist( $j, b$ )
   $\sqcap$  return enotdir( $\iota$ )  $\sqcap$  return enotdir( $j$ )  $\sqcap$  return err_nodir_links( $\iota, a$ )

let enotdir( $\iota$ )  $\triangleq \forall FS. \langle fs(FS) \wedge \neg isdir(FS(\iota)), fs(FS) * ret = ENOTDIR \rangle$ 

let enoent( $\iota, a$ )  $\triangleq \forall FS. \langle fs(FS) \wedge isdir(FS(\iota)), a \notin FS(\iota) \Rightarrow fs(FS) * ret = ENOENT \rangle$ 

let eexist( $\iota, a$ )  $\triangleq \forall FS. \langle fs(FS) \wedge isdir(FS(\iota)), a \in FS(\iota) \Rightarrow fs(FS) * ret = EEXIST \rangle$ 

let err_nodir_links( $\iota, a$ )  $\triangleq \forall FS. \langle fs(FS) \wedge isdir(FS(\iota)), isdir(FS(\iota)(a)) \Rightarrow fs(FS) * ret = EPERM \rangle$ 

```

■ **Figure 3** Specifications of atomic operations for links and associated error cases.

not only resolve the paths in any order, but also to interleave the two resolutions. The link insertion is attempted when both resolutions succeed. In that case, analogously to `unlink`, we use angelic choice between `link_insert` and `link_insert_notdir`. The former allows the link to be created for any link, even to a directory, whereas the latter considers this erroneous. The atomic specification statements for both are defined in figure 3. Error handling must be robust against errors from both resolutions. Note that if both resolutions error, either error code is returned. In general, a client is unable to determine which path resolution triggered the error.

3.2 I/O operations on regular files

POSIX defines `read` and `write` as the primitive operations for reading and writing data to regular files. The `read` operation reads a sequence of bytes from a regular file to a buffer in the heap, whereas the `write` operation writes a sequence of bytes stored in the buffer to a regular file. These operations do not identify the file they update with a path, but with a *file descriptor* which acts as a reference to a file. To create a file descriptor for a file, a client must first open the file for I/O using the operator `open(path, fl)`, where *path* describes the file to be opened and *fl* controls the behaviour of `open` on subsequent I/O operations such as `read` and `write`.

$$\text{let write_off}(fd, ptr, sz) \triangleq$$

$$\forall FS, o \in \mathbb{N}. \left\langle \begin{array}{l} \text{fs}(FS) \wedge \text{isfile}(FS(\iota)) * \text{fd}(fd, \iota, o, fl) \wedge \text{iswrfd}(fl) * \text{buf}(ptr, \bar{b}) \wedge \text{len}(\bar{b}) = sz, \\ \text{fs}(FS[\iota \mapsto FS(\iota)[o \leftarrow \bar{b}]]) * \text{fd}(fd, \iota, o + sz, fl) * \text{buf}(ptr, \bar{b}) * \text{ret} = sz \end{array} \right\rangle$$

$$\text{let write_badf}(fd) \triangleq \forall o \in \mathbb{N}. \langle \text{fd}(fd, \iota, o, fl) \wedge \text{O_RDONLY} \in fl, \text{fd}(fd, \iota, o, fl) * \text{ret} = \text{EBADF} \rangle$$

$$\text{let read_norm}(fd, ptr, sz) \triangleq$$

$$\forall FS, o \in \mathbb{N}. \left\langle \begin{array}{l} \text{fs}(FS) \wedge \text{isfile}(FS(\iota)) * \text{fd}(fd, \iota, o, fl) * \text{buf}(ptr, \bar{b}_s) \wedge \text{len}(\bar{b}_s) = sz, \\ \exists \bar{b}_t. \text{fs}(FS) * \text{fd}(fd, \iota, o + \text{ret}, fl) * \text{buf}(ptr, \bar{b}_s \uparrow \bar{b}_t) \wedge \bar{b}_t = FS(\iota)[o, sz] * \text{ret} = \text{len}(\bar{b}_t) \end{array} \right\rangle$$

$$\text{let read_badf}(fd) \triangleq \forall o \in \mathbb{N}. \langle \text{fd}(fd, \iota, o, fl) \wedge \text{O_WRONLY} \in fl, \text{fd}(fd, \iota, o, fl) * \text{ret} = \text{EBADF} \rangle$$

where we write $\forall \vec{x}, x \in X. \langle P, Q \rangle$ to mean $\forall \vec{x}, x. \langle P \wedge x \in X, Q \wedge x \in X \rangle$.

■ **Figure 4** Specification of atomic read and write abstract operations.

POSIX mandates that implementations of `read` and `write` must behave atomically when used on regular files [2]. We give the following refinement specifications to `read` and `write`, defined using the demonic choice of abstract operations given in figure 4:

$$\text{read}(fd, ptr, sz) \sqsubseteq \text{return read_norm}(fd, ptr, sz) \sqcap \text{read_badf}(fd)$$

$$\text{write}(fd, ptr, sz) \sqsubseteq \text{return write_off}(fd, ptr, sz) \sqcap \text{write_badf}(fd)$$

where fd identifies the appropriate file descriptor and ptr references the buffer storing a sequence of bytes with size sz .

In figure 4, consider the atomic specification statement of `write_off`. The precondition requires fd to be a file descriptor for the file with inode ι , with current file offset o and flags fl . Note that the current file offset is bound by the universal quantifier, meaning that until `write_off` takes effect, the environment can concurrently modify it, with the proviso it remains a valid offset (a natural number). The predicate $\text{iswrfd}(fl) \triangleq \text{O_WRONLY} \in fl \vee \text{O_RDWR} \in fl$ states that file descriptor must have been opened for writing. Furthermore, the predicate $\text{buf}(ptr, \bar{b})$ states that ptr points to a heap-based buffer storing the byte sequence \bar{b} . The postcondition states that the byte sequence \bar{b} stored in the ptr buffer is written to the file, offset from the start of the file (offset 0) by o . Any existing bytes from offset o onward, up to the length of \bar{b} , are overwritten. The current file offset associated with the file descriptor is incremented by the number of bytes written, which the operation also returns. The `write_badf` abstract operation returns the `EBADF` error, if the file descriptor has not been opened for writing, and does not modify the file.

Note that we have specified both operations as happening atomically, as is mandated by POSIX. However, not all implementations follow the POSIX specification. For example, in the ext2 file system, the reads and writes are only atomic up to page-size number of bytes. Reads and writes of larger size are split into multiple atomic steps. It is straightforward to specify this kind of implementation-specific behaviour in our specification language. In addition, reading and writing to the heap buffer is not atomic in some modern implementations. In such implementations, the I/O operations behave atomically on the file contents and the file descriptor, but non-atomically on the heap buffer. To account for such behaviour, we require specification statements that combine atomic and non-atomic effects.

In TaDA-Refine, it is possible to derive the *hybrid specification statement*:

$$\forall \vec{x}. \exists \vec{y}. \{P'\} \langle P(\vec{x}), Q(\vec{x}, \vec{y}) \rangle \{Q'(\vec{x}, \vec{y})\}$$

We discuss this statement in detail in §5.3. Intuitively, this statement combines the atomic update from $P(\vec{x})$ to $Q(\vec{x}, \vec{y})$ with a non-atomic update from P' to $Q'(\vec{x}, \vec{y})$. Its purpose is twofold. First, it allows us to specify complex operations that have both atomic and non-atomic effects on different parts of the state, such as the I/O operations of some file-system implementations discussed earlier. Second, it is useful during atomicity proofs of implementations that also sequentially update privately owned resources.

4 TaDA-Refine Client Reasoning I: Lock Files

Ntzik’s thesis [27] provides several examples of client reasoning based on the formal specifications of POSIX file-system operations, such as those discussed in §3. Examples of client reasoning include real-world lock files, an implementation of named pipes using regular file I/O and lock files, and a concurrent interaction between an email client and email server that is highly sensitive to the multi-atomic nature of path resolution. In this paper, we concentrate on lock files.

The lock-file module is a widely-used module for implementing locks over the file system. We describe the lock-file module and provide verified specifications for its operations to demonstrate our reasoning with TaDA-Refine. These specifications are, however, limited. They are only valid under the assumption that the file system is shared via the lock-module interface. This assumption is not valid in general as the file system is a public namespace that can be accessed and modified by concurrently executing applications. In §6, we revisit this example and introduce *context invariants* to address this issue.

The lock-file concept is simple. A lock file is a regular file, under a fixed path. If the lock file exists, the lock it represents is locked. Otherwise, the lock is unlocked. For example, `/tmp/.X0-lock` is a typical lock file in contemporary Linux systems and, in figure 1, the lock it represents is locked.

Consider the following implementation of a lock-file module with two operations, `lock(lf)` and `unlock(lf)`, where `lf` is the path identifying the lock file:

```

letrec lock(lf)  $\triangleq$ 
  let fd = open(lf, O_CREAT|O_EXCL);
  if iserr(fd) then lock(lf)
  else close(fd) fi

unlock(lf)  $\triangleq$  unlink(lf)

```

The `lock` operation attempts to create the lock file at path `lf` by invoking `open`. This operation is used to open files for input/output (I/O) and to create new files. The second argument to `open` is a composition of the flags `O_CREAT` and `O_EXCL`, which causes `open` to create a file at the given path if one does not already exist; otherwise, an error is returned. Thus, if `open` returns an error we try again, with a recursive call to `lock`. If it succeeds, we invoke `close` to close the file descriptor returned by `open`. Note that lock files, essentially, follow the same implementation pattern as spin locks.

The `open` operation exhibits different behaviour depending on the flags used as the second argument and we give its full specification in the technical report [29]. For presentation simplicity we define the specification only in terms of the flags used in `lock`:

```

open(path, O_CREAT|O_EXCL)
  ⊑ let p = dirname(path);
    let a = basename(path);
    let r = resolve(p,  $\iota_0$ );
    if  $\neg$ iserr(r) then
      return link_new_file(r, a)
        □ eexist(r, a) □ enotdir(r)
    else return r fi

```

where `link_new_file(ι , a)` is defined as follows:

$$\forall FS. \langle \text{fs}(FS) \wedge \text{isdir}(FS(\iota)), a \notin FS(\iota) \Rightarrow \exists \iota'. \text{fs}(FS[\iota \mapsto FS(\iota)[a \mapsto \iota']][\iota' \mapsto \epsilon]) * \text{fd}(\text{ret}, \iota', 0) \rangle$$

This specifies the creation of a new empty, regular file at inode ι' , and the addition of a link named a to the new file within the directory with inode ι , if the link does not already exist. The operation allocates and returns a new file descriptor. The predicate `fd(ret , ι' , 0)` asserts that the return value is a file descriptor for the file with inode ι' , and the offset from which reads and writes to the file occur, via this file descriptor, is set to 0.

By contextual refinement, we can replace the `open` and `unlink` with their specifications and thus derive a specification for `lock` and `unlock` respectively. However, this would not be useful for reasoning about locks since it fails to capture the abstract lock behaviour. Instead, we want aim to establish a general abstract specification, such as the following:

$$\text{lock}(lf) \sqsubseteq \forall v \in \{0, 1\}. \langle \text{Lock}(s, lf, v), \text{Lock}(s, lf, 1) * v = 0 \rangle$$

$$\text{unlock}(lf) \sqsubseteq \langle \text{Lock}(s, lf, 1), \text{Lock}(s, lf, 0) \rangle$$

The abstract predicate `Lock(s , lf , v)` states the existence of a lock represented by a lock file at path lf , with state v , the value of which is either 0, if the lock is unlocked, or 1 if the lock is locked. The parameter, $s \in \mathbb{T}_1$, is a variable ranging over an abstract type. It serves to capture invariant information, specific to the implementation of the `Lock` predicate and is opaque to the client. The specification states that we can abstract each lock-file operation to a single atomic step that updates the state of the lock. In particular, the `lock` specification states that the environment can arbitrarily lock and unlock the lock, but the lock is atomically locked only when it is previously unlocked; the operation blocks while the lock is locked. The `unlock` specification states that the lock can only be atomically unlocked when the lock is locked.

The environmental interference allowed by the specification of the lock operation is due to the stuttering refinement law:

$$\text{ASTUTTER} \\ \forall \vec{x}. \langle P, P \rangle; \forall \vec{x}. \langle P, Q \rangle \sqsubseteq \forall \vec{x}. \langle P, Q \rangle$$

The environment can interleave between the two sequentially composed atomic specifications, allowing it to change the state of the lock over time, as long as the state remains in the set $\{0, 1\}$, otherwise, the precondition of the lock specification would be violated when it executes, leading to undefined behaviour.

In order to justify the two refinements of the module's specification, we must refine the abstract `Lock` predicates to the shared lock-file path lf in the file system according to the state of the lock. Additionally, we must enforce that the updates to the abstract state

$$\begin{array}{l}
\text{unlock}(lf) \equiv \text{unlink}(lf) \sqsubseteq \\
\begin{array}{l}
\text{let } p = \text{dirname}(\text{path}); \text{let } a = \text{basename}(\text{path}); \text{let } r = \text{resolve}(p, t_0) \sqsubseteq \\
\forall FS \in \mathcal{LF}(\text{path}). \langle \text{fs}(FS), \text{fs}(FS) \wedge p \xrightarrow{FS} r \wedge \text{isdir}(FS(r)) \wedge a \in FS(r) \rangle \\
\hline \text{HSTRENGTHEN} \\
\sqsubseteq \forall FS \in \mathcal{LF}(\text{path}). \{\text{true}\} \langle \text{fs}(FS), \text{fs}(FS) \rangle \{ p \xrightarrow{FS} r \wedge \text{isdir}(FS(r)) \wedge a \in FS(r) \}; \\
\text{if } \neg \text{iserr}(r) \text{ then} \\
\quad \text{return link_delete}(r, a) \\
\quad \quad \sqcup \text{link_delete_notdir}(r, a) \\
\quad \equiv \text{return link_delete}(r, a) \\
\quad \quad \sqcup (\text{link_delete}(r, a) \sqcap \text{err_nodir_links}(t, a)) \\
\hline \text{ABSORB} \\
\equiv \text{return link_delete}(r, a) \\
\hline \text{DCHOICEINTRO} \\
\sqsubseteq \forall FS. \langle \text{fs}(FS) \wedge \text{isdir}(FS(r)), a \in FS(r) \Rightarrow \text{fs}(FS[r \mapsto FS(r) \setminus \{a\}]) * \text{ret} = 0 \rangle \\
\hline \text{ACONS, AFRAME} \\
\sqsubseteq \forall FS \in \mathcal{LK}(\text{path}). \left\langle \begin{array}{l} \text{fs}(FS) \wedge p \xrightarrow{FS} r \wedge \text{isdir}(FS(r)) \wedge a \in FS(r) \wedge \neg \text{iserr}(r), \\ \exists FS'. \text{fs}(FS') \wedge \text{ulk}(FS, FS', lf) \end{array} \right\rangle \\
\text{else} \\
\quad \text{return } r \\
\quad \sqsubseteq \langle \text{true}, \text{ret} = r \rangle \\
\hline \text{ACONS} \\
\sqsubseteq \langle \text{true}, \text{true} \rangle \\
\hline \text{AFRAME} \\
\sqsubseteq \forall FS \in \mathcal{LK}(\text{path}). \left\langle \begin{array}{l} \text{fs}(FS) \wedge p \xrightarrow{FS} r \wedge \text{isdir}(FS(r)) \wedge a \in FS(r) \wedge \text{iserr}(r), \\ \text{fs}(FS) \wedge p \xrightarrow{FS} r \wedge \text{isdir}(FS(r)) \wedge a \in FS(r) \wedge \text{iserr}(r) \end{array} \right\rangle \\
\hline \text{ACONS} \\
\sqsubseteq \forall FS \in \mathcal{LK}(\text{path}). \left\langle \begin{array}{l} \text{fs}(FS) \wedge p \xrightarrow{FS} r \wedge \text{isdir}(FS(r)) \wedge a \in FS(r) \wedge \text{iserr}(r), \\ \exists FS'. \text{fs}(FS') \wedge \text{ulk}(FS, FS', lf) \end{array} \right\rangle \\
\text{fi} \\
\sqsubseteq \forall FS \in \mathcal{LK}(\text{path}). \left\langle \begin{array}{l} \text{fs}(FS) \wedge p \xrightarrow{FS} r \wedge \text{isdir}(FS(r)) \wedge a \in FS(r) \wedge \text{iserr}(r), \\ \exists FS'. \text{fs}(FS') \wedge \text{ulk}(FS, FS', lf) \end{array} \right\rangle \\
\hline \text{HSTRENGTHEN} \\
\left\{ p \xrightarrow{FS} r \wedge \text{isdir}(FS(r)) \wedge a \in FS(r) \right\} \\
\sqsubseteq \forall FS \in \mathcal{LK}(\text{path}). \langle \text{fs}(FS), \exists FS'. \text{fs}(FS') \wedge \text{ulk}(FS, FS', lf) \rangle \\
\quad \quad \quad \{ \text{true} \} \\
\sqsubseteq \forall FS \in \mathcal{LK}(lf). \{ \text{true} \} \langle \text{fs}(FS), \exists FS'. \text{fs}(FS') \wedge \text{ulk}(FS, FS', lf) \rangle \{ \text{true} \} \\
\equiv \forall FS \in \mathcal{LK}(lf). \langle \text{fs}(FS), \exists FS'. \text{fs}(FS') \wedge \text{ulk}(FS, FS', lf) \rangle \\
\hline \text{ACONS, AUSEATOMIC} \\
\sqsubseteq \langle \mathbf{Lock}_\alpha(lf, 1) * [G]_\alpha, \mathbf{Lock}_\alpha(lf, 0) * [G]_\alpha \rangle \\
\sqsubseteq \langle \mathbf{Lock}(s, lf, 1), \mathbf{Lock}(s, lf, 0) \rangle
\end{array}
\end{array}$$

■ **Figure 5** `unlock()` specification proof sketch.

of the lock by multiple threads follow a protocol: a thread can lock the lock only if it is unlocked and, similarly, can unlock the lock only if it is locked. We use *shared regions* to describe shared resources that are updated according to a particular protocol, using a technique first developed with RGSep [37] and now used by many of the concurrent separation logics [13, 11, 21, 34, 9, 23]. A shared region is an abstract object that encapsulates some underlying (concrete) state that is shared between multiple threads, with the proviso that it is only accessed atomically. We use $\mathbf{t}_\alpha(\vec{y}, x)$, to denote a shared region with identifier α from the set RID, of type \mathbf{t} , with parameters \vec{y} and abstract state x . For our current example, we introduce a region type **Lock** where regions of this type are parameterised by the lock-file path and the abstract state of the region corresponds to the state of the lock.

The shared region enforces a protocol on updates to the abstract state via a *labelled transition system*. The transitions between abstract region states are labelled by *guards*. Guards are abstract resources that can be taken from any user-defined separation algebra [6]. For our current example we only need a simple separation algebra with a single, indivisible guard \mathbf{G} and the empty guard $\mathbf{0}$. The partial, associative and commutative composition operator between these guards is defined by the axioms: $x \bullet \mathbf{0} = x = \mathbf{0} \bullet x$ for all $x \in \{\mathbf{0}, \mathbf{G}\}$. We define the labelled transition system for the **Lock** as follows:

$$\mathbf{G} : 0 \rightsquigarrow 1 \qquad \mathbf{G} : 1 \rightsquigarrow 0$$

A thread can only perform a transition between abstract region states if it owns the guard resource associated with the transition; in our current example that is the guard \mathbf{G} .

Having defined the **Lock**, its transition system and guards we can now define the interpretation of the abstract **Lock** predicate and abstract type \mathbb{T}_1 in terms of the region as follows:

$$\mathbb{T}_1 \triangleq \text{RID} \quad \text{Lock}(\alpha, lf, 0) \triangleq \mathbf{Lock}_\alpha(lf, 0) * [\mathbf{G}]_\alpha \quad \text{Lock}(\alpha, lf, 1) \triangleq \mathbf{Lock}_\alpha(lf, 1) * [\mathbf{G}]_\alpha$$

With the above interpretation we can refine the atomic specification statement of **lock** as follows:

$$\begin{aligned} \forall v \in \{0, 1\}. \langle \mathbf{Lock}_\alpha(lf, v) * [\mathbf{G}]_\alpha, \mathbf{Lock}_\alpha(lf, 1) * [\mathbf{G}]_\alpha * v = 0 \rangle \\ \sqsubseteq \forall v \in \{0, 1\}. \langle \text{Lock}(s, lf, v), \text{Lock}(s, lf, 1) * v = 0 \rangle \end{aligned}$$

and similarly for the atomic specification statement of **unlock**:

$$\langle \mathbf{Lock}_\alpha(lf, 1) * [\mathbf{G}]_\alpha, \mathbf{Lock}_\alpha(lf, 0) * [\mathbf{G}]_\alpha \rangle \sqsubseteq \langle \text{Lock}(s, lf, 1), \text{Lock}(s, lf, 0) \rangle$$

To refine the two updates further, we can apply the AUSEATOMIC refinement law which allows us to refine an update to the abstract region state to an update on the (concrete) state the region encapsulates. A slightly simplified version of this refinement law is as follows:

$$\frac{\forall x. (x, f(x)) \in \mathcal{T}_\mathbf{t}(\mathbf{G})^*}{\forall x. \langle I(\mathbf{t}_\alpha(y, x)) * P(x) * [\mathbf{G}]_\alpha, I(\mathbf{t}_\alpha(y, f(x))) * Q(x) \rangle \sqsubseteq \forall x. \langle \mathbf{t}_\alpha(y, x) * P(x) * [\mathbf{G}]_\alpha, \mathbf{t}_\alpha(y, f(x)) * Q(x) \rangle}$$

The premise of the law requires that an update from the abstract region state x to $f(x)$ must be in the reflexive, transitive closure of transitions guarded by \mathbf{G} denoted by $\mathcal{T}_\mathbf{t}(\mathbf{G})^*$. In the conclusion, an atomic update, satisfying the premise, on the interpretation of a region refines the same atomic update on the region itself.

However, in order to refine the **lock** atomic specification statement further according to AUSEATOMIC, we must define the interpretation of the **Lock** region states 0 and 1 to the underlying file-system states.

In the state 0 the lock file does not exist, whereas in the state 1 it does. In both states however, the path to the directory containing the lock file must exist. To assert the aforementioned statements we define the predicate $p \xrightarrow{FS} \iota$ to assert that the path p resolves to the file with inode ι in the file-system graph FS as follows:

$$\begin{aligned} \emptyset_p \xrightarrow{FS} \iota &\triangleq \iota \in \text{dom}(FS) & p \xrightarrow{FS} \iota &\triangleq (p, \iota_0) \xrightarrow{FS} \iota & (a, \iota) \xrightarrow{FS} \iota' &\triangleq FS(\iota)(a) = \iota' \\ (a/p, \iota) \xrightarrow{FS} \iota' &\triangleq \exists \iota''. FS(\iota)(a) = \iota'' \wedge (p, \iota'') \xrightarrow{FS} \iota' \end{aligned}$$

With this predicate we can now define the sets of file systems that correspond to the lock being unlocked and locked respectively as follows:

$$\begin{aligned} \mathcal{ULK}(p/a) &\triangleq \left\{ FS \mid \exists \iota. p \xrightarrow{FS} \iota \wedge \text{isdir}(FS(\iota)) \wedge a \notin FS(\iota) \right\} \\ \mathcal{LK}(p/a) &\triangleq \left\{ FS \mid \exists \iota. p \xrightarrow{FS} \iota \wedge \text{isdir}(FS(\iota)) \wedge a \in FS(\iota) \right\} \end{aligned}$$

Additionally, we define the union of the above sets: $\mathcal{LF}(p/a) \triangleq \mathcal{ULK}(p/a) \cup \mathcal{LK}(p/a)$, and the following predicates that describe the updates from FS to FS' that create and remove the lock file in its directory respectively:

$$\begin{aligned} \text{lk}(FS, FS', p/a) &\triangleq \exists \iota, \iota'. p \xrightarrow{FS} \iota \wedge FS' = FS[\iota \mapsto FS(\iota)[a \mapsto \iota']] [\iota' \mapsto \epsilon] \\ \text{ulk}(FS, FS', p/a) &\triangleq \exists \iota. p \xrightarrow{FS} \iota \wedge FS' = FS[\iota \mapsto FS(\iota) \setminus \{a\}] \end{aligned}$$

Now we define the interpretation to the **Lock** region as follows:

$$I(\mathbf{Lock}_\alpha(lf, 0)) \triangleq \exists FS \in \mathcal{ULK}(lf). \text{fs}(FS) \quad I(\mathbf{Lock}_\alpha(lf, 1)) \triangleq \exists FS \in \mathcal{LK}(lf). \text{fs}(FS)$$

We can now proceed with the refinement by applying the AUSEATOMIC law. For simplicity, let us consider the refinement of **unlock**:

$$\begin{aligned} &\langle \exists FS \in \mathcal{LK}(lf). \text{fs}(FS) * [G]_\alpha, \exists FS \in \mathcal{ULK}(lf). \text{fs}(FS) * [G]_\alpha \rangle \\ &\sqsubseteq \langle \mathbf{Lock}_\alpha(lf, 1) * [G]_\alpha, \mathbf{Lock}_\alpha(lf, 0) * [G]_\alpha \rangle \end{aligned}$$

Now we can work to refine this atomic update on the file-system state to the specification program of **unlink** that we defined in section 3. A proof sketch can be seen in figure 5. First we can frame-off the guard resource as it is no longer required by using the AFRAME refinement law which is directly analogous to the frame rule of separation logics:

$$\begin{aligned} &\langle \exists FS \in \mathcal{LK}(lf). \text{fs}(FS), \exists FS \in \mathcal{ULK}(lf). \text{fs}(FS) \rangle \\ &\sqsubseteq \langle \exists FS \in \mathcal{LK}(lf). \text{fs}(FS) * [G]_\alpha, \exists FS \in \mathcal{ULK}(lf). \text{fs}(FS) * [G]_\alpha \rangle \end{aligned}$$

Next, we strengthen the post-condition and eliminate the existential quantification over file-system graphs:

$$\begin{aligned} &\forall FS \in \mathcal{LK}(lf). \langle \text{fs}(FS), \exists FS'. \text{fs}(FS') \wedge \text{ulk}(FS, FS', lf) \rangle \\ &\sqsubseteq \langle \exists FS \in \mathcal{LK}(lf). \text{fs}(FS), \exists FS \in \mathcal{LK}(lf), FS'. \text{fs}(FS') \wedge \text{ulk}(FS, FS', lf) \rangle \\ &\sqsubseteq \langle \exists FS \in \mathcal{LK}(lf). \text{fs}(FS), \exists FS \in \mathcal{ULK}(lf). \text{fs}(FS) \rangle \end{aligned}$$

Our next challenge is that **unlinkSpec** consists of several steps in sequence. Earlier we introduced the ASTUTTER refinement law, however, **unlinkSpec** requires the manipulation of hidden, intermediary, non-atomic state about the existence of the lock file being manipulated. To deal with this, we introduce a generalisation of the ASTUTTER rule, HSTUTTER,

which chains together the non-atomic preconditions and postconditions as in the sequential composition of Hoare triples:

$$\begin{aligned} & \forall \vec{x}. \{P'\} \langle P(\vec{x}), P(\vec{x}) \rangle \{P''\}; \forall \vec{x}. \exists \vec{y}. \{P''\} \langle P(\vec{x}), Q(\vec{x}, \vec{y}) \rangle \{Q'(\vec{x}, \vec{y})\} \\ & \sqsubseteq \forall \vec{x}. \exists \vec{y}. \{P'\} \langle P(\vec{x}), Q(\vec{x}, \vec{y}) \rangle \{Q'(\vec{x}, \vec{y})\} \end{aligned}$$

We will also need the **HSTRENGTHEN** refinement law which allows us to refine a non-atomic update to a part of the state to an atomic update:

$$\begin{aligned} & \forall \vec{x}. \exists \vec{y}. \{P'\} \langle P' * P(\vec{x}), Q(\vec{x}, \vec{y}) * Q'(\vec{x}, \vec{y}) \rangle \{Q'(\vec{x}, \vec{y})\} \\ & \sqsubseteq \forall \vec{x}. \exists \vec{y}. \{P' * P'\} \langle P(\vec{x}), Q(\vec{x}, \vec{y}) \rangle \{Q'(\vec{x}, \vec{y}) * Q'(\vec{x}, \vec{y})\} \end{aligned}$$

The **HSTRENGTHEN** refinement law will be used to move stable information about the file system state, i.e. assertions that cannot be invalidated by the environment, into the non-atomic assertions so that they can be used to reason about the behaviour of subsequent steps of the program.

To proceed with our refinement to **unlinkSpec**, we first use the fact that $\forall \vec{x}. \langle P, Q \rangle \equiv \forall \vec{x}. \{\text{true}\} \langle P, Q \rangle \{\text{true}\}$ and the **HSTUTTER** refinement law to further refine the current specification:

$$\begin{aligned} & \forall FS \in \mathcal{LK}(lf). \{\text{true}\} \langle \text{fs}(FS), \text{fs}(FS) \rangle \left\{ p \xrightarrow{FS} r \wedge \text{isdir}(FS(r)) \wedge a \in FS(r) \right\}; \\ & \forall FS \in \mathcal{LK}(lf). \left\{ p \xrightarrow{FS} r \wedge \text{isdir}(FS(r)) \wedge a \in FS(r) \right\} \langle \text{fs}(FS), \exists FS'. \text{fs}(FS') \wedge \text{ulk}(FS, FS', lf) \rangle \{\text{true}\} \\ & \sqsubseteq \forall FS \in \mathcal{LK}(lf). \langle \text{fs}(FS), \exists FS'. \text{fs}(FS') \wedge \text{ulk}(FS, FS', lf) \rangle \end{aligned}$$

Where variables p , a and r correspond to the path prefix, the last name in the path and the inode corresponding to the directory p respectively. The first of these two specifications is then refined using the **HSTRENGTHEN** refinement law to move all of the content of the non-atomic postcondition into the atomic postcondition. This refinement is valid as the environment is restricted to maintaining the existence of the path lf , the recursive calls to resolve can be thought of as a single atomic step since the result of the individual atomic resolution steps will not be invalidated by the environment.

This specification is further refined to the first three lines of code of **unlinkSpec**. In this specification, we substitute lf for $path$ due to the function call. For the derivation of the recursive **resolve** function we rely on standard fixpoint induction law:

$$\text{IND} \frac{\lambda x. \phi [\psi/A] \sqsubseteq \lambda x. \psi}{\mu A. \lambda x. \phi \sqsubseteq \lambda x. \psi}$$

We omit the full derivation for brevity.

$$\begin{aligned} & \text{let } p = \text{dirname}(path); \text{let } a = \text{basename}(path); \\ & \text{let } r = \text{resolve}(p, \iota_0) \\ & \sqsubseteq \forall FS \in \mathcal{LK}(path). \langle \text{fs}(FS), \text{fs}(FS) \wedge p \xrightarrow{FS} r \wedge \text{isdir}(FS(r)) \wedge a \in FS(r) \rangle \end{aligned}$$

Next, we must verify that:

$$\begin{aligned} & \text{if } \neg \text{iserr}(r) \text{ then} \\ & \quad \text{return link_delete}(r, a) \\ & \quad \sqcup \text{link_delete_notdir}(r, a) \\ & \text{else return } r \text{ fi} \\ & \sqsubseteq \forall FS \in \mathcal{LK}(path). \left\{ p \xrightarrow{FS} r \wedge \text{isdir}(FS(r)) \wedge a \in FS(r) \right\} \\ & \quad \langle \text{fs}(FS), \exists FS'. \text{fs}(FS') \wedge \text{ulk}(FS, FS', lf) \rangle \\ & \quad \{\text{true}\} \end{aligned}$$

When verifying that an if statement refines an atomic specification, it suffices to verify that both branches of the if statement satisfy the atomic specification given that the precondition is extended with the if statement's condition and its negation for each branch respectively, as is done in figure 5.

First however, before applying this rule, figure 5. applies the **HSTRENGTHEN** refinement law to move the stable information about the file system in the non-atomic precondition back into the atomic precondition.

We can now check that the false branch of the if refines:

$$\forall FS \in \mathcal{LK}(\text{path}). \left\langle \begin{array}{l} \text{fs}(FS) \wedge p \xrightarrow{FS} r \wedge \text{isdir}(FS(r)) \wedge a \in FS(r) \wedge \text{iserr}(r), \\ \exists FS'. \text{fs}(FS') \wedge \text{ulk}(FS, FS', lf) \end{array} \right\rangle$$

As $\text{iserr}(r) \Rightarrow r \in \text{ERRS}$ and $p \xrightarrow{FS} r \Rightarrow r \in \text{INODES}$, and since $\text{INODES} \cap \text{ERRS} = \emptyset$, $p \xrightarrow{FS} r \wedge \text{iserr}(r) \Rightarrow \text{false}$ holds. Since $\text{false} \Rightarrow \exists FS'. \text{fs}(FS') \wedge \text{ulk}(FS, FS', lf)$, by using the atomic consequence rule, figure 5 reaches the goal for this branch of the if statement.

To finish showing that **unlinkSpec** refines the goal specification, it remains to check the true branch of the if statement. First we apply the consequence and frame rule, to frame away $p \xrightarrow{FS} r \wedge a \in FS(r) \wedge FS \in \mathcal{LK}(\text{path})$, on the specification for the true branch of the if statement:

$$\forall FS. \langle \text{fs}(FS) \wedge \text{isdir}(FS(r)), a \in FS(r) \Rightarrow \text{fs}(FS[r \mapsto FS(r) \setminus \{a\}]) * \text{ret} = 0 \rangle \sqsubseteq \forall FS \in \mathcal{LK}(\text{path}). \left\langle \begin{array}{l} \text{fs}(FS) \wedge p \xrightarrow{FS} r \wedge \text{isdir}(FS(r)) \wedge a \in FS(r) \wedge \neg \text{iserr}(r), \\ \exists FS'. \text{fs}(FS') \wedge \text{ulk}(FS, FS', lf) \end{array} \right\rangle$$

Using the specification of **link_delete()** and the **DCHOICEINTRO** refinement law, $\phi \sqcap \psi \sqsubseteq \phi$, we can show that the current specification is refined by **link_delete()**.

Finally, the **ABSORB** law, $\phi \sqcup (\phi \sqcap \psi) \equiv \phi \equiv \phi \sqcap (\phi \sqcup \psi)$, asserts that a specification made up of an angelic choice between a specification, ϕ , and a second, strictly less permissive one, $\phi \sqcap \psi$, is equivalent to ϕ , as in both cases, it must be satisfied. This law can be used in conjunction with the definition of **link_delete_notdir**(r, a) to show that **return link_delete**(r, a) \sqcup **link_delete_notdir**(r, a) \sqsubseteq **return link_delete**(r, a)⁴, which completes the proof.

This proof encapsulates the entirety of the file system within the **Lock** shared region, which effectively prohibits sharing of the file system via means other than the lock-file module's interface. This assumption is not valid in general as the file system is a public namespace that can be accessed and modified by concurrently executing applications. In section 6, we will extend this reasoning to be able to express the necessary restrictions on the context in which the program executes.

5 TADA-Refine Specification Language and Refinement Calculus

We describe TADA-Refine, our concurrent specification language and associated refinement calculus, giving just a sketch here and providing full details in the technical report [29]. We discovered that, due to the stuttering and mumbling laws, we have simpler laws and soundness proof compared with those of TaDA.

⁴ **link_delete_notdir**(r, a) can be rewritten as **link_delete**(r, a) \sqcap **return err_nodir_links**(ι, a).

Specifications	$\phi, \psi ::= \phi; \psi \mid \phi \parallel \psi \mid \mathbf{let} f = F \mathbf{in} \phi \mid Fe$ $\mid \phi \sqcup \psi \mid \phi \sqcap \psi \mid \exists x. \phi \mid \forall \vec{x}. \langle P, Q \rangle_k$
Functions	$F ::= f \mid A \mid \mu A. \lambda x. \phi \mid \lambda x. \phi$
Expressions	$e ::= \dots$
Assertions	$P, Q, R ::= \mathbf{false} \mid \mathbf{true} \mid P * Q \mid P \wedge Q \mid P \vee Q \mid \exists x. P \mid \forall x. P \mid P \Rightarrow Q$ $\mid \mathbf{t}_\alpha^k(\vec{y}, x) \mid [\mathbf{G}]_\alpha \mid \dots$

where x denotes a variable, \vec{x} a sequence of variables, A a recursive variable and f a function.

■ **Figure 6** The specification language of TaDA-Refine: specifications and assertions.

5.1 The Specification Language

The syntax of the specification language for TADA-Refine is given in figure 6, using the grammars of the specifications and assertions. Following Turon and Wand [36], we do not distinguish between specifications and concrete programs, taking the view that programs are the most concrete of specifications. The specifications are built from traditional programming constructs: sequential composition $\phi; \psi$, parallel composition $\phi \parallel \psi$, recursion and first-order procedures. We use additional constructs to express specification non-determinism: angelic choice, $\phi \sqcup \psi$, behaves either as ϕ or as ψ ; and demonic choice, $\phi \sqcap \psi$, behaves as ϕ and ψ . Angelic and demonic non-determinism are motivated in the `unlink` specification of §3.1. Existential quantification, $\exists x. \phi$, behaves as ϕ for some choice of x .

The atomic specification statement, $\forall \vec{x}. \langle P, Q \rangle_k$, is motivated in §3.1. It specifies any operation that atomically updates a state satisfying the precondition P to a state satisfying the postcondition Q . The universal quantifier binds \vec{x} to both the precondition and postcondition, declaring that the update is atomic for all possible values of \vec{x} . The statement includes a *region level* subscript, k , explained below. The assertions, P and Q , are based on the intuitionistic assertions of TaDA [9]. They are built from the standard connectives from first-order logic and intuitionistic separation logic [31], together with shared region assertions and guard assertions of TaDA and first-order recursive predicates. In addition, we will use a collection of abstract and pure predicates signified by the \dots and introduced by example in the other sections. We implicitly require that the pre- and postconditions are *stable*: they must account for interference from other threads. The region assertion, $\mathbf{t}_\alpha^k(\vec{y}, x)$, asserts the existence of a region with superscript k with identifier α of type \mathbf{t} and parameters \vec{y} and abstract state x . The guard assertion, $[\mathbf{G}]_\alpha$, asserts the ownership of guard \mathbf{G} for region α . As described in §4, associated with a region type is a guard separation algebra, a labelled transition system and an interpretation function.

A region assertion also has a region-level superscript, k , analogous to the region-level subscript of a specification statement. The region level is an integer, signifying that only regions below level k may be replaced by their interpretation (opened) in the refinement of a specification statement. Their purpose is to ensure that we cannot open the same region twice during a refinement derivation, as this could unsoundly duplicate resources encapsulated by the region.

We keep the specification language minimal. For simplicity and to keep specifications declarative, variables are immutable. Additional programming constructs used in the specifications given throughout this paper can be easily encoded. For example, the specification

$$\begin{array}{c}
\text{AEELIM} \\
\forall \vec{y}. x. \langle P, Q \rangle_k \sqsubseteq \forall \vec{y}. \langle \exists x. P, \exists x. Q \rangle_k \\
\\
\text{AFRAME} \\
\forall \vec{x}. \langle P, Q \rangle_k \sqsubseteq \forall \vec{x}. \langle P * R, Q * R \rangle_k \\
\\
\text{AUSEATOMIC} \\
\frac{\forall x. (x, f(x)) \in \mathcal{T}_t(G)^*}{\forall x, \vec{x}. \langle I(\mathbf{t}_\alpha^k(\vec{e}, x)) * P(\vec{x}) * [\mathbf{G}]_\alpha, I(\mathbf{t}_\alpha^k(\vec{e}, f(x))) * Q(\vec{x}) \rangle_k} \\
\sqsubseteq \forall x, \vec{x}. \langle \mathbf{t}_\alpha^k(\vec{e}, x) * P(\vec{x}) * [\mathbf{G}]_\alpha, \mathbf{t}_\alpha^k(\vec{e}, f(x)) * Q(\vec{x}) \rangle_{k+1} \\
\\
\text{ASTUTTER} \\
\forall \vec{x}. \langle P, P \rangle_k; \forall \vec{x}. \langle P, Q \rangle_k \sqsubseteq \forall \vec{x}. \langle P, Q \rangle_k \\
\\
\text{AMUMBLE} \\
\forall \vec{x}. \langle P, Q \rangle_k \sqsubseteq \forall \vec{x}. \langle P, P' \rangle_k; \forall \vec{x}. \langle P', Q \rangle_k \\
\\
\text{ACONS} \\
\frac{\forall \vec{x}. P \preceq P' \quad \forall \vec{x}. Q' \preceq Q}{\forall \vec{x}. \langle P', Q' \rangle_k \sqsubseteq \forall \vec{x}. \langle P, Q \rangle_k} \\
\\
\text{ARLEVEL} \\
\frac{k_1 \leq k_2}{\forall \vec{x}. \langle P, Q \rangle_{k_1} \sqsubseteq \forall \vec{x}. \langle P, Q \rangle_{k_2}}
\end{array}$$

■ **Figure 7** Some refinement laws for the atomic specifications.

let $x = F(e)$ **in** ϕ can be written as $\exists x. F(e, x); \phi$ which binds the return variable **ret** to x . Control flow can be encoded with angelic choice. For example, **if** P **then** ϕ **else** ψ **fi** can be written as $(\langle \text{true}, P \rangle; \phi) \sqcup (\langle \text{true}, \neg P \rangle; \psi)$. Encodings for the other syntactic features used in our specifications are given in the technical report [29]. In §5.3, we discuss the encoding of hybrid specification statements.

In the technical report [29], we also define the operational semantics for our specification language as a transition relation, $\phi, h \Downarrow o$, from specifications ϕ and concrete heaps h to outcomes $o ::= h \mid \zeta$, where ζ denotes a fault.

5.2 The Refinement Calculus

The refinement calculus for TaDA-Refine comprises standard laws of refinement, refinement laws for atomic specification statements adapted from [36], and laws associated with TaDA's program-logic rules. Unlike TaDA, where stuttering and mumbling is hidden in its underlying semantics, the stuttering and mumbling laws are first-class citizens in TaDA-Refine. This enables us to simplify significantly the laws associated with the TaDA rules and the proof of adequacy (Theorem 1). The full calculus is given in the technical report [29].

In figure 7, we provide a selection of refinement laws for atomic specification statements. The AEELIM is analogous to the existential elimination rule of Hoare logics. The ACONS law is a semantic consequence law, originating from the views framework [10]. It generalises the standard logical consequence relation, using a view-shift relation \preceq adapted from TaDA instead of the usual logical implication. The AFRAME law is a frame law for atomic statements, originating from Turon and Wand's work [36]. The AUSEATOMIC and ARLEVEL laws are taken from analogous rules of TaDA. AUSEATOMIC allows us to refine an atomic update on the state of a shared region into an atomic update on the region's interpretation given by the interpretation function I . Note that by doing so the region level associated with the specification statement is decremented. This prevents the same region to be re-opened again in subsequent refinements. Unlike TaDA, we do not require other laws for shared regions due to the presence of the stuttering and mumbling laws as first-class citizens. Stuttering and mumbling originate from the work on trace semantics by Brookes [5]. ASTUTTER allows us to refine a single atomic update to a sequence of atomic steps, as long as the state before the update is maintained. AMUMBLE allows us to refine a sequence of atomic steps into a

single atomic update, as long as the environment does not invalidate the intermediate states. Note that, by combining the two laws, we can derive a stuttering equivalence.

The laws of the refinement calculus are sound with respect to a denotational refinement relation which is adequate with respect to an operational refinement relation. We define the denotational refinement relation using a denotational semantics of specifications, denoted $\llbracket \phi \rrbracket^\rho$ with variable context ρ , which is defined as sets of observed traces. This gives us a denotational refinement relation defined as the subset relation between traces. Our adequacy proof follows the methodology of Turon and Wand [36], significantly adapted to handle TaDA assertions [9]; the details of adequacy and soundness are in the technical report [29].

► **Definition 1** (Denotational Refinement). $\phi \sqsubseteq \psi \iff \llbracket \phi \rrbracket^\rho \subseteq \llbracket \psi \rrbracket^\rho$.

We define the operational refinement relation as a partial-correctness contextual refinement, using our operational semantics given in the technical report [29]:

$$\phi \sqsubseteq_{\text{op}} \psi \iff \forall C, h. \begin{cases} C[\phi], h \Downarrow \xi \Rightarrow C[\psi], h \Downarrow \xi \\ C[\phi], h \Downarrow h' \Rightarrow C[\psi], h \Downarrow h' \vee C[\psi], h \Downarrow \xi \end{cases}$$

where C is a specification context. If the specification ψ faults, then ϕ is allowed to do anything since a fault is treated as *unspecified* behaviour.

► **Theorem 2** (Adequacy). *If $\phi \sqsubseteq \psi$, then $\phi \sqsubseteq_{\text{op}} \psi$*

5.3 Derived Hybrid Specification Statement

The derived hybrid atomic statement, $\forall \vec{x}. \exists \vec{y}. \{P'\} \langle P(\vec{x}), Q(\vec{x}, \vec{y}) \rangle \{Q'(\vec{x}, \vec{y})\}_k$, extends the atomic specification statement with non-atomic components: the atomic specification statement $P(\vec{x})$ is atomically updated to $Q(\vec{x}, \vec{y})$ for all values of \vec{x} ; at the same time, the non-atomic precondition P' is updated to $Q'(\vec{x}, \vec{y})$ without any atomicity guarantees. The quantification extends to the end of the statement and is a little subtle. The non-atomic precondition is independent of the universally quantified \vec{x} because the environment may be modifying it before the atomic update takes effect. The two postconditions are linked by the existentially quantified \vec{y} , non-deterministically chosen by the implementation at the point the atomic update takes effect.

The hybrid specification statement is a derived construct, defined as a specification program comprising a sequence of atomic specification statements.

► **Definition 3** (Hybrid Specification Statement). The *hybrid specification statement* is defined by:

$$\forall \vec{x}. \exists \vec{y}. \{P'\} \langle P(\vec{x}), Q(\vec{x}, \vec{y}) \rangle \{Q'(\vec{x}, \vec{y})\}_k \triangleq \begin{pmatrix} \exists p. \forall \vec{x}. \langle P' * P(\vec{x}), P' \wedge p * P(\vec{x}) \rangle_k; \\ \left(\begin{array}{l} \mu A. \lambda p. \exists p'. \forall \vec{x}. \langle p * P(\vec{x}), p' * P(\vec{x}) \rangle_k; Ap' \\ \sqcup \exists \vec{x}, \vec{y}. \exists p''. \langle p * P(\vec{x}), p'' * Q(\vec{x}, \vec{y}) \rangle_k; \\ \left(\begin{array}{l} \mu B. \lambda p''. \exists p'''. \langle p'', p''' \rangle_k; Bp''' \\ \sqcup \langle p'', Q'(\vec{x}, \vec{y}) \rangle_k \end{array} \right) p'' \end{array} \right) p'' \end{pmatrix}^p$$

The first atomic specification statement solely serves to capture the states satisfied by the non-atomic precondition P' into the variable p , so that it can be passed as an argument to the subsequent recursive function. It is a silent atomic step: since it does not change the state, the first atomic specification statement is not observable by ASTUTTER. The recursive function that follows consists of two branches that are non-deterministically chosen using

$$\begin{array}{l}
\text{HMUMBLE} \\
\forall \vec{x}. \exists \vec{y}. \{P'\} \langle P(\vec{x}), Q(\vec{x}, \vec{y}) \rangle \{Q'(\vec{x}, \vec{y})\}_k \sqsubseteq \\
\forall \vec{x}. \exists \vec{y}. \{P'\} \langle P(\vec{x}), P'(\vec{x}) \rangle \{P''\}_k; \forall \vec{x}. \exists \vec{y}. \{P''\} \langle P'(\vec{x}), Q(\vec{x}, \vec{y}) \rangle \{Q'(\vec{x}, \vec{y})\}_k \\
\\
\text{HSTUTTER} \\
\forall \vec{x}. \{P'\} \langle P(\vec{x}), P(\vec{x}) \rangle \{P''\}_k; \forall \vec{x}. \exists \vec{y}. \{P''\} \langle P(\vec{x}), Q(\vec{x}, \vec{y}) \rangle \{Q'(\vec{x}, \vec{y})\}_k \sqsubseteq \\
\forall \vec{x}. \exists \vec{y}. \{P'\} \langle P(\vec{x}), Q(\vec{x}, \vec{y}) \rangle \{Q'(\vec{x}, \vec{y})\}_k \\
\\
\text{HSTRENGTHEN} \\
\forall \vec{x}. \exists \vec{y}. \{P'\} \langle P' * P(\vec{x}), Q(\vec{x}, \vec{y}) * Q'(\vec{x}, \vec{y}) \rangle \{Q'(\vec{x}, \vec{y})\}_k \sqsubseteq \\
\forall \vec{x}. \exists \vec{y}. \{P' * P'\} \langle P(\vec{x}), Q(\vec{x}, \vec{y}) \rangle \{Q'(\vec{x}, \vec{y}) * Q'(\vec{x}, \vec{y})\}_k
\end{array}$$

■ **Figure 8** Select hybrid specification statement refinement laws.

angelic choice. Note that these branches operate on both the non-atomically updated state captured by the logical variables p, p', \dots and the atomically updated state specified by the P and Q assertions. The first branch updates the non-atomic state p , while maintaining the atomic precondition $P(\vec{x})$ for all \vec{x} , and then recursively continues with the resulting non-atomic state p' . Effectively, this specifies that while P' is being modified in multiple steps the concurrent environment may change \vec{x} as long as it maintains $P(\vec{x})$. The second branch of the angelic choice terminates the recursion by performing the atomic update from $P(\vec{x})$ to $Q(\vec{x}, \vec{y})$, for some \vec{x} and \vec{y} (determined by the the atomic update takes affect). The same update may also update the non-atomic part of the state. After this point, the non-atomic part of the state can continue to be updated until we reach a state satisfying $Q'(\vec{x}, \vec{y})$; the atomic part cannot be further updated by the thread, and the environment need not be constrained, as the atomic step has been done by the thread.

We can derive refinement laws for hybrid specification statements which are analogous those given for the atomic specification statements. Most are straightforward extensions accounting for the non-atomic state component. We focus on the most interesting cases in figure 8. The HSTRENGTHEN law allows us to refine part of the non-atomic component update into an atomic update, essentially making the whole operation “more” atomic. The HMUMBLE law simply extends AMUMBLE to the hybrid case. However, consider HSTUTTER. For the atomic component it acts in the same as ASTUTTER, whereas for the non-atomic component it acts as the sequencing rule of Hoare logics. In fact, a property of these derived hybrid laws is when the atomic pre- and postconditions are **true** then the hybrid refinement laws correspond to standard Hoare rules, and when the non-atomic pre- and postconditions are **true** then the hybrid refinement laws correspond to the laws of atomic specification statements.

6 TaDA-Refine Client Reasoning II: Context Invariants

In section 4, we introduced the key elements of our client reasoning by examining a simple lock file module and proving a refinement between the abstract specification and implementation for the `unlock` operation. However, it encapsulates the entire file system within the **Lock** shared region and thus also the abstract predicate `Lock`. This prevents us from using the abstract specification to reason about clients that make further use of the file system.

We are unable to abstract the details of how a module’s implementation shares the file system and maintain compositionality at the same time. This is due to the nature of POSIX file systems. In POSIX all possible file system paths are usable by everyone at all times,

even if they do not exist. The concept of a path being private to an application simply does not exist ⁵. In other words the file system is a *public namespace*. In contrast, the traditional heap memory is a private namespace. Heap addresses are usable only when allocated. When an address is first allocated, it is only known to the allocating thread and thus we can programmatically control how they are shared with other threads. Dereferencing an unallocated heap address is undefined behaviour, typically resulting in a crash.

Effectively, in POSIX file systems all locations are shared, with everyone. Therefore, in §4, by fully encapsulating the file system state in the **Lock** shared region, we restrict all file system access to the lock-file module. This is too restrictive; we cannot reason about the module's use in contexts that also use the file system. The solution is to place restrictions on the context itself.

In the case of the lock-file module, we require that the context keeps the sub-graph formed by the path to the lock-file directory unmodified, and that the only way to create or remove the lock file is via the module's operations. Otherwise, the context could interfere with the resolution of the path, rendering it unresolvable or diverting the resolution to a different location. The lock-file module cannot enforce such restrictions on its own. Instead, these restrictions form a proof obligation for the context. We express such proof obligations with *context invariants*. For our lock-file module, **LFctx** denotes the context invariant under which its specification holds.

In order to define **LFctx**, we first encapsulate the file system within the global file-system shared region of type **GFS**. There is only a single instance of this region with a known identifier which we keep implicit. All clients accessing the file system do so via this region. The region's state is a file-system graph, $FS \in \mathcal{FS}$, with the straightforward interpretation: $I(\mathbf{GFS}(FS)) \triangleq \text{fs}(FS)$. The guards and labelled transition system of this region, are defined by the context. However, we define **LFctx** to place restrictions on the guards and transition system.

To aid our definitions, we introduce some notation: $\mathcal{G}_{\mathbf{t}}$ denotes the set of guards associated with the region type \mathbf{t} ; $\mathbf{G} \bullet \mathbf{G}'$ denotes the partial, associative and commutative composition of guards; $\mathbf{G} \# \mathbf{G}'$ states that the composition of guards \mathbf{G} and \mathbf{G}' is defined; and $\mathcal{T}_{\mathbf{t}}(\mathbf{G})^*$ denotes the transitions for guard \mathbf{G} of the region type \mathbf{t} , where the superscript $*$ denotes the reflexive-transitive closure. We also define the following auxiliary predicates:

$$\begin{aligned} !\mathbf{G} \in \mathcal{G}_{\mathbf{t}} &\triangleq \mathbf{G} \in \mathcal{G}_{\mathbf{t}} \wedge \mathbf{G} \bullet \mathbf{G} \text{ undefined} \\ (x, y) \dagger_{\mathbf{t}} \mathbf{G} &\triangleq (x, y) \in \mathcal{T}_{\mathbf{t}}(\mathbf{G})^* \wedge \forall \mathbf{G}' \in \mathcal{G}_{\mathbf{t}}. \mathbf{G}' \# \mathbf{G} \Rightarrow (x, y) \notin \mathcal{T}_{\mathbf{t}}(\mathbf{G}')^* \end{aligned}$$

The predicate $!\mathbf{G} \in \mathcal{G}_{\mathbf{t}}$ states that there is only one instance of the guard \mathbf{G} of the region type \mathbf{t} , and $(x, y) \dagger_{\mathbf{t}} \mathbf{G}$ states that in regions of type \mathbf{t} , the transition from state x to y is defined only for \mathbf{G} . Additionally, we define the expression $FS \downarrow_p$ to identify the sub-graph of FS that is formed by the path p as follows:

$$\begin{aligned} FS \downarrow_p &\triangleq FS \downarrow_p^{\iota_0} & FS \downarrow_a^{\iota} &\triangleq \iota \mapsto (a \mapsto FS(\iota)(a)) \\ FS \downarrow_{a/p}^{\iota} &\triangleq \iota \mapsto (a \mapsto FS(\iota)(a)) \uplus FS \downarrow_p^{FS(\iota)(a)} \end{aligned}$$

⁵ File-access permissions restrict access to only the processes with the same privileges.

We can now define the context invariant as follows:

$$\begin{aligned} \text{LFCtx}(p/a) \triangleq & \\ & \exists FS \in \mathcal{LF}(p/a). \mathbf{GFS}(FS) \wedge ![\text{LF}(p/a)] \in \mathcal{G}_{\mathbf{GFS}} \\ & \wedge \forall FS \in \mathcal{ULK}(p/a). \exists FS'. \text{lk}(FS, FS', p/a) \wedge (FS, FS') \dagger_{\mathbf{GFS}} \text{LF}(p/a) \\ & \wedge \forall FS \in \mathcal{LK}(p/a). \exists FS'. \text{ulk}(FS, FS', p/a) \wedge (FS, FS') \dagger_{\mathbf{GFS}} \text{LF}(p/a) \\ & \wedge \forall G \in \mathcal{G}_{\mathbf{GFS}}. \forall FS, FS' \in \mathcal{LK}(lf). (FS, FS') \in \mathcal{T}_{\mathbf{GFS}}(G)^* \Rightarrow FS \upharpoonright_p = FS' \upharpoonright_p \end{aligned}$$

The first line of the definition restricts the states of the global file-system region to those in which the path to the lock-file directory exists and the lock file itself may exist or not. Additionally, it requires the indivisible guard $\text{LF}(p/a)$ to be defined for the global file-system region. The second and third lines of the definition state that transitions creating or removing the lock file in its directory are only defined for the guard $\text{LF}(p/a)$. Therefore, only ownership of this guard grants a thread the capability to transition between locked and unlocked states. Finally, the last line of the definition requires all transitions between lock-file states to maintain the same file-system sub-graph for the path p . This guarantees that the context does not concurrently modify the sub-graph such that the path resolution is diverted to a different location.

Assuming the context satisfies $\text{LFCtx}(lf)$, we can now redefine the interpretation of the **Lock** region as:

$$\begin{aligned} I(\mathbf{Lock}_\alpha(lf, 0)) &\triangleq \exists FS \in \mathcal{ULK}(lf). \mathbf{GFS}(FS) * [\text{LF}(lf)] \\ I(\mathbf{Lock}_\alpha(lf, 1)) &\triangleq \exists FS \in \mathcal{LK}(lf). \mathbf{GFS}(FS) * [\text{LF}(lf)] \end{aligned}$$

Instead of the **Lock** fully encapsulating the file system itself in section 4, we now encapsulate only the possible ways in which the file system is shared with everyone else by means of the global file system region.

The global file system shared region is an abstraction breaker. All modules accessing the file system use it. Therefore, all file-system module specifications are effectively parametric to its definition. Thus we amend the original lock-file specification of section 4 accordingly:

$$\begin{aligned} \text{LFCtx}(lf) \vdash \text{lock}(lf) &\sqsubseteq \forall v \in \{0, 1\}. \langle \text{Lock}(s, lf, v), \text{Lock}(s, lf, 1) * v = 0 \rangle \\ \text{LFCtx}(lf) \vdash \text{unlock}(lf) &\sqsubseteq \langle \text{Lock}(s, lf, 1), \text{Lock}(s, lf, 0) \rangle \end{aligned}$$

It remains to prove the refinement between the implementation and our specification. In figure 9 we give a sketch proof for the **lock** operation. Throughout the proof we assume that LFCtx holds. On the other hand, LFCtx is a proof obligation for the context. The main difference from the proof of **unlock** in § 4 is that we use **AUSEATOMIC** twice; first to refine the atomic update on the **Lock** predicate into an update on the global file system region **GFS** and then again to refine that update to an atomic update on the underlying file system in the refinement of **close** and **open**. The refinement of **open** proceeds similarly to **unlock** and is given in the technical report [29].

7 Related Work

There has been substantial work on the formal specification of key fragments of POSIX file systems, even leading to a verification challenge by Joshi and Holzmann [22]. Refinements from specifications to implementations have been widely studied [3, 19, 15, 16]. Of particular note are the specifications based on Z notation, and the use of refinement calculus to construct verified implementations [24, 15, 16]. Recently, specifications based on separation

$$\begin{array}{l}
\text{lock}(lf) \equiv \\
\begin{array}{l}
\text{let } fd = \text{open}(lf, 0_CREAT|0_EXCL) \\
\hline
\text{AUseAtomic} \\
\forall FS \in \mathcal{LF}(lf) \\
\text{GFS}(FS) * [\text{LF}(lf)], \\
\hline
\sqsubseteq \left\langle \left((\text{GFS}(FS) * fd = \text{EEXIST}) \vee (\exists FS' \in \mathcal{LK}(lf). \text{GFS}(FS') * \text{fd}(fd, -, -)) \right) * [\text{LF}(lf)] \right\rangle \\
\hline
\text{AEELIM} \\
\left\langle \begin{array}{l}
\exists FS \in \mathcal{ULK}(lf). \text{GFS}(FS) * [\text{LF}(lf)], \\
(\exists FS \in \mathcal{LK}(lf). \text{GFS}(FS) * [\text{LF}(lf)]) \vee \\
(\exists FS \in \mathcal{ULK}(lf). \text{GFS}(FS) * [\text{LF}(lf)] * fd = \text{EEXIST})
\end{array} \right\rangle \\
\text{if iserr}(fd) \text{ then} \\
\quad \text{lock}(lf) \\
\hline
\text{IND} \\
\sqsubseteq \langle \exists FS \in \mathcal{ULK}(lf). \text{GFS}(FS) * [\text{LF}(lf)], \exists FS \in \mathcal{LK}(lf). \text{GFS}(FS) * [\text{LF}(lf)] \rangle \\
\hline
\text{AFRAME} \\
\sqsubseteq \left\langle \begin{array}{l}
\exists FS \in \mathcal{ULK}(lf). \text{GFS}(FS) * [\text{LF}(lf)] * fd = \text{EEXIST}, \\
\exists FS \in \mathcal{LK}(lf). \text{GFS}(FS) * [\text{LF}(lf)] * fd = \text{EEXIST}
\end{array} \right\rangle \\
\text{else} \\
\quad \text{close}(fd) \sqsubseteq \langle \text{fd}(fd, -, -), \text{true} \rangle \\
\quad \sqsubseteq \langle \exists FS \in \mathcal{ULK}(lf). \text{GFS}(FS) * [\text{LF}(lf)], \exists FS \in \mathcal{LK}(lf). \text{GFS}(FS) * [\text{LF}(lf)] \rangle \\
\text{fi} \\
\sqsubseteq \langle \exists FS \in \mathcal{ULK}(lf). \text{GFS}(FS) * [\text{LF}(lf)], \exists FS \in \mathcal{LK}(lf). \text{GFS}(FS) * [\text{LF}(lf)] \rangle \\
\hline
\text{ACONS, AUseAtomic} \\
\sqsubseteq \forall v \in \{0, 1\}. \langle \text{Lock}_\alpha(lf, v) * [\text{G}]_\alpha, \text{Lock}_\alpha(lf, 1) * [\text{G}]_\alpha * v = 0 \rangle
\end{array}
\end{array}$$

■ **Figure 9** lock() specification proof sketch.

logic [31] have been introduced, focusing on scalable client reasoning [17, 30]. This work [17] demonstrates that first-order reasoning scales poorly when reasoning about file-system clients, hence the introduction of a specification based on separation logic. Taking inspiration from this work, we demonstrate scalable reasoning about clients of POSIX file-systems using TaDA-Refine. Separation logics have also been used to build a verified fault-tolerant file-system implementation in the Coq theorem prover [7] and to verify elements of the Linux Virtual Filesystem Switch (VFS) [12]. All the aforementioned works are on sequential fragments of POSIX and do not handle concurrency.

Fisher *et al.* develop Forest [14], a declarative DSL in Haskell for safe manipulation of file systems. Forest clients use the typing discipline to specify the file-system structures they need and file-system access preserves the application invariants identified by static types. This work is an attempt to bridge the gap between the untyped world of sequential file-systems and the strongly-typed world of programming languages.

Ridge *et al.* have developed a coarse-grained concurrent specification of a fragment of the POSIX file system, based an operational semantics [33] with adaptations in the semantics to capture real-world implementations. The specification is used as a test oracle in a substantial test suite which they generate for major real-world implementations. However, since their concurrent specification is coarse-grained, their tests can only expose sequential behaviour. We can derive such coarse-grained specifications from the specifications we give in this paper, by a trivial application of the AMUMBLE refinement law. Such coarse-grained specifications could be used to verify coarse-grained implementations. However, they are not suitable as a general POSIX specification for client reasoning, as the implicit assumption of additional synchronisation is too strong.

We have given a specification of the complex concurrent behaviour associated with POSIX file systems by introducing TaDA-Refine, a concurrent specification language based on TaDA assertions [9, 8] and an associated refinement calculus. Our approach is inspired by the work of Turon and Wand [36]. However, we do not adopt their notion of fenced refinement to reason about fine-grained concurrent data structures as its applicability is more limited than more recent mechanisms of expressing sharing protocols and capabilities found in concurrent program logics. More significantly, fenced refinement proofs carry the built-in assumption that the module's state can only ever be changed by the module's operations which is not appropriate for reasoning about file-system clients. For this purpose we adopt TaDA's assertions and introduce context invariants for client reasoning. Furthermore, we introduce the hybrid specification statement as useful derived construct for reasoning about combinations of atomic and non-atomic effects.

Recently, various concurrent separation logics have been introduced to support reasoning about atomic operations [20, 35, 34, 9, 23, 26]. However, the examples have generally been limited to those using operations comprising single atomic steps. In contrast, our work on specifying POSIX file systems requires operations comprising multiple atomic steps. With higher-order logics such as [35, 34, 23], it is possible to encode multi-atomic specifications as auxiliary code in the style of [20]. With logics based on histories such as [26], it should also be possible to support multi-atomic specifications, although it is unclear if this method is applicable to operations that have concurrent path traversals such as `link`. It is important to note that, for client reasoning, all these formalisms require additional constraints on the context analogous to our context invariants.

8 Conclusions & Future Work

We have developed TaDA-Refine, a concurrent specification language and an associated refinement calculus which is able to specify the complex concurrent behaviour of POSIX file systems. To the best of our knowledge, this is the first specification of file-system concurrency that captures the intended POSIX semantics. Here, we have verified the lock-file client module. In Ntzik's thesis [27], we have also verified an implementation of named pipes which regular file I/O and lock file, and the concurrent interaction between and email client and server that is sensitive to the multi-atomic nature of path resolution. This client verification is not straightforward due to the file system being a public namespace. We introduce specifications conditional on context invariants to restrict interference.

Our research on file-system specification and client verification is far from over. We believe we have formalised the established consensus of the concurrent behaviour of POSIX file systems. Our methodology is, however, flexible enough to explore other choices, if desired. We plan to extend the specification to larger fragments, for example, covering symbolic links and file-access permissions. These are orthogonal to POSIX file-system concurrency and should not affect our reasoning methodology presented here.

For this paper, we justify our specification by appealing to the standard and the community consensus regarding the atomicity of operations. In future, we will justify our specification with respect to implementations. We plan to systematically justify the specification against real-world implementations by generating tests and using the specification as a test oracle, similarly to the approach of Ridge *et al* [33]. Another approach, following our refinement laws, is to refine the specification to a fine-grained concurrent reference implementation. Both approaches will require a mechanised version of our POSIX specification. Additionally, we want to build on the works of Chen *et al.*[7] and Ntzik *et al.* [28] to extend our specifications with fault-tolerance guarantees. We also want to study Network File Systems (NFS), which exhibit concurrent behaviours that are not sequentially consistent.

References

- 1 The Austin Group Mailing List. <https://www.opengroup.org/austin/lists.html>. Accessed: September 30, 2016.
- 2 POSIX.1-2008, IEEE 1003.1-2008, The Open Group Base Specifications Issue 7. URL: <http://pubs.opengroup.org/onlinepubs/9699919799/>.
- 3 Konstantine Arkoudas, Karen Zee, Viktor Kuncak, and Martin Rinard. Verifying a file system implementation. In Jim Davies, Wolfram Schulte, and Mike Barnett, editors, *Formal Methods and Software Engineering*, volume 3308 of *Lecture Notes in Computer Science*, pages 373–390. Springer Berlin Heidelberg, 2004. doi:10.1007/978-3-540-30482-1_32.
- 4 Ralph-Johan Back and Joakim Wright. *Refinement calculus: a systematic introduction*. Springer Science & Business Media, 2012.
- 5 Stephen Brookes. Full abstraction for a shared-variable parallel language. *Information and Computation*, 127(2):145–163, 1996. doi:10.1006/inco.1996.0056.
- 6 C. Calcagno, P. W. O’Hearn, and H. Yang. Local action and abstract separation logic. In *22nd Annual IEEE Symposium on Logic in Computer Science (LICS 2007)*, pages 366–378, July 2007. doi:10.1109/LICS.2007.30.
- 7 Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nikolai Zeldovich. Using crash hoare logic for certifying the fscq file system. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP ’15*, pages 18–37, New York, NY, USA, 2015. ACM. doi:10.1145/2815400.2815402.
- 8 Pedro da Rocha Pinto. *Reasoning with Time and Data Abstractions*. PhD thesis, Imperial College London, 2016.
- 9 Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. Tada: A logic for time and data abstraction. In Richard Jones, editor, *ECOOP 2014 – Object-Oriented Programming*, volume 8586 of *Lecture Notes in Computer Science*, pages 207–231. Springer Berlin Heidelberg, 2014. doi:10.1007/978-3-662-44202-9_9.
- 10 Thomas Dinsdale-Young, Lars Birkedal, Philippa Gardner, Matthew Parkinson, and Hongseok Yang. Views: Compositional reasoning for concurrent programs. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL ’13*, pages 287–300, New York, NY, USA, 2013. ACM. doi:10.1145/2429069.2429104.
- 11 Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. Concurrent abstract predicates. In Theo D’Hondt, editor, *ECOOP 2010 – Object-Oriented Programming*, volume 6183 of *Lecture Notes in Computer Science*, pages 504–528. Springer Berlin Heidelberg, 2010. doi:10.1007/978-3-642-14107-2_24.
- 12 Gidon Ernst, Gerhard Schellhorn, Dominik Haneberg, Jörg Pfähler, and Wolfgang Reif. Verification of a virtual filesystem switch. In Ernie Cohen and Andrey Rybalchenko, editors, *Verified Software: Theories, Tools, Experiments*, volume 8164 of *Lecture Notes in Computer Science*, pages 242–261. Springer Berlin Heidelberg, 2014. doi:10.1007/978-3-642-54108-7_13.
- 13 Xinyu Feng. Local rely-guarantee reasoning. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL ’09*, pages 315–327, New York, NY, USA, 2009. ACM. doi:10.1145/1480881.1480922.
- 14 Kathleen Fisher, Nate Foster, David Walker, and Kenny Q. Zhu. Forest: A language and toolkit for programming with filestores. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming, ICFP ’11*, pages 292–306, New York, NY, USA, 2011. ACM. doi:10.1145/2034773.2034814.
- 15 L. Freitas, Zheng Fu, and J. Woock. Posix file store in z/eves: an experiment in the verified software repository. In *Engineering Complex Computer Systems, 2007. 12th IEEE International Conference on*, pages 3–14, July 2007. doi:10.1109/ICECCS.2007.36.

- 16 Leo Freitas, Jim Woodcock, and Andrew Butterfield. Posix and the verification grand challenge: A roadmap. *2014 19th International Conference on Engineering of Complex Computer Systems*, 0:153–162, 2008. doi:10.1109/ICECCS.2008.35.
- 17 Philippa Gardner, Gian Ntzik, and Adam Wright. Local reasoning for the posix file system. In Zhong Shao, editor, *Programming Languages and Systems*, volume 8410 of *Lecture Notes in Computer Science*, pages 169–188. Springer Berlin Heidelberg, 2014. doi:10.1007/978-3-642-54833-8_10.
- 18 Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990. doi:10.1145/78969.78972.
- 19 WimH. Hesselink and MuhammadIkram Lali. Formalizing a hierarchical file system. *Formal Aspects of Computing*, 24(1):27–44, 2012. doi:10.1007/s00165-010-0171-2.
- 20 Bart Jacobs and Frank Piessens. Expressive modular fine-grained concurrency specification. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 271–282, New York, NY, USA, 2011. ACM. doi:10.1145/1926385.1926417.
- 21 Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. *VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java*, pages 41–55. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. doi:10.1007/978-3-642-20398-5_4.
- 22 Rajeev Joshi and GerardJ. Holzmann. A mini challenge: build a verifiable filesystem. *Formal Aspects of Computing*, 19(2):269–272, 2007. doi:10.1007/s00165-006-0022-3.
- 23 Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 637–650, New York, NY, USA, 2015. ACM. doi:10.1145/2676726.2676980.
- 24 Carroll Morgan and Bernard Sufrin. Specification of the unix filing system. *Software Engineering, IEEE Transactions on*, SE-10(2):128–142, March 1984. doi:10.1109/TSE.1984.5010215.
- 25 Carroll Morgan and Trevor Vickers. *On the refinement calculus*. Springer Science & Business Media, 2012.
- 26 Aleksandar Nanevski, Ruy Ley-Wild, Ilya Sergey, and GermánAndrés Delbianco. Communicating state transition systems for fine-grained concurrent resources. In Zhong Shao, editor, *Programming Languages and Systems*, volume 8410 of *Lecture Notes in Computer Science*, pages 290–310. Springer Berlin Heidelberg, 2014. doi:10.1007/978-3-642-54833-8_16.
- 27 Gian Ntzik. *Reasoning About POSIX File Systems*. PhD thesis, Imperial College London, sep 2016.
- 28 Gian Ntzik, Pedro da Rocha Pinto, and Philippa Gardner. *Programming Languages and Systems: 13th Asian Symposium, APLAS 2015, Pohang, South Korea, November 30 - December 2, 2015, Proceedings*, chapter Fault-Tolerant Resource Reasoning, pages 169–188. Springer International Publishing, Cham, 2015. doi:10.1007/978-3-319-26529-2_10.
- 29 Gian Ntzik, Pedro da Rocha Pinto, Julian Sutherland, and Philippa Gardner. A concurrent specification of POSIX file systems (technical report). Technical Report 2018/3, Department of Computing, Imperial College London, 2018. URL: <https://www.doc.ic.ac.uk/research/technicalreports/2018/#3>.
- 30 Gian Ntzik and Philippa Gardner. Reasoning about the posix file system: Local update and global pathnames. In *Proceedings of the 2015 ACM SIGPLAN International Conference*

- on *Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 201–220, New York, NY, USA, 2015. ACM. doi:10.1145/2814270.2814306.
- 31 J.C. Reynolds. Separation logic: a logic for shared mutable data structures. In *Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on*, pages 55–74, 2002. doi:10.1109/LICS.2002.1029817.
 - 32 John C. Reynolds. Intuitionistic reasoning about shared mutable data structure. In *Millennial Perspectives in Computer Science*, pages 303–321. Palgrave, 2000.
 - 33 Tom Ridge, David Sheets, Thomas Tuerk, Andrea Giugliano, Anil Madhavapeddy, and Peter Sewell. Sibylfs: Formal specification and oracle-based testing for posix and real-world file systems. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP '15*, pages 38–53, New York, NY, USA, 2015. ACM. doi:10.1145/2815400.2815411.
 - 34 Kasper Svendsen and Lars Birkedal. Impredicative concurrent abstract predicates. In Zhong Shao, editor, *Programming Languages and Systems*, volume 8410 of *Lecture Notes in Computer Science*, pages 149–168. Springer Berlin Heidelberg, 2014. doi:10.1007/978-3-642-54833-8_9.
 - 35 Kasper Svendsen, Lars Birkedal, and Matthew Parkinson. Modular reasoning about separation of concurrent data structures. In Matthias Felleisen and Philippa Gardner, editors, *Programming Languages and Systems*, volume 7792 of *Lecture Notes in Computer Science*, pages 169–188. Springer Berlin Heidelberg, 2013. doi:10.1007/978-3-642-37036-6_11.
 - 36 Aaron Joseph Turon and Mitchell Wand. A separation logic for refining concurrent objects. *ACM SIGPLAN Notices*, 46(1):247–258, 2011.
 - 37 Viktor Vafeiadis and Matthew Parkinson. *A Marriage of Rely/Guarantee and Separation Logic*, pages 256–271. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007. doi:10.1007/978-3-540-74407-8_18.

A Characteristic Study of Parameterized Unit Tests in .NET Open Source Projects

Wing Lam

University of Illinois at Urbana-Champaign, USA
winglam2@illinois.edu

Siwakorn Srisakaokul

University of Illinois at Urbana-Champaign, USA
srisaka2@illinois.edu

Blake Bassett

University of Illinois at Urbana-Champaign, USA
rbasset2@illinois.edu

Peyman Mahdian

University of Illinois at Urbana-Champaign, USA
mahdian2@illinois.edu

Tao Xie

University of Illinois at Urbana-Champaign, USA
taoxie@illinois.edu

Pratap Lakshman

Microsoft, India
pratapl@microsoft.com

Jonathan de Halleux

Microsoft Research, USA
jhalleux@microsoft.com

Abstract

In the past decade, parameterized unit testing has emerged as a promising method to specify program behaviors under test in the form of unit tests. Developers can write parameterized unit tests (PUTs), unit-test methods with parameters, in contrast to conventional unit tests, without parameters. The use of PUTs can enable powerful test generation tools such as Pex to have strong test oracles to check against, beyond just uncaught runtime exceptions. In addition, PUTs have been popularly supported by various unit testing frameworks for .NET and the JUnit framework for Java. However, there exists no study to offer insights on how PUTs are written by developers in either proprietary or open source development practices, posing barriers for various stakeholders to bring PUTs to widely adopted practices in software industry. To fill this gap, we first present categorization results of the Microsoft MSDN Pex Forum posts (contributed primarily by industrial practitioners) related to PUTs. We then use the categorization results to guide the design of the first characteristic study of PUTs in .NET open source projects. We study hundreds of PUTs that open source developers wrote for these open source projects. Our study findings provide valuable insights for various stakeholders such as current or prospective PUT writers (e.g., developers), PUT framework designers, test-generation tool vendors, testing researchers, and testing educators.

2012 ACM Subject Classification Software and its engineering → Software testing and debugging

Keywords and phrases Parameterized unit testing, automated test generation, unit testing



© Wing Lam, Siwakorn Srisakaokul, Blake Bassett, Peyman Mahdian, Tao Xie, Pratap Lakshman, and Jonathan de Halleux;

licensed under Creative Commons License CC-BY

32nd European Conference on Object-Oriented Programming (ECOOP 2018).

Editor: Todd Millstein; Article No. 5; pp. 5:1–5:27



Leibniz International Proceedings in Informatics

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

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2018.5

Acknowledgements This work was supported in part by National Science Foundation under grants no. CCF-1409423, CNS-1513939, and CNS1564274.

1 Introduction

With advances in test generation research such as dynamic symbolic execution [23, 35], powerful test generation tools are now at the fingertips of software developers. For example, Pex [37, 39], a state-of-the-art tool based on dynamic symbolic execution, has been shipped as *IntelliTest* [32, 26] in Microsoft Visual Studio 2015 and 2017, benefiting numerous developers in software industry. Such test generation tools allow developers to automatically generate input values for the code under test, comprehensively covering various program behaviors and consequently achieving high code coverage. These tools help alleviate the burden of extensive manual software testing, especially on test generation.

Although such tools provide powerful support for automatic test generation, when they are applied directly to the code under test, only a predefined limited set of properties can be checked. These predefined properties serve as test oracles for these automatically generated input values, and violating these predefined properties leads to various runtime exceptions, such as null dereferencing or division by zero. Despite being valuable, these predefined properties are *weak test oracles*, which do not aim for checking functional correctness but focus on robustness of the code under test.

To supply strong test oracles for automatically generated input values, developers can write formal specifications such as code contracts [25, 30, 16] in the form of preconditions, postconditions, and object invariants in the code under test. However, just like writing other types of formal specifications, writing code contracts, especially postconditions, can be challenging. According to a study on code contracts [34], 68% of code contracts are preconditions while only 26% of them are postconditions (the remaining 6% are object invariants). Section 2 shows an example of a method under test whose postconditions are difficult to write.

In the past decade, parameterized unit testing [40, 38] has emerged as a practical alternative to specify program behaviors under test in the form of unit tests. Developers can write parameterized unit tests (PUTs), unit-test methods with parameters, in contrast to conventional unit tests (CUTs), without parameters. Then developers can apply an automatic test generation tool such as Pex to automatically generate input values for a PUT's parameters. Note that algebraic specifications [24] can be naturally written in the form of PUTs but PUTs are not limited to being used to specify algebraic specifications.

Since parameterized unit testing was first proposed in 2005 [40], PUTs have been popularly supported by various unit testing frameworks for .NET along with recent versions of the JUnit framework (as parameterized tests [14] and theories [33, 5]). However, there exists no study to offer insights on how PUTs are written by developers in development practices of either proprietary or open source software, posing barriers for various stakeholders to bring PUTs to widely adopted practices in software industry. Example stakeholders are current or prospective PUT writers (e.g., developers), PUT framework designers, test-generation tool vendors, testing researchers, and testing educators.

To address the lack of studies on PUTs, we first conduct a categorization of 93 Microsoft MSDN Pex Forum posts [31] (contributed primarily by industrial practitioners) related to parameterized unit tests. We then use the categorization results to guide the design of the

first characteristic study of PUTs in .NET open source projects (with a focus on PUTs written using the Pex framework, given that Pex is one of the most widely used test generation tools in industry [39]). Our findings from the categorization results of the forum posts show the following top three PUT-related categories that developers are most concerned with:

1. “Assumption/Assertion/Attribute usage” problems, which involve the discussion of using certain PUT assumptions, assertions, and attributes to address the issues faced by developers, are the most popular category of posts (occupying 23 of the 93 posts).
2. “Non-primitive parameters/object creation” problems, which involve the discussion of generating objects for PUTs with parameters of a non-primitive type, are the second most popular category of posts (occupying 17 of the 93 posts).
3. “PUT concept/guideline” problems, which involve the discussion of the PUT concept and general guidelines for writing good PUTs, are the third most popular category of posts (occupying 11 of the 93 posts).

Upon further investigation into these top PUT-related categories, we find that developers in general are concerned with when and what assumptions, assertions, and attributes they should use when they are writing PUTs. We find that a significant number of forum posts are directly related to how developers should replace hard-coded method sequences with non-primitive parameters of PUTs. We also find that developers often question what patterns their PUTs should be written in. Using our categorization and investigation results, we formulate three research questions and answer these questions using 11 open-source projects, which contain 741 PUTs.

In particular, we investigate the following three research questions and attain corresponding findings:

1. **What are the extents and the types of assumptions, assertions, and attributes being used in PUTs?** We present a wide range of assumption, assertion, and attribute types used by developers as shown in Tables 3a, 3b, and 5, and tool vendors or researchers can incorporate this data with their tools to better infer assumptions, assertions, and attributes to assist developers. For example, tool vendors or researchers who care about the most commonly used assumptions should focus on `PexAssumeUnderTest` or `PexAssumeNotNull`, since these two are the most commonly used assumptions. Lastly, based on the studied PUTs, we find that increasing the default value of attributes as suggested by tools such as Pex rarely contributes to increased code coverage. Tool vendors or researchers should aim to improve the quality of the attribute recommendations provided by their tools, if any are provided at all.
2. **How often can hard-coded method sequences in PUTs be replaced with non-primitive parameters and how useful is it to do so?** There are a significant number of receiver objects in the PUTs (written by developers) that could be promoted to non-primitive parameters, and a significant number of existing non-primitive parameters that lack factory methods (i.e., methods manually written to help the tools generate desirable object states for non-primitive parameters). It is worthwhile for tool researchers or vendors to provide effective tool support to assist developers to promote these receiver objects (resulted from hard-coded method sequences), e.g., inferring assumptions for a non-primitive parameter promoted from hard-coded method sequences. Additionally, once hard-coded method sequences are promoted to non-primitive parameters, developers can also use assistance in writing more factory methods for such parameters.
3. **What are common design patterns and bad code smells of PUTs?** By understanding how developers write PUTs, testing educators can teach developers appropriate ways to improve PUTs. For example, developers should consider splitting PUTs with

multiple conditional statements into separate PUTs each covering a case of the conditional statements. Doing so makes the PUTs easier to understand and eases the effort to diagnose the reason for test failures. Tool vendors and researchers can also incorporate this data with their tools to check the style of PUTs for suggesting how the PUTs can be improved. For example, checking whether a PUT contains conditionals, contains hard-coded test data, and contains duplicate test code, etc. often accurately identifies a PUT that can be improved.

In summary, this paper makes the following major contributions:

- The categorization of the Microsoft MSDN Pex Forum posts (contributed primarily by industrial practitioners) related to PUTs.
- The first characteristic study of PUTs in open source projects, with a focus on hundreds of real-world PUTs, producing study findings that provide valuable insights for various stakeholders.
- A collection of real-world open-source projects equipped with developer-written PUTs and a suite of tools for analyzing PUTs (both are used for our study and are released on our project website [2]). These PUTs and analysis tools can be used by the community to conduct future empirical studies or to evaluate enhancements to automated test generation tools.

The work in this paper is part of the efforts of our industry-academia team (including university/industrial testing researchers and tool vendors) for bringing parameterized unit testing to broad industrial practices of software development. To understand how automatic test generation tools interact with PUTs, we specifically study PUTs written with the Pex framework. Besides the Pex framework, other .NET frameworks such as NUnit also support PUTs. In recent years, PUTs are also increasingly adopted among Java developers, partly due to the inclusion of parameterized test [14] and theories [33, 5] in the JUnit framework. However, unlike the Pex framework, these other frameworks lack powerful test generation tools such as Pex to support automatic generation of tests with high code coverage, and part of our study with PUTs, specifically the part described in Section 5, does investigate the code coverage of the input values automatically generated from PUTs.

The remainder of this paper is organized as follows. Section 2 presents an example of parameterized unit testing. Section 3 discusses the categorization of Pex forum posts that motivates our study. Section 4 discusses the setup of our study. Section 5 presents our study findings and discusses the implications to stakeholders. Section 6 discusses threats to validity of our study. Section 7 presents our related work, and Section 8 concludes the paper.

2 Background

Consider the method under test from the open source project of NUnit Console [11] in Figure 1. One way to supply strong test oracles for automatically generated input values is to write preconditions and postconditions for this method under test. It is relatively easy to specify preconditions for the method as `(sn != null) && (sv != null)` but it is actually quite challenging to specify comprehensive postconditions to capture this method's intended behaviors. The reason is that this method's intended behaviors depend on the behaviors of all the method calls inside the `SaveSetting` method. In order to write postconditions for `SaveSetting`, we would need to know the postconditions of the other method calls in `SaveSetting` (e.g., `GetSetting`) as well. In addition, the postconditions can be very long since there are many conditional statements with complex conditions (e.g., Lines 8-11). If a method

```

1 public class SettingsGroup {
2     private Hashtable storage = new Hashtable();
3     public event SettingsEventHandler Changed;
4     public void SaveSetting(string sn, object sv) {
5         object ov = GetSetting(settingName);
6         //Avoid change if there is no real change
7         if(ov != null) {
8             if((ov is string && sv is string && (string)ov == (string)sv) ||
9                (os is int && sv is int && (int)ov == (int)sv) ||
10              (os is bool && sv is bool && (bool)ov == (bool)sv) ||
11              (os is Enum && sv is Enum && ov.Equals(sv)))
12                 return;
13         }
14         storage[settingName] = settingValue;
15         if(Changed != null)
16             Changed(this, new SettingsEventArgs(sn));
17     }
18 }

```

■ **Figure 1** SaveSetting method under test from the SettingsGroup class of NUnit Console [11].

contains loops, its postcondition may be even more difficult to write, since we would need to know the loop invariants and the postconditions may need to contain quantifiers. Thus, there is a need for a practical method to specify program behaviors under test in the form of unit tests. Specifying program behaviors in the form of unit tests can be easier since we do not need to specify all the intended behaviors of the method under test as a single logical formula. Instead, we can write test code to specify the intended behaviors of the method under test for a specific scenario (e.g., interacting with other specific methods). For example, a real-world conventional unit test (CUT) written by the NUnit developers is shown in Figure 2. The CUT in this figure checks that after we save a setting by calling the `SaveSetting` method, we should be able to retrieve the same setting by calling the `GetSetting` method. Despite seemingly comprehensive, the CUT in Figure 2 is insufficient, since it is unable to cover Lines 8-12 of the method in Figure 1. Figure 3 shows an additional CUT that developers can write to cover Lines 8-12; this additional CUT checks that saving the same setting twice does not invoke the `Changed` event handler twice. These two CUTs' corresponding, and more powerful, PUT is shown in Figure 4.

The beginning of the PUT (Lines 3-5) include `PexAssume` statements that serve as assumptions for the three PUT parameters. During test generation, Pex filters out all the generated input values (for the PUT parameters) that violate the specified assumptions. These assumptions are needed to specify the state of `SettingsGroup` that one may want to test. For example, according to Lines 2-3 in Figure 2, `sg` initially does not have "X" and "NAME" set. Thus, we need to add `PexAssume.IsNull(st.Getting(sn))` (Line 5) to force Pex to generate only an object of `SettingsGroup` that satisfies the same condition as Lines 2-3 in Figure 2. Otherwise, without such assumptions, the input values generated by Pex may largely be of no interest to the developers. The `PexAssert` statements in Lines 7 and 10 are used as the assertions to be verified when running the generated input values. More specifically, the assumption on Line 5 and the assertion on Line 7 in the PUT correspond to Lines 2-3 and Lines 6-7, respectively, in the CUT from Figure 2. Lines 8-9 in the PUT then cover the case of calling the `SaveSetting` method twice with the same parameters as accomplished in the CUT shown in Figure 3. Note that writing the PUT allows the test to be more general as variable `sn` can be any arbitrary string, better than hard-coding it to be only "X" or "NAME" (as done in the CUTs).

A PUT is annotated with the `[PexMethod]` attribute, and is sometimes attached with optional attributes to provide configuration options for automatic test generation tools. An example attribute is `[PexMethod(MaxRuns = 200)]` as shown in Figure 4. The `MaxRuns`

5:6 A Characteristic Study of Parameterized Unit Tests

```
1 public void SaveAndLoadSettings() {
2     Assert.IsNull(sg.GetSetting("X"));
3     Assert.IsNull(sg.GetSetting("NAME"));
4     sg.SaveSetting("X", 5);
5     sg.SaveSetting("NAME", "Charlie");
6     Assert.AreEqual(5, sg.GetSetting("X"));
7     Assert.AreEqual("Charlie", sg.GetSetting("NAME"));
8 }
```

■ **Figure 2** A real-world CUT for the method in Figure 1.

```
1 public void SaveSettingsWhenSettingIsAlreadyInitialized() {
2     Assert.IsNull(sg.GetSetting("X"));
3     sg.SaveSetting("X", 5);
4     sg.SaveSetting("X", 5);
5     // Below assert that Changed only got invoked once in SaveSetting
6     ...
7 }
```

■ **Figure 3** An additional CUT for the method in Figure 1 to cover the lines that the CUT in Figure 2 does not cover.

```
1 [PexMethod(MaxRuns = 200)]
2 public void TestSave1(SettingsGroup sg, string sn, object sv) {
3     PexAssume.IsTrue(sg != null && sg.Changed != null);
4     PexAssume.IsTrue(sn != null && sv != null);
5     PexAssume.IsNull(sg.GetSetting(sn));
6     sg.SaveSetting(sn, sv);
7     PexAssert.AreEqual(sv, sg.GetSetting(sn));
8     sg.SaveSetting(sn, sv);
9     // Below assert that Changed only got invoked once in SaveSetting
10    ...
11 }
```

■ **Figure 4** The PUT corresponding to the CUTs in Figures 2 and 3.

attribute along with the attribute value of 200 indicates that Pex can take a maximum of 200 runs/iterations during Pex’s path exploration phase for test generation. Since the default value of `MaxRuns` is 1000, setting the value of `MaxRuns` to be just 200 decreases the time that Pex may take to generate input values. Note that doing so may also cause Pex to generate fewer input values.

3 Categorization of Forum Posts

This section presents our categorization results of the Microsoft MSDN Pex Forum posts [31] related to parameterized unit tests. As of January 10th, 2018, the forum includes 1,436 posts asked by Pex users around the world. These users are primarily industrial practitioners. To select the forum posts related to parameterized unit tests, we search the forum with each of the keywords “parameterized”, “PUT”, and “unit test”. Searching the forum with these three keywords returns 14, 18, and 243 posts, respectively. We manually inspect each of these returned posts to select only posts that are actually related to parameterized unit tests. Finally among the returned posts, we identify 93 posts as those related to parameterized unit tests. Then we categorize these 93 posts into 8 major categories and one miscellaneous category, as shown in Table 1. The categorization details of the 93 posts can be found on our project website [2]. We next describe each of these categories and the number of posts falling into each category.

The posts falling into the top 1 category “assumption/assertion/attribute usage” (25% of the posts) involve discussion of using certain PUT assumptions, assertions, and attributes to address the issues faced by PUT users. The posts falling into the second most popular category “non-primitive parameters/object creation” (18% of the posts) involve discussion

■ **Table 1** Categorization results of the Microsoft MSDN Pex Forum posts related to parameterized unit tests.

Category	#Posts
Assumption/Assertion/Attribute usage	25% (23/93)
Non-primitive parameters/object creation	18% (17/93)
PUT concept/guideline	12% (11/93)
Test generation	11% (10/93)
PUT/CUT relationship	9% (8/93)
Testing interface/generic class/abstract class	6% (6/93)
Code contracts	5% (5/93)
Mocking	5% (5/93)
Miscellaneous	9% (8/93)
Total	100% (93/93)

of generating objects for PUTs with non-primitive-type parameters, one of the two major issues [42] for Pex to generate input values for PUTs. The posts falling into category “PUT concept/guideline” (12% of the posts) involve discussion of the PUT concept and general guideline for writing good PUTs. The posts falling into category “test generation” (11% of the posts) involve discussion of Pex’s test generation for PUTs. The posts falling into category “PUT/CUT relationship” (9% of the posts) involve discussion of co-existence of both CUTs and PUTs for the code under test. The posts falling into category “testing interface/generic class/abstract class” (6% of the posts) involve discussion of writing PUTs for interfaces, generic classes, or abstract classes. The posts falling into category “code contracts” (5% of the posts) involve discussion of writing PUTs for code under test equipped with code contracts [25, 30, 16]. The posts falling into category “mocking” (5% of the posts) involve discussion of writing mock models together with PUTs. The miscellaneous category (9% of the posts) includes those other posts that cannot be classified into one of the 8 major categories.

We use the posts from the top 3 major categories to guide our study design described in the rest of the paper, specifically with research questions RQ1-RQ3 listed in Section 5. In particular, our study focuses on quantitative aspects of assumption, assertion, and attribute usage (top 1 category) in RQ1, non-primitive parameters/object creation (top 2 category) in RQ2, and PUT concept/guideline (top 3 category) in RQ3.

4 Study Setup

This section describes our process for collecting subjects (e.g., open source projects containing PUTs) and the tools that we develop to collect and process data from the subjects. The details of these subjects and our tools can be found on our project website [2].

4.1 Subject-collection Procedure

The subject-collection procedure (including subject sanitization) is a multi-stage process. At a coarse granularity, this process involves (1) comprehensive and extensive subject collection from searchable online source code repositories, (2) deduplication of subjects obtained multiple times from different repositories, and (3) verification of developer-written parameterized unit tests (e.g., filtering out subjects containing only automatically-generated parameterized test stubs).

For comprehensive collection of subjects, we query a set of widely known code search services. The used query is “`PexMethod Assert`”, requiring both “`PexMethod`” and “`Assert`” to appear in the source file of the search results. The two code search services that return non-empty results based on our search criteria are GitHub [9] and SearchCode [4]. For each code search service, we first search with our query, and then we extract the source code repositories containing the files in the search results. When a particular repository is available from multiple search services, we extract the version of the repository from the search service that has the most recent commit. Lastly, we manually verify that each of our source code repositories has at least one PUT with one or more parameters and one or more assertions.

4.2 Analysis Tools

We develop a set of tools to collect metrics from the subjects. We use Roslyn [10], the .NET Compiler Platform, to build our tools. These tools parse C# source files to produce an abstract syntax tree, which is traversed to collect information and statistics of interest. More specifically, the analysis tools statically analyze the C# source code in the .cs files of each subject. The outputs of the tools include but are not limited to the following: PUTs, PUTs with `if` statements, results in Tables 3 and 6, the number of assumption and assertion clauses, and attributes of the subjects’ PUTs. In general, the results that we present in the remainder of the paper are collected either directly with the analysis tools released on our website [2], manual investigation conducted by the authors, or a combination of the two (e.g., using the PUTs with `if` statements to manually categorize the number of PUTs that have unnecessary `if` statements).

4.3 Collected Subjects

In total, we study 77 subjects and retain only the subjects that contain at least 10 PUTs and are not used for university courses or academic research (e.g., creating PUTs to experiment with Pex’s capability of achieving high code coverage). This comprehensive list of subjects that we study can be found on our project website [2].

Table 2 shows the information on the subjects that contain at least 10 PUTs. We count a test method as a PUT if the test method is annotated with attribute “`PexMethod`” and has at least one parameter. Our detailed study for research questions focuses on subjects with at least 10 PUTs because a subject with fewer PUTs often includes occasional tryouts of PUTs instead of serious use of them for testing the functionalities of the open source project. Column 1 shows the name of each subject, and Columns 2-3 shows the number of PUTs and CUTs in each subject. Columns 4-6 show the number of the lines of production source code, PUTs and CUTs, respectively, in each subject. Columns 7-8 shows the percentage of statements covered in the project under test by the PUTs on which Pex is applied and by the CUTs of the subject. Column 9 shows the version of Pex a subject’s PUTs were written with. If a subject contains PUTs written with multiple versions of Pex, the most recent version of Pex used to write the subject’s PUTs is shown. Altogether, we identify 11 subjects with at least 10 PUTs, and these subjects contain a total of 741 PUTs. When we examine the profiles of the contributors to the subjects, we find that all but one of the subjects have contributors who work in industry. The remaining one subject, `PurelyFunctionalDataStructures`, referred to as PFDS in our tables, is developed by a graduate student imitating the algorithms in a data structure textbook. The table shows the percentage of statements covered for only 5 out of 11 subjects because we have difficulties compiling the other subjects (e.g., a subject misses some dependencies). Part of our future work is to debug the remaining subjects so that we

■ **Table 2** Subjects collected for our study.

Subject Name	#Methods		Source	#LOC		Code Cov.		Pex Version
	PUT	CUT		PUT	CUT	PUT	CUT	
Atom	240	297	127916	3570	3983	N/A	N/A	0.20.41218.2
BBCode	17	22	1576	188	219	84%	69%	0.94.0.0
ConcurrentList	23	57	315	243	645	51%	75%	0.94.0.0
Functional-dotnet	41	87	14002	355	1666	N/A	N/A	0.15.40714.1
Henoch	63	149	4793	142	2816	N/A	N/A	0.94.0.0
OpenMheg	45	6	21809	382	100	N/A	N/A	0.6.30728.0
PFDS	10	2	1818	120	34	50%	12%	0.93.0.0
QuickGraph	205	123	38530	1478	2186	5%	50%	0.94.0.0
SerialProtocol	34	0	7603	269	0	49%	0%	0.94.0.0
Shweet	12	42	2481	295	703	N/A	N/A	0.91.50418.0
Utilities-net	51	0	3224	475	0	26%	0%	0.94.0.0
Total	741	785	223158	7496	12352	-	-	-
Average	67	71	22174	681	1123	52%	41%	-

can compile them. More details about the subjects (e.g., the contributors of the subjects, the number of public methods in the subjects) can be found on our project website [2].

5 Study Results

Our study is based on forum posts asked by Pex users around the world as detailed in Sections 5.1 to 5.3. Our study findings aim to benefit various stakeholders such as current or prospective PUT writers (e.g., developers), PUT framework designers, test-generation tool vendors, testing researchers, and testing educators. In particular, our study intends to address the following three main research questions:

- **RQ1:** What are the extents and the types of assumptions, assertions, and attributes being used in PUTs?
 - We address RQ1 because addressing it can help understand developers' current practice of writing assumptions, assertions, and attributes in PUTs, and better inform stakeholders future directions on providing effective tool support or training on writing assumptions, assertions, and attributes in PUTs.
- **RQ2:** How often can hard-coded method sequences in PUTs be replaced with non-primitive parameters and how useful is it to do so?
 - We address RQ2 because addressing it can help understand the extent of writing sufficiently general PUTs (e.g., promoting an object produced by a method sequence hard-coded in a PUT to a non-primitive parameter of the PUT) to fully leverage automatic test generation tools.
- **RQ3:** What are common design patterns and bad code smells of PUTs?
 - We address RQ3 because addressing it can help understand how developers are currently writing PUTs and identify better ways to write good PUTs.

5.1 RQ1. Assumptions, Assertions, and Attributes

To understand developers' practices of writing assumptions, assertions, and attributes in PUTs, we study our subjects' common types of assumptions, assertions, and attributes. Our study helps provide relevant insights to the posts from the Assumption/Assertion/Attribute

■ **Table 3**

(a) Different types of assumptions in subjects.

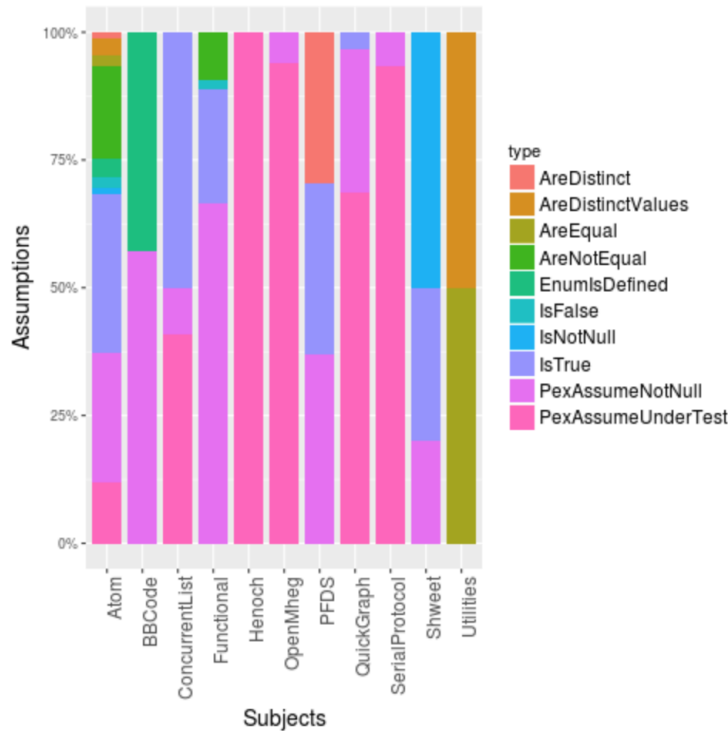
(b) Different types of assertions in subjects.

PexAssume Type	#	#NC	PexAssert Type	#	#NC
PexAssumeUnderTest	273	273	AreEqual	355	0
PexAssumeNotNull	211	211	IsTrue	199	2
IsTrue	158	2	IsFalse	75	3
AreNotEqual	73	0	Inconclusive	43	0
EnumIsDefined	22	0	IsNotNull	26	26
AreDistinct	13	0	Equal	21	1
AreDistinctValues	13	0	TrueForAll	19	0
IsNotNull	10	10	That	17	0
IsFalse	9	0	AreElementsEqual	16	0
AreEqual	9	0	IsNull	9	9
TrueForAll	7	2	AreNotEqual	5	0
IsNotNullOrEmpty	4	4	Fail	5	0
Fail	4	0	Throws	5	0
InRange	3	0	AreBehaviorsEqual	4	0
AreElementsNotNull	1	1	ImpliesIsTrue	3	0
Total	810	503	FALSE	3	0
Null Check Percentage	62% (503/810)		TRUE	3	0
			Empty	2	0
			Implies	2	0
			Contains	1	0
			DoesNotContain	1	0
			ReachEventually	1	0
			Total	815	41
			Null Check Percentage	5% (41/815)	

usage category described in Section 3. For example, the original poster of the forum post titled “New to Unit Testing” questions what type of assertions she/he should use. Another forum post titled “Do I use NUnit Assert or PexAssert inside my PUTs?” reveals that the original poster does not understand when and what assumptions to use.

5.1.1 Assumption Usage

As shown in Table 3a, `PexAssumeUnderTest` is the most common type of assumption, used 273 times in our subjects. `PexAssumeUnderTest` marks parameters as non-null and to be that precise type. The second most common type of assumption, `PexAssumeNotNull`, is used 211 times. Similar to `PexAssumeUnderTest`, `PexAssumeNotNull` marks parameters as non-null except that it does not require their types to be precise. Both `PexAssumeUnderTest` and `PexAssumeNotNull` are specified as attributes of parameters, but they are essentially a convenient alternative to specifying assumptions (e.g., the use of attribute `PexAssumeNotNull` on a parameter `X` is the same as `PexAssume.IsNotNull(X)`). Since PUTs are commonly written to test the behavior of non-null objects as the class under test or use non-null objects as arguments to a method under test, it is reasonable that the common assumption types used by developers are ones that mark parameters as non-null. Figure 5 shows that the



■ **Figure 5** Assumption-type distribution for each of our subjects.

combination of `PexAssumeUnderTest`, `PexAssumeNotNull`, and `IsNotNull`, which are for nullness checking, appears the most in all of our subjects. Note that Figure 5 contains only the top 10 commonly used assumption types in our subjects. Furthermore, according to the last row of Tables 3a and 3b, developers perform null checks much more frequently for assumptions than assertions. Our findings about the frequency of assumption types and assertion types that check whether objects are null are similar to the findings of a previous study [34] on how frequently preconditions and postconditions in code contracts are used to check whether objects are null. Similar to code contracts, we find that 62% of assumptions perform null checks while the study on code contracts finds that 77% (1079/1356) of preconditions perform null checks. Our study also finds that 5% of assertions perform null checks while the study on code contracts finds that 43% (165/380) of postconditions perform null checks. Since assertions are validated at the end of a PUT and it is less often that code before the assertions manipulates or produces a null object, it is reasonable that assumptions check for null much more frequently than assertions do. For assumption and assertion types such as `TrueForAll`, developers' low number of uses may be due to the unawareness of such types' existence. `TrueForAll` checks whether a predicate holds over a collection of elements. In our subjects, we find cases such as the one in Figure 6 where a collection is iterated over to check whether a predicate is true for all of its elements; instead, developers could have used the `TrueForAll` assumption or assertion. More specifically, the developers of the method in Figure 6 could have replaced Lines 5-8 with `PexAssert.TrueForAll(enumerable.Cast<T>(), item => matrix.Contains(item))`. It is important to note that in versions of Pex after 0.94.0.0, certain assumption and assertion types were removed (e.g., `TrueForAll`). However, as shown in Table 2, none of our subjects used versions of Pex after 0.94.0.0.

```

1  [PexMethod]
2  public void GetEnumerator_WhenMatrixConvertedToEnumerable_IteratesOverAllElements<T>(
3      [PexAssumeNotNull]ObjectMatrix<T> matrix ) {
4      System.Collections.IEnumerable enumerable = matrix;
5      foreach(var item in enumerable.Cast<T>())
6      {
7          Assert.IsTrue( matrix.Contains( item ) );
8      }
9  }

```

■ **Figure 6** PUT (in Atom [1]) that could benefit from Pex’s `TrueForAll` assertion.

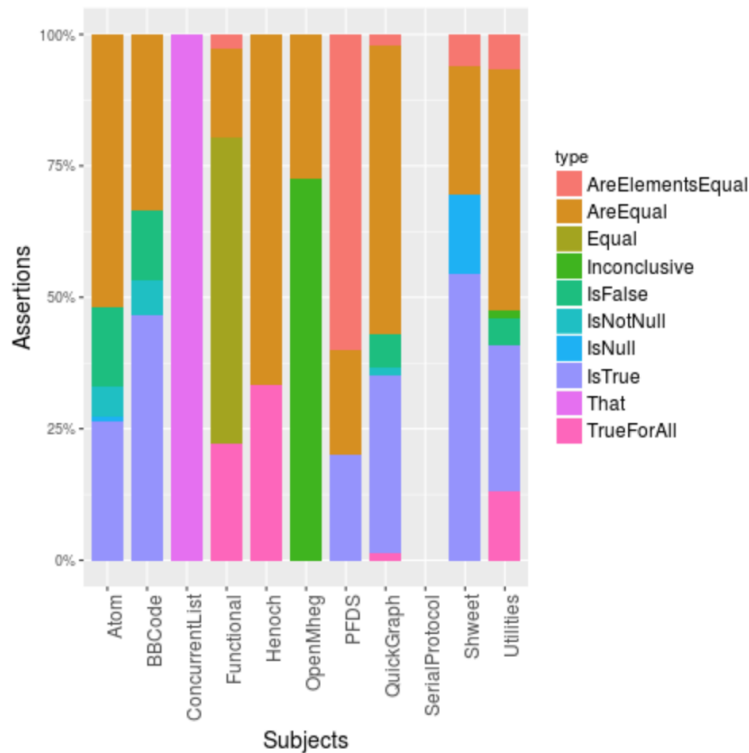
5.1.2 Assertion Usage

According to Figure 7, in all of the subjects except OpenMhcg, the PUTs usually contain assertions for nullness or equality checking. Instead, OpenMhcg’s assertions are mainly `Assert.Inconclusive`. `Assert.Inconclusive` is used to indicate that a test is still incomplete. From our inspection of the PUTs with `Assert.Inconclusive` in OpenMhcg, we find that developers write `Assert.Inconclusive("this test has to be reviewed")` in the PUTs. When we investigate the contents of these PUTs, we find that the developers indeed use these assertions to keep track of which tests are still incomplete. One example of OpenMhcg’s PUT that contains `Assert.Inconclusive` is shown in Figure 8. The example is one of many PUTs in OpenMhcg that create a new object but then do nothing with the object and contain no other assertions but `Assert.Inconclusive`. When we ignore all PUTs of OpenMhcg that contain only `Assert.Inconclusive`, we find that the remaining assertions are similar to our other subjects in that most of them are for nullness or equality checking.

As shown in Table 4, the PFDS subject has the highest number of assume clauses per PUT method. Upon closer investigation of PFDS’s assume clauses, we find that these clauses are necessary because PUTs in PFDS test various data structures and the developers of PFDS have to specify assumptions for all of its PUTs to guide Pex to generate data-structure inputs that are not null and contain some elements. When we examine the assume clauses in Atom, the subject with the second highest number of assume clauses per PUT method, we also find similar cases. On the other hand, the Shweet subject has the highest number of assert clauses per PUT method. Shweet’s high number of assert clauses per PUT method can be attributed to the fact that the subject has multiple PUTs each of which contains around 8 assertions. The reason why some of Shweet’s PUTs each have around 8 assertions is that the subject’s PUTs test a web service, and the service returns 8 values every time it is triggered. Therefore, multiple of Shweet’s PUTs assert for whether these 8 values are correctly returned or not.

5.1.3 Attribute Usage

To investigate developers’ practices of configuring Pex via PUT attributes, we study the number and settings of attributes, as configuration options for running Pex, written by developers in PUTs. Our findings from the forum posts related to attributes suggest that developers are often confused on what attributes to use or how they should configure attributes. More specifically, 5 out of 23 of the Assumption/Assertion/Attribute usage forum posts involve an answer recommending the use of a particular attribute or configuring an attribute in a specific way. For example, a post titled “the test state was: path bounds exceeded - infinite loop” discusses how developers should set the `MaxBranches` attribute of Pex. The setting of `MaxBranches` controls the maximum number of branches taken by Pex along a single execution path.



■ **Figure 7** Assertion-type distribution for each of our subjects.

```

1 [PexMethod]
2 public Content Constructor03(GenericContentRef genericContentRef) {
3     Content target = new Content(genericContentRef);
4     Assert.Inconclusive("this test has to be reviewed");
5     return target;
6 }

```

■ **Figure 8** PUT (in OpenMheg [12]) that contains `Assert.Inconclusive`.

The fourth column of Table 4 shows the average number of attributes added per PUT. The results show that developers add only 1 attribute for every 3-4 PUTs. Table 5 shows the number of attributes added for our subjects. Common attributes that developers add are `MaxRuns`, `MaxConstraintSolverTime`, and `MaxBranches`. The setting of `MaxRuns` controls the maximum number of runs before Pex terminates. Developers commonly set this attribute to be 100 runs when the default value is 1,000. Upon our inspection, most of the PUTs that use this attribute test methods related to inserting objects into a data structure. By setting the value of this attribute, developers make Pex terminate faster. In fact, 14 out of 18 attributes used in QuickGraph are `MaxRuns`.

`MaxConstraintSolverTime` is another type of attribute that some projects contain. The attribute controls the constraint solver's timeout value during Pex's exploration. By default, `MaxConstraintSolverTime` is set to 10 seconds. Similar to `MaxRuns`, we find that developers often set the value to be lower than the default value so that Pex would finish sooner. For example, BBCode contains PUTs with `MaxConstraintSolverTime` set to 5 seconds, and Atom contains PUTs with `MaxConstraintSolverTime` set to 2 seconds.

■ **Table 4** Number of PexAssume clauses, PexAssert clauses, and Pex Attributes per PUT.

Subject Name	# of Assume Cl. / PUT	# of Assert Cl. / PUT	# of Attrs / PUT
Atom	1.72 (412/240)	1.71 (411/240)	0.07 (16/240)
BBCode	1.71 (29/ 17)	1.47 (25/ 17)	2.18 (37/ 17)
ConcurrentList	0.96 (22/ 23)	0.74 (17/ 23)	0.26 (6/ 23)
Functional-dotnet	1.39 (57/ 41)	1.24 (51/ 41)	0.17 (7/ 41)
Henoch	0.78 (49/ 63)	0.05 (3/ 63)	0.38 (24/ 63)
OpenMheg	0.76 (34/ 45)	1.29 (58/ 45)	0.00 (0/ 45)
PFDS	2.70 (27/ 10)	1.10 (11/ 10)	0.00 (0/ 10)
QuickGraph	0.91 (186/205)	0.85 (175/205)	0.10 (21/205)
SerialProtocol	0.44 (15/ 34)	0.00 (0/ 34)	0.00 (0/ 34)
Shweet	1.00 (12/ 12)	3.42 (41/ 12)	0.33 (4/ 12)
Utilities-net	0.18 (9/ 51)	1.37 (70/ 51)	0.00 (0/ 51)
Average	1.14	1.20	0.32

■ **Table 5** Different types of Pex attributes in our subjects' PUTs.

Pex Attribute Type	#
MaxBranches	36
MaxRuns	18
MaxConstraintSolverTime	12
MaxConditions	8
MaxRunsWWithoutNewTests	6
MaxStack	5
Timeout	4
MaxExecutionTreeNodes	4
MaxWorkingSet	4
MaxConstraintSolverMemory	4
Total	101

In contrast to `MaxRuns`, we find that developers commonly set the value of `MaxBranches` to be higher than the default value. A common value set by developers is 20,000 when the default value is 10,000. When we study these PUTs, we find that the code tested by these PUTs all has loops, and the developers' intention when using this attribute is to increase the number of loop iterations allowed by Pex. For example, `ConcurrentList` contains several PUTs with `MaxBranches` = 20000 set. When we run Pex without this attribute, Pex suggests to set `MaxBranches` to 20000. However, when we compare the code coverage with and without the attribute being set, we find that the code coverage does not increase with the attribute set. In fact, we find that when we manually unset all attributes of `ConcurrentList`, the code coverage does not change at all. The number of input values (generated by Pex) that exhibit a failed test result also does not change. Our findings indicate that increasing the default values of attributes often does not help increase the code coverage. In fact, for some of `BBCode`'s PUTs, its developers set 9 different attributes all to the value of 1,000,000,000. Based on our estimation of running Pex on these PUTs, it would take approximately 2000 days for Pex to terminate. When we run Pex with a time limit of three hours on `BBCode`'s PUTs with the developer-specified attributes, we notice that the coverage increases marginally by less

than 1% compared to running Pex with the same time limit on BBCode's PUTs without any attributes.

5.1.4 Implications

With the wide range of assumption and assertion types used by developers as shown in Tables 3a and 3b, tool vendors or researchers can incorporate this data with their tools to better infer assumptions and assertions to assist developers. For example, tool vendors or researchers who care about the most commonly used assumption types should focus on `PexAssumeUnderTest` or `PexAssumeNotNull`, since these two are the most commonly used assumption types. Lastly, based on our subjects' PUTs, we find that increasing the default value of attributes as suggested by tools such as Pex rarely contributes to increased code coverage. Tool vendors or researchers should aim to improve the quality of the attribute recommendations provided by their tools, if any are provided at all.

5.2 RQ2. Non-primitive Parameters

Typically developers are expected to avoid hard-coding a method sequence in a PUT to produce an object used for testing the method under test. Instead, developers are expected to promote such objects to a non-primitive parameter of the PUT. In this way, the PUT can be made more general, to capture the intended behavior and enable an automatic test generation tool such as Pex to generate objects of various states for the non-primitive parameter. We find that 4 out of 17 answers from our non-primitive parameters/object creation category of forum posts described in Section 3 are directly related to how developers should replace hard-coded method sequences with non-primitive parameters. For example, in a forum post titled "Can Pex Generate a `List<T>` for my PUT", one of the answers to the question is that the developer should write a PUT that takes `List` as a non-primitive parameter instead of hard-coding a specific method sequence for producing a `List` object. Doing so enables Pex to generate non-empty, non-null objects of that list. Since many of our forum posts are related to how developers should replace hard-coded method sequences with non-primitive parameters, we decide to study how frequently developers write PUTs with non-primitive parameters and how often hard-coded method sequences in these PUTs could be replaced with non-primitive parameters. More details about the forum posts specifically related to this research question can be found on our project website [2].

5.2.1 Non-primitive Parameter Usage

As shown in Table 6, our result indicates that developers on average write non-primitive parameters 59.0% of the time for the PUTs in our subjects. In other words, for every 10 parameters used by developers, 5-6 of those parameters are non-primitive. However, developers write factory methods for only 17.9% of the non-primitive parameters used in our subjects' PUTs. The lack of non-primitive parameters and factory methods for such parameters inhibits test generation tools such as Pex from generating high-quality input values. For example, Figure 9 depicts 1 out of 16 PUTs that tests the `BinaryHeap` data structure in the `QuickGraph` subject. Promoting the object that it is testing (`BinaryHeap`) to a non-primitive parameter enables Pex to use factory methods such as the one depicted in Figure 10 to generate high-quality input values. Without promoting the `BinaryHeap` object to a parameter and using a factory method such as the one in Figure 10, the input values generated by Pex with the 16 PUTs can cover only 13% of the code blocks in the `BinaryHeap`

■ **Table 6** Statistics for factory methods and non-primitive parameters of our subjects. Average is calculated by dividing the sum of the two relevant columns (e.g., 59.0% is from the sum of Column 3 / the sum of Column 2).

Subject Name	Total Params	Non-prim Params	Non-prim / Params	Non-prim Params w/ Factory	w/ Factory / Non-prim Params
Atom	456	290	63.6%	66	22.8%
BBCode	33	9	27.3%	0	0.0%
ConcurrentList	16	0	0.0%	0	-
Functional-dotnet	50	5	10.0%	2	40.0%
Henoch	54	48	88.9%	0	0.0%
OpenMheg	75	55	73.3%	0	0.0%
PFDS	10	10	100.0%	0	0.0%
QuickGraph	125	111	88.8%	21	18.9%
SerialProtocol	51	21	41.2%	12	57.1%
Shweet	21	1	4.8%	0	0.0%
Utilities-net	66	15	22.7%	0	0.0%
Average			59.0%		17.9%

class as opposed to 80% when the `BinaryHeap` object is promoted and a factory method is provided for it. When developers do not promote non-primitive objects to a non-primitive parameter or provide factory methods for it, the effectiveness of their tests really depends on the values that the developers use to initialize the objects in their tests. For example, if developers do not promote the `BinaryHeap` object to a parameter or provide factory methods for it, then depending on the values that the developers would use to initialize the `BinaryHeap` object, the code blocks covered by the 16 PUTs could actually range from 13% to 80% (the same as that achieved by promoting the `BinaryHeap` object to a parameter and providing a factory method for it). Promoting the `BinaryHeap` object to a parameter and providing factory methods for it not only enable tools such as Pex to generate objects of `BinaryHeap` that the developers may not have thought of themselves, but also alleviate the burden of developers to choose the right values for their tests to properly exercise the code under test. It is important to note that if we just promote the `BinaryHeap` object in the 16 PUTs but do not provide a factory method for it, the percentage of code blocks covered by the PUTs is 52%. Our findings here suggest that to enable tools such as Pex to generate input values that cover the most code, it is desirable to promote non-primitive objects in PUTs to non-primitive parameters and provide factory methods for such parameters. However, even if no factory methods are provided, simply promoting non-primitive objects in PUTs may already increase the code coverage achieved by the input values generated by tools such as Pex.

5.2.2 Promoting Receiver Object

To determine how often developers could have replaced a hard-coded method sequence with a non-primitive parameter, we manually inspect each PUT to determine the number of them that could have had their receiver objects be replaced with a non-primitive parameter. We consider an object of a PUT to be a receiver object if the object directly or indirectly affects the PUT's assertions. The detailed results of our manual inspection effort can be found on


```

1 [PexMethod(MaxRuns = 100)]
2 [PexAllowedExceptionFromTypeUnderTest(typeof(InvalidOperationException))]
3 public void InsertAndRemoveMinimum<TPriority, TValue>(
4     [PexAssumeUnderTest]BinaryHeap<TPriority, TValue> target,
5     [PexAssumeNotNull] KeyValuePair<TPriority, TValue>[] kvs)
6 {
7     var count = target.Count;
8     foreach (var kv in kvs)
9         target.Add(kv.Key, kv.Value);
10    TPriority minimum = default(TPriority);
11    for (int i = 0; i < kvs.Length; ++i)
12    {
13        if (i == 0)
14            minimum = target.RemoveMinimum().Key;
15        else
16        {
17            var m = target.RemoveMinimum().Key;
18            Assert.IsTrue(target.PriorityComparison(minimum, m) <= 0);
19            minimum = m;
20        }
21        AssertInvariant(target);
22    }
23    Assert.AreEqual(0, target.Count);
24 }

```

■ **Figure 9** InsertAndRemoveMinimum PUT from the BinaryHeapTest class of QuickGraph [3].

```

1 [PexFactoryMethod(typeof(BinaryHeap<int, int>))]
2 public static BinaryHeap<int, int> Create(int capacity)
3 {
4     var heap = new BinaryHeap<int, int>(capacity, (i, j) => i.CompareTo(j));
5     return heap;
6 }

```

■ **Figure 10** Factory method for the BinaryHeapTest class of QuickGraph [3].

our project website [2] under “PUT Patterns”. As shown in Table 7, 95.7% (709/741) of the PUTs in our subjects have at least one receiver object. However, we find that 49.2% (349/709) of these PUTs with receiver objects do not have a parameter for the receiver objects, and 89.4% (312/349) of them can actually be modified so that all receiver objects in the PUT are promoted to PUT parameters. As shown in Table 8, we categorize the 349 PUTs whose receiver objects could be promoted into the following four different categories. (1) In 47.9% (167/349) of the PUTs, we can easily promote their receiver objects into a non-primitive parameter (e.g., removing the object creation line and adding a parameter). (2) In 41.5% (145/349) of the PUTs, their receiver objects are static (which cannot be instantiated). (3) In 9.7% (34/349) of the PUTs, they are testing their receiver objects’ constructors. (4) In 1.6% (3/349) of the PUTs, they are testing multiple receiver objects with shared variables (e.g., testing the equals method of an object).

Of the PUTs belonging to the first category shown in Table 8, 33.0% (55/167) of them test specific object states. Figure 11 shows an example of a PUT that tests a specific object state. The developers of this PUT could have promoted `_list` and `element` to parameters and updated `index` accordingly before the assertion in Line 9. Figure 12 depicts a more general version of the PUT in Figure 11. Notice how the initial contents of the list and the element being added to the list are hard-coded in Figure 11 but not in Figure 12.

Upon further investigation, we find that the 145 PUTs in the second category shown in Table 8 can and should actually be promoted by making the class under test not be static. On the other hand, the PUTs that test their receiver objects’ constructors have no need to be improved by promotion. Lastly, the PUTs that test multiple receiver objects are best left not promoted. In the end we find that the 167 PUTs in the first category (their receiver objects can be easily promoted) and the 145 PUTs in the second category (their receiver objects are static) are PUTs whose receiver objects could be promoted and they should actually be

■ **Table 7** Statistics of PUTs with receiver objects (ROs).

Subject Name	# of PUTs w/ ROs	# of PUTs w/o promoted ROs	# of PUTs whose ROs should be promoted
Atom	90.4% (217/240)	59.4% (129/217)	98.4% (127/129)
BBCode	88.2% (15/ 17)	100.0% (15/ 15)	100.0% (15/ 15)
ConcurrentList	100.0% (23/ 23)	56.5% (13/ 23)	100.0% (13/ 13)
Functional-dotnet	85.4% (35/ 41)	91.4% (32/ 35)	100.0% (32/ 32)
Henoch	100.0% (63/ 63)	25.4% (16/ 63)	43.8% (7/ 16)
OpenMheg	100.0% (45/ 45)	25.0% (11/ 45)	18.2% (2/ 11)
PFDS	100.0% (10/ 10)	100.0% (10/ 10)	100.0% (10/ 10)
QuickGraph	99.5% (204/205)	20.1% (41/204)	73.2% (30/ 41)
SerialProtocol	100.0% (34/ 34)	55.9% (19/ 34)	68.4% (13/ 19)
Shweet	100.0% (12/ 12)	100.0% (12/ 12)	100.0% (12/ 12)
Utilities-net	100.0% (51/ 51)	100.0% (51/ 51)	100.0% (51/ 51)
Total	95.7% (709/741)	49.2% (349/709)	89.4% (312/349)

```

1 [PexMethod]
2 public void GetItem(int index) {
3     IList<int> _list = new ConcurrentList<int>();
4     PexAssume.IsTrue(index >= 0);
5     const int element = 5;
6     for (int i = 0; i < index; i++)
7         _list.Add(0);
8     _list.Add(element);
9     Assert.That(_list[index], Is.EqualTo(element));
10 }

```

■ **Figure 11** PUT testing a specific object state in ConcurrentList [7].

```

1 [PexMethod]
2 public void GetItem_Promoted(int index, IList<int> _list, int element) {
3     int size = _list.Count;
4     PexAssume.IsTrue(index >= 0);
5     for(int i = 0; i < index; i++)
6         _list.Add(0);
7     _list.Add(element);
8     index += size;
9     Assert.That(_list[index], Is.EqualTo(element));
10 }

```

■ **Figure 12** Version of the PUT in Figure 11 with receiver object promoted.

promoted. These two categories of PUTs form the total of 89.4% (312/394) of the PUTs that could be promoted and should be promoted. Promoting these objects enables test generation tools such as Pex to use factory methods to generate different states of the receiver objects (beyond specific hard-coded ones) for the PUTs.

Based on our promotion experiences, often the time, after we promote receiver objects (resulted from hard-coded method sequences) to non-primitive parameters of PUTs, we need to add assumptions to constrain the non-primitive parameters so that test generation tools will not generate input values that are of no interest to developers. For example, for the `GetItem_Promoted` PUT in Figure 12, one of the input values generated by Pex with this PUT can be found in Figure 13. Although the value of `index` (0) from the `GetItem_CUT` in Figure 13 is reasonable for both the `GetItem` and `GetItem_Promoted` PUTs and the value of `element` (5) is reasonable for the `GetItem_Promoted` PUT, the additional value of `_list` (null) is unreasonable. The value is unreasonable because the `GetItem` PUT is expected to test

■ **Table 8** Categorization results of the PUTs whose receiver objects could be promoted.

Category	#PUTs
(1) Their receiver objects can be easily promoted	167 (47.9%)
(2) Their receiver objects are static	145 (41.5%)
(3) Testing their receiver objects' constructors	34 (9.7%)
(4) Testing multiple receiver objects with shared variables	3 (0.9%)
Total	349

```

1 [TestMethod]
2 public void GetItem_CUT()
3 {
4     GetItem_Promoted(0, null, 5);
5 }

```

■ **Figure 13** Example of a CUT generated from the PUT in Figure 12.

adding various elements to `_list` but it is not expected to test the case when `_list` is null. However, due to our promotion of `_list`'s hard-coded method sequence to a non-primitive parameter, input values generated from `GetItem_Promoted` would actually test such a case. In order for developers to prevent such nonsensical input values from being generated, the developers would have to add the assumption of `PexAssume.IsNotNull(_list)` before Line 3 of `GetItem_Promoted`. Such assumption writing can be time-consuming: essentially promoting hard-coded method sequences to be non-primitive parameters and adding assumptions to these parameters are going from specifying “how” (to generate specific object states) to specifying “what” (specific object states need to be generated).

5.2.3 Implications

There are a significant number of receiver objects (in the PUTs written by developers) that could be promoted to non-primitive parameters, and a significant number of existing non-primitive parameters that lack factory methods. It is worthwhile for tool researchers or vendors to provide effective tool support to assist developers to promote these receiver objects (resulted from hard-coded method sequences), e.g., inferring assumptions for a non-primitive parameter promoted from hard-coded method sequences. Additionally, once hard-coded method sequences are promoted to non-primitive parameters, developers can also use assistance in writing effective factory methods for such parameters.

5.3 RQ3. PUT Design Patterns and Bad Smells

Our categorization of forum posts as described in Section 3 shows that 5 out of 11 of the PUT concept/guideline posts discuss patterns in which PUTs should be written in. For example, two of the posts titled “Assertions in PUT” and “PUT with PEX” involve answers informing the original poster that assertions are typically necessary for PUTs. One such forum post contains the following response: “You should write Asserts, in order to ensure that the Function (TestInvoice in this case) really does what it is intended to do”. To better understand how developers write PUTs, we manually inspect all of the PUTs in our subjects to see what the common design patterns and bad smells are. The detailed results of our manual inspection effort can be found on our project website [2] under “PUT Patterns”.

```

1  [PexMethod]
2  public void Clear<T>([PexAssumeUnderTest]ConcurrentList<T> target) {
3      target.Clear();
4  }

```

■ **Figure 14** PUT (in `ConcurrentList` [7]) that should be improved with assertions.

■ **Table 9** Categorization results of bad smells in PUTs

Category	#PUTs
(1) Code duplication	55
(2) Unnecessary conditional statement	39
(3) Hard-coded test data	37
Total	131

5.3.1 PUT Design Patterns

We find that the majority of the PUTs are written in the following patterns: “AAA” (Triple-A) and Parameterized Stub. Triple-A is a well-known design pattern for writing unit tests [13]. These tests are organized into three sections: setting up the code under test (Arrange), exercising the code under test (Act), and verifying the behavior of the code under test (Assert). On the other hand, a Parameterized Stub test is used to test the code under test that already contains many assertions (e.g., code equipped with code contracts [25, 30, 16]). In general, Parameterized Stub tests are easy to write and understand, since the test body is short and contains only a few method calls to the code under test. In our subjects, we find that 34.6% (270/741) and 32.1% (251/741) of the PUTs to exhibit the Triple-A and Parameterized Stub test pattern, respectively. Of the 251 PUTs that exhibit the Parameterized Stub pattern, we find that 74.5% (187/251) of them are PUTs that should be improved with assertions, given that the code under test itself does not contain any code-contract assertions or any other type of assertions. For example, the PUT in Figure 14 contains only a single statement to test the robustness of the `Clear` method, which by itself does not contain any assertions. Developers of this PUT should at least add an assertion such as `Assert.That(target.Count, Is.EqualTo(0))`; to the end of the PUT to ensure that once `Clear` is invoked, then the number of elements in a `ConcurrentList` object will be 0.

Similar to the bad smells typically found in conventional unit tests [29], we consider the following three categories of bad smells in our PUTs: (1) code duplication, (2) unnecessary conditional statement, and (3) hard-coded test data. These three categories of bad smells can cause tests to be difficult to understand and maintain. Table 9 shows the number of PUTs containing each category of bad smells. Our analysis tools as described in Section 4.2 assist our manual inspections of the PUTs by listing the PUTs that contain conditional statements or hard-coded test data (as arbitrary strings). Using these lists of PUTs, we then manually inspect each of these PUTs to determine whether it really has bad code smells. To determine code duplication, we manually compare every PUT with every other PUT of the same class. Next, we discuss each of the categories in detail.

5.3.2 Code Duplication in PUTs

Similar to conventional unit tests, PUTs also contain the bad smell of test-code duplication. Test-code duplication is a poor practice because it increases the cost of maintaining tests. Duplication often arises when developers clone tests and do not put enough thought into how to reuse test logic intelligently. As the number of tests increases, it is important that the

```

1 [PexMethod]
2 public void GetItem(int index)
3 {
4     PexAssume.IsTrue(index >= 0);
5     const int element = 5;
6     for (int i = 0; i < index; i++)
7     {
8         _list.Add(0);
9     }
10    _list.Add(element);
11    Assert.That(_list[index], Is.EqualTo(element));
12 }

```

■ **Figure 15** PUT (from the `ConcurrentListHandWrittenTests` class of `ConcurrentList` [7]) that contains many lines of test-code duplication with another PUT named `SetItem` from the same class.

■ **Table 10** Categorization results of why conditional statements exist in PUTs.

Category	#PUTs
(1) Testing particular cases	16
(2) Forcing Pex to explore particular cases	9
(3) Testing different cases according to boolean conditions	9
(4) Unnecessary if statements	5
Total	39

developers either factor out commonly used sequences of statements into helper methods that can be reused by various tests, or in the case of PUTs, consider merging the PUTs and using assumptions/attributes to ensure that the specific cases being tested previously are still tested. In our subjects' PUTs, we find that 7.4% (55/741) of them contain test-code duplication. In other words, for 55 of our subjects' PUTs, there exist another PUT (in the same subject) that contains a significant amount of duplicate test code. One example of such PUT is shown in Figure 15. The PUT in this example is from the `ConcurrentListHandWrittenTests` class of `ConcurrentList` [7] and is almost identical to another PUT named `SetItem` in the same class. More specifically, the only lines that differ between the two PUTs are Lines 6 and 10. For Line 6 the loop terminating condition is set to `i <= index` as opposed to `i < index`. For Line 10, instead of adding an element with the `Add` method, the line is `_list[index] = element;`. In .NET, the use of brackets and an index value to add elements to a collection is enabled by `Indexers` [6]. Since the intention of the two PUTs is to test whether setting and getting an element from a list of arbitrary size correctly set and get the correct element, the two differences in Lines 6 and 10 between the two PUTs actually do not matter. Instead of duplicating so many lines of test code, the developers of these two PUTs should just delete one of them. Doing so will not only help decrease the cost for developers to maintain the tests, but also to speed up the testing time, since there will be fewer tests that cover the same parts of the code under test. Developers can also make use of existing tools for detecting code clones [18, 19] to automatically help detect code duplication in PUTs.

5.3.3 Unnecessary Conditional Statements in PUTs

Typically developers are expected not to write any conditional statements in their tests, because tests should be simple, linear sequences of statements. When a test has multiple execution paths, one cannot be sure exactly how the test will execute in a specific case. In our subjects, 7.0% (52/741) of the PUTs contain at least one conditional branch. To understand why developers write PUTs with conditionals, we study whether the conditionals

```

1 IList<int> _list = new ConcurrentList<int>();
2 [PexMethod(MaxBranches = 20000)]
3 public void Clear(int count)
4 {
5     var numClears = 100;
6     var results = new List<int>(numClears * 2);
7     var numCpus = Environment.ProcessorCount;
8     var sw = Stopwatch.StartNew();
9     using (SaneParallel.For(0, numCpus, x =>
10    {
11        for (var i = 0; i < count; i++)
12            _list.Add(i);
13    })
14    {
15        for (var i = 0; i < numClears; i++)
16        {
17            Thread.Sleep(100);
18            results.Add(_list.Count);
19            _list.Clear();
20            results.Add(_list.Count);
21        }
22    }
23    sw.Stop();
24    for (var i = 0; i < numClears; i++)
25        Console.WriteLine("Before/After Clear #{0}: {1}/{2}", i, results[i << 1], results[(i << 1) + 1]);
26    Console.WriteLine("ClearParallelSane took {0}ms", sw.ElapsedMilliseconds);
27    _list.Clear();
28    Assert.That(_list.Count, Is.EqualTo(0));
29 }

```

■ **Figure 16** PUT with hard-coded test data in the `SaneParallelTests` class of `ConcurrentList` [7].

in these PUTs are necessary and if they are not, why the developers write such conditionals in their PUTs. We find that 25% (13/52) of the PUTs contain conditional statements that could not be removed. These PUTs are typically testing the interactions of two or more operations of the code under test (e.g., adding and removing from a data structure). The remaining 75.0% (39/52) of the PUTs with conditionals can have their conditionals removed or each of these PUTs should be split into two or more PUTs. Table 10 shows the reasons for why the conditionals of such PUTs should be removed and the number of PUTs for each of the reasons. The PUTs in the first and second categories should replace their conditionals with `PexAssume()` statements to force Pex to explore and test particular cases. The PUTs in the third category should be each split into multiple PUTs each of which tests a different case of the conditional. For the PUTs created from the third category, developers can use `PexAssume()` statements in the new PUTs to filter out inputs that do not satisfy the boolean conditions of the case that the new PUTs are responsible for. The PUTs in the last category contain conditionals that can be removed with a slight modification to the test (e.g., some conditionals in a loop can be removed by amending the loop and/or adding code before the loop). The automatic detection and fixing of unnecessary conditional statements in PUTs would be a valuable and challenging line of future work due to the following. There are various reasons for why a PUT may have conditionals as shown in Table 10, and depending on the reason why a PUT may have conditionals, the fix for removing the conditionals, if removal is possible, can be quite different.

5.3.4 Hard-coded Test Data in PUTs

Another bad smell that we identify in our subjects' PUTs is hard-coded test data. This smell can be problematic for three main reasons. (1) Tests are more difficult to understand. A developer debugging the tests would need to look at the hard-coded data and deduce how each value is related to another and how these values affect the code under test. (2) Tests

are more likely to be flaky [28, 22, 15]. A common reason for tests to be flaky is the reliance on external dependencies such as databases, file system, and global variables. Hard-coded data in these tests often lead to multiple tests modifying the same external dependency and these modifications could cause these tests to fail unexpectedly. (3) Hard-coded test data prevent automatic test generation tools such as Pex from generating high-quality input values. In our subjects' PUTs, we find that 5.0% (37/741) of them use hard-coded test data. One example of such PUT is shown in Figure 16. In this example, the developers are testing the `Clear` method of the `ConcurrentList` object (`_list`). The PUT adds an arbitrary number of elements to the `_list` object, clears the list, and records the number of elements in the list. The process of adding and clearing the list repeats 100 times as decided by `numClears` on Line 5. As far as we can tell, the developers arbitrarily choose the value of 100 for `numClears` on Line 5. When we parameterize the `numClears` variable and add an assumption that the variable should be between 1 and 1073741823 (to prevent `ArgumentOutOfRangeException`), we find that the input values generated by Pex for the `numClears` variable to be 1 and 2. These two values exercise the same lines of the `Clear` method just as the value of 100 would. The important point here is that contrary to the developers' arbitrarily chosen value of 100, Pex is able to systematically find that using just the values of 1 and 2 would already sufficiently test the `Clear` method. That is, as we manually confirm, even if the developers devote more computation time to testing the `Clear` method by setting `numClears` to 100, they would not cover any additional code or find any additional test failures. Therefore, the developers of this PUT should not hard code the test data, and instead they should parameterize the `numClears` variable. Doing so would enable automatic test generation tools such as Pex to generate high-quality input values that sufficiently test the code under test. Developers can also make use of existing program analysis tools [41] to automatically detect whether certain hard-coded test data may exist between multiple PUTs.

5.3.5 Implications

By understanding how developers write PUTs, testing educators can suggest ways to improve PUTs. For example, developers should consider splitting PUTs with multiple conditional statements into separate PUTs each covering a case of the conditional statements. Doing so makes the developer's PUTs easier to understand and eases the effort to diagnose the reason for test failures. Tool vendors and researchers can incorporate this data with their tools to check the style of PUTs for better suggestions on how the PUTs can be improved. For example, checking whether a PUT is a Parameterized Stub, contains conditionals, contains hard-coded test data, and contains duplicate test code often correctly identifies a PUT that can be improved.

6 Threats to Validity

There are various threats to validity in our study. We broadly divide the main threats into internal and external validity.

6.1 Internal Validity

Threats to internal validity are concerned with the validity of our study procedure. Due to the complexity of software, faults in our analysis tools could have affected our results. However, our analysis tools are tested with a suite of unit tests, and samples of the results are manually verified. Results from our manual analyses are confirmed by at least two of the

authors. Furthermore, we rely on various other tools for our study, such as dotCover [8] to measure the code coverage of the input values generated by Pex. These tools could have faults as well and consequently such faults could have affected our results.

6.2 External Validity

There are two main threats to external validity in our study.

1. We use the categorization of the Microsoft MSDN Pex Forum posts [31] to determine the issues surrounding parameterized unit testing. These forum posts enable us and the research community to access the issues of developers objectively and quantitatively, but the issues identified from the posts may not be representative of all the issues that developers encounter.
2. Our findings may not apply to subjects other than those that we study, especially since we are able to find only 11 subjects matching the criteria defined in Section 4. Furthermore, we primarily focus on projects using PUTs in the context of automated test generation, so our findings from such subjects may not generalize to situations outside of this setting (e.g., general usage of Theories [33] in Java). In addition, our analyses focus specifically on subjects that contain PUTs written using the Pex framework, and the API differences or idiosyncrasies of other frameworks may impact the applicability of our findings. All of our subjects are written in C#, but vary widely in their application domains and project sizes. Finally, all of our subjects are open source software, and therefore our findings may not generalize to proprietary software.

7 Related Work

To the best of our knowledge, our characteristic study is the first on parameterized unit testing in open source projects. In contrast, previous work focuses on proposing new techniques for parameterized unit testing and does not provide any insight on the practices of parameterized unit testing. For example, Xie et al. [43] propose a technique for assessing the quality of PUTs using mutation testing. Thummalapenta et al. [36] propose manual retrofitting of CUTs to PUTs, and show that new faults are detected and coverage is increased after such manual retrofitting is conducted. Fraser et al. [21] propose a technique for generating PUTs starting from concrete test inputs and results.

Our work is related to previous work on studying developer-written formal specifications such as code contracts [16]. Schiller et al. [34] conduct case studies on the use of code contracts in open source projects in C#. They analyze 90 projects using code contracts and categorize their use of various types of specifications, such as null checks, bound checks, and emptiness checks. They find that checks for nullity and emptiness are the most common types of specifications. Similarly we find that the most common types of PUT assumptions are also used for nullness specification. However, the most common types of PUT assertions are used for equality checking instead of null and emptiness.

Estler et al. [20] study code contract usage in 21 open source projects using JML [27] in Java, Design By Contract in Eiffel [30], and code contracts [16] in C#. Their study also includes an analysis of the change in code contracts over time, relative to the change in the specified source code. Their findings agree with Schiller's on the majority use of nullness code contracts. Furthermore, Chalin [17] studies code contract usage in over 80 Eiffel projects. They show that programmers using Eiffel tend to write more assertions than programmers using any other languages do.

8 Conclusion

To fill the gap of lacking studies of PUTs in development practices of either proprietary or open source software, we have presented categorization results of the Microsoft MSDN Pex Forum posts (contributed primarily by industrial practitioners) related to PUTs. We then use the categorization results to guide the design of the first characteristic study of parameterized unit testing in open source projects. Our study involves hundreds of PUTs that open source developers write for various open source projects.

Our study findings provide the following valuable insights for various stakeholders such as current or prospective PUT writers (e.g., developers), PUT framework designers, test-generation tool vendors, testing researchers, and testing educators.

1. We have studied the extents and types of assumptions, assertions, and attributes being used in PUTs. Our study has identified assumption and assertion types that tool vendors or researchers can incorporate with their tools to better infer assumptions and assertions to assist developers. For example, tool vendors or researchers who care about the most commonly used assumption types should focus on `PexAssumeUnderTest` or `PexAssumeNotNull`, since these two are the most commonly used assumption types. We have also found that increasing the default value of attributes as suggested by tools such as Pex rarely contributes to increased code coverage. Tool vendors or researchers should aim to improve the quality of the attribute recommendations provided by their tools, if any are provided at all.
2. We have studied how often hard-coded method sequences in PUTs can be replaced with non-primitive parameters and how useful it is for developers to do so. Our study has found that there are a significant number of receiver objects in the PUTs written by developers that could be promoted to non-primitive parameters, and a significant number of existing non-primitive parameters that lack factory methods. Tool researchers or vendors should provide effective tool support to assist developers to promote these receiver objects (resulted from hard-coded method sequences), e.g., inferring assumptions for a non-primitive parameter promoted from hard-coded method sequences. Additionally, once hard-coded method sequences are promoted to non-primitive parameters, developers can also use assistance in writing effective factory methods for such parameters.
3. We have studied the common design patterns and bad smells in PUTs, and have found that there are a number of patterns that often correctly identify a PUT that can be improved. More specifically, checking whether a PUT is a Parameterized Stub, contains conditionals, contains hard-coded test data, and contains duplicate test code often correctly identifies a PUT that can be improved. Tool vendors and researchers can incorporate this data with their tools to check the style of PUTs for better suggestions on how these PUTs can be improved.

The study is part of our ongoing industry-academia team efforts for bringing parameterized unit testing to broad industrial practices of software development.

References

- 1 Atom. URL: <https://github.com/tivtag/Atom>.
- 2 PUT study project web. URL: <https://sites.google.com/site/putstudy>.
- 3 QuickGraph. URL: <https://github.com/tathanhdinh/QuickGraph>.
- 4 SearchCode code search. URL: <https://searchcode.com>.
- 5 Theories in JUnit. URL: <https://github.com/junit-team/junit/wiki/Theories>.
- 6 Using Indexers (C# Programming Guide). URL: <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/indexers/using-indexers>.

- 7 ConcurrentList. URL: <https://github.com/damageboy/ConcurrentList>.
- 8 dotCover. URL: <https://www.jetbrains.com/dotcover>.
- 9 GitHub code search. URL: <https://github.com/search>.
- 10 The .NET compiler platform Roslyn. URL: <https://github.com/dotnet/roslyn>.
- 11 NUnit Console. URL: <https://github.com/nunit/nunit-console>.
- 12 OpenMhcg. URL: <https://github.com/orryverducci/openmhcg>.
- 13 Parameterized Test Patterns for Microsoft Pex). URL: <http://citeseerx.ist.psu.edu/viewdoc/download?rep=rep1&type=pdf&doi=10.1.1.216.282>.
- 14 Parameterized tests in JUnit. URL: <https://github.com/junit-team/junit/wiki/Parameterized-tests>.
- 15 Stephan Arlt, Tobias Morciniec, Andreas Podelski, and Silke Wagner. If A fails, can B still succeed? Inferring dependencies between test results in automotive system testing. In *ICST 2015: Proceedings of the 8th International Conference on Software Testing, Verification and Validation*, pages 1–10, Graz, Austria, apr 2015.
- 16 Michael Barnett, Manuel Fähndrich, Peli de Halleux, Francesco Logozzo, and Nikolai Tillmann. Exploiting the synergy between automated-test-generation and programming-by-contract. In *ICSE 2009: Proceedings of the 31st International Conference on Software Engineering*, pages 401–402, Vancouver, BC, Canada, may 2009.
- 17 Patrice Chalin. Are practitioners writing contracts? In *Rigorous Development of Complex Fault-Tolerant Systems*, pages 100–113. Springer, 2006.
- 18 Yingnong Dang, Dongmei Zhang, Song Ge, Chengyun Chu, Yingjun Qiu, and Tao Xie. XIAO: Tuning code clones at hands of engineers in practice. In *ACSAC 2012: Proceedings of 28th Annual Computer Security Applications Conference*, pages 369–378, Orlando, FL, USA, December 2012.
- 19 Yingnong Dang, Dongmei Zhang, Song Ge, Ray Huang, Chengyun Chu, and Tao Xie. Transferring code-clone detection and analysis to practice. In *ICSE 2017: Proceedings of the 39th International Conference on Software Engineering, Software Engineering in Practice (SEIP)*, pages 53–62, Buenos Aires, Argentina, May 2017.
- 20 H-Christian Estler, Carlo A Furia, Martin Nordio, Marco Piccioni, and Bertrand Meyer. Contracts in practice. In *FM 2014: Proceedings of the 19th International Symposium on Formal Methods*, pages 230–246. Springer, Singapore, 2014.
- 21 Gordon Fraser and Andreas Zeller. Generating parameterized unit tests. In *ISSTA 2011: Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pages 364–374, Toronto, ON, Canada, jul 2011.
- 22 Zebao Gao, Yalan Liang, Myra B. Cohen, Atif M. Memon, and Zhen Wang. Making system user interactive tests repeatable: When and what should we control? In *ICSE 2015: Proceedings of the 37th International Conference on Software Engineering*, pages 55–65, Florence, Italy, may 2015.
- 23 Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed automated random testing. In *PLDI 2005: Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, Chicago, IL, USA, jun 2005.
- 24 John V. Guttag and James J. Horning. The algebraic specification of abstract data types. *Acta Informatica*, pages 27–52, 1978.
- 25 C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, pages 576–580, 1969.
- 26 Pratap Lakshman. Visual Studio 2015 – Build better software with Smart Unit Tests. *MSDN Magazine*, 2015.
- 27 Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report TR 98-06i, Department of Computer Science, Iowa State University, Jun 1998.

- 28 Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. An empirical analysis of flaky tests. In *FSE 2014: Proceedings of the ACM SIGSOFT 22nd Symposium on the Foundations of Software Engineering*, pages 643–653, Hong Kong, nov 2014.
- 29 Gerard Meszaros. *XUnit Test Patterns: Refactoring Test Code*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2006.
- 30 Bertrand Meyer. Applying "Design by Contract". *Computer*, pages 40–51, oct 1992.
- 31 Microsoft. Pex MSDN discussion forum, April 2011. URL: <http://social.msdn.microsoft.com/Forums/en-US/pex>.
- 32 Microsoft. Generate unit tests for your code with IntelliTest, 2015. URL: <https://msdn.microsoft.com/library/dn823749>.
- 33 David Saff. Theory-infected: Or how I learned to stop worrying and love universal quantification. In *OOPSLA Companion: Proceedings of the Object-Oriented Programming Systems, Languages, and Applications*, pages 846–847, Montreal, QC, Canada, oct 2007.
- 34 Todd W Schiller, Kellen Donohue, Forrest Coward, and Michael D Ernst. Case studies and tools for contract specifications. In *ICSE 2014: Proceedings of the 36th International Conference on Software Engineering*, pages 596–607, Hyderabad, India, jun 2014.
- 35 Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A concolic unit testing engine for C. In *ESEC/FSE 2005: Proceedings of the 10th European Software Engineering Conference and the 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 263–272, Lisbon, Portugal, sep 2005.
- 36 Suresh Thummalapenta, Madhuri R Marri, Tao Xie, Nikolai Tillmann, and Jonathan de Halleux. Retrofitting unit tests for parameterized unit testing. In *FASE 2011: Proceedings of the Fundamental Approaches to Software Engineering*, pages 294–309. Springer, Saarbrücken, Germany, mar 2011.
- 37 Nikolai Tillmann and Jonathan De Halleux. Pex: White box test generation for .NET. In *TAP 2008: Proceedings of the 2nd International Conference on Tests And Proofs (TAP)*, pages 134–153, Prato, Italy, apr 2008.
- 38 Nikolai Tillmann, Jonathan de Halleux, and Tao Xie. Parameterized unit testing: Theory and practice. In *ICSE 2010: Proceedings of the 32nd International Conference on Software Engineering*, pages 483–484, Cape Town, South Africa, may 2010.
- 39 Nikolai Tillmann, Jonathan de Halleux, and Tao Xie. Transferring an automated test generation tool to practice: From Pex to Fakes and Code Digger. In *ASE 2014: Proceedings of the 29th Annual International Conference on Automated Software Engineering*, pages 385–396, Västerås, Sweden, sep 2014.
- 40 Nikolai Tillmann and Wolfram Schulte. Parameterized unit tests. In *ESEC/FSE 2005: Proceedings of the 10th European Software Engineering Conference and the 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 253–262, Lisbon, Portugal, 2005.
- 41 Matias Waterloo, Suzette Person, and Sebastian Elbaum. Test analysis: Searching for faults in tests. In *ASE 2015: Proceedings of the 30th Annual International Conference on Automated Software Engineering*, pages 149–154, Lincoln, NE, USA, nov 2015.
- 42 Xusheng Xiao, Tao Xie, Nikolai Tillmann, and Jonathan de Halleux. Precise identification of problems for structural test generation. In *ICSE 2011: Proceedings of the 33rd International Conference on Software Engineering*, pages 611–620, Waikiki, HI, USA, may 2011.
- 43 Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Wolfram Schulte. Mutation analysis of parameterized unit tests. In *ICSTW 2009: Proceedings of the International Conference on Software Testing, Verification and Validation Workshops*, pages 177–181, Denver, CO, USA, 2009.

Learning to Accelerate Symbolic Execution via Code Transformation

Junjie Chen

Key Laboratory of High Confidence Software Technologies (Peking University), MoE
Institute of Software, EECS, Peking University, Beijing, 100871, China
chenjunjie@pku.edu.cn

Wenxiang Hu

Key Laboratory of High Confidence Software Technologies (Peking University), MoE
Institute of Software, EECS, Peking University, Beijing, 100871, China
huwx@pku.edu.cn

Lingming Zhang

Department of Computer Science, University of Texas at Dallas, 75080, USA
lingming.zhang@utdallas.edu

Dan Hao¹

Key Laboratory of High Confidence Software Technologies (Peking University), MoE
Institute of Software, EECS, Peking University, Beijing, 100871, China
haodan@pku.edu.cn

Sarfraz Khurshid

Department of Electrical and Computer Engineering, University of Texas at Austin, 78712, USA
khurshid@ece.utexas.edu

Lu Zhang

Key Laboratory of High Confidence Software Technologies (Peking University), MoE
Institute of Software, EECS, Peking University, Beijing, 100871, China
zhanglucs@pku.edu.cn

Abstract

Symbolic execution is an effective but expensive technique for automated test generation. Over the years, a large number of refined symbolic execution techniques have been proposed to improve its efficiency. However, the symbolic execution efficiency problem remains, and largely limits the application of symbolic execution in practice. Orthogonal to refined symbolic execution, in this paper we propose to accelerate symbolic execution through semantic-preserving code transformation on the target programs. During the initial stage of this direction, we adopt a particular code transformation, compiler optimization, which is initially proposed to accelerate program concrete execution by transforming the source program into another semantic-preserving target program with increased efficiency (e.g., faster or smaller). However, compiler optimizations are mostly designed to accelerate program *concrete* execution rather than *symbolic* execution. Recent work also reported that unified settings on compiler optimizations that can accelerate symbolic execution for any program do not exist at all. Therefore, in this work we propose a machine-learning based approach to tuning compiler optimizations to accelerate symbolic execution, whose results may also aid further design of specific code transformations for symbolic execution. In particular, the proposed approach LEO separates source-code functions and libraries through our program-splitter, and predicts individual compiler optimization (i.e., whether a type of code transformation is chosen) separately through analyzing the performance of existing symbolic execution. Finally, LEO applies symbolic execution on the code transformed by

¹ corresponding author.



compiler optimization (through our local-optimizer). We conduct an empirical study on GNU Coreutils programs using the KLEE symbolic execution engine. The results show that LEO significantly accelerates symbolic execution, outperforming the default KLEE configurations (i.e., turning on/off all compiler optimizations) in various settings, e.g., with the default training/testing time, LEO achieves the highest line coverage in 50/68 programs, and its average improvement rate on all programs is 46.48%/88.92% in terms of line coverage compared with turning on/off all compiler optimizations.

2012 ACM Subject Classification Software and its engineering → Software testing and debugging

Keywords and phrases Symbolic Execution, Code Transformation, Machine Learning

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2018.6

Funding This work is partially supported by the National Key Research and Development Program of China (Grant No. 2017YFB1001803), and NSFC 61672047, 61529201, 61522201, 61861130363. This work is also partially supported by NSF Grant No. CCF-1566589, UT Dallas start-up fund, Google, Huawei, and Samsung. This work is also partially supported by NSF Grant No. CCF-1704790.

1 Introduction

Symbolic execution is a systematic analysis methodology to explore program behaviors, and has been widely used in test input generation [29, 15, 30]. In particular, symbolic execution takes test inputs as symbolic values instead of concrete values so as to generate test inputs by solving the constraints for program paths. Although symbolic execution facilitates test generation to a large extent, it is widely recognized to suffer from the efficiency problem due to the exponential number of paths and constraint solving cost. To relieve the efficiency problem of symbolic execution, various optimization techniques have been proposed, e.g., compositional symbolic execution [41, 72], incremental symbolic execution [79, 88], and parallel symbolic execution [78, 76]. However, symbolic execution remains one of the most expensive testing methodologies [18].

Instead of refining symbolic execution techniques, in this paper, we aim to accelerate symbolic execution via another orthogonal dimension – transforming the programs under test. Intuitively, if a program under test can be transformed into a semantic-preserving but easy-to-analyze program, the efficiency of symbolic execution will be improved. Moreover, all the refined symbolic execution techniques will be also further improved because of the orthogonality. That is, semantic-preserving code transformation rules for symbolic execution are needed. However, few semantic-preserving code transformation rules studied in the literature targets at symbolic execution, and designing such rules is a complex process and will be a long-term project. During the initial stage of this direction, we borrow code transformation rules for concrete execution to learn code transformation rules for symbolic execution, because of the substantial knowledge accumulated over 30 years in the field of concrete execution as well as the similarity between concrete execution and symbolic execution. In particular, we borrow compiler optimization, which is one of the most mature code transformation approaches to transforming the source program into another semantic-preserving target program with increased efficiency (e.g., faster or smaller) and has been widely recognized by its effectiveness on accelerating *concrete* execution [2, 32, 38, 27].

Since compiler optimizations are specially designed for compilers to optimize program *concrete* execution, they may reduce the efficiency of *symbolic* execution due to the difference between concrete and symbolic execution. As reported by recent work [33, 14], some compiler optimizations indeed largely accelerate symbolic execution for some programs, but some compiler optimizations even make symbolic execution much slower for some programs. Moreover, there is no unified configuration on the compiler optimizations guaranteeing the efficiency of symbolic execution for all programs. If we can learn how to utilize compiler optimizations to accelerate symbolic execution for each individual program, it will become a very light-weight approach to accelerating symbolic execution via code transformation, and is also helpful in designing specific effective code transformation rules for symbolic execution. Therefore, in this paper we focus on *learning to tune compiler optimizations* to accelerate symbolic execution.

In particular, we propose the first machine-LEarning-based approach to tuning compiler Optimizations for symbolic execution (abbreviated as **LEO**). LEO tunes compiler optimizations for each code portion (e.g., each function) of a program individually rather than for the whole program, because compiler optimizations transform different code portions in different ways. More specifically, for any program under test, LEO first divides it into source-code portions and libraries used in the program, and then learns their settings on compiler optimizations separately. Library optimizations can be directly applied with the corresponding compiler. To enable different code portions with different settings on compiler optimizations, we design and implement two components. The first one is program-splitter, which splits a program into multiple files so that each file contains only one source-code portion (e.g., function). The second one is local-optimizer, which optimizes each preceding file by its learnt compiler optimization settings. With these tools, LEO integrates the optimized files and optimized libraries into a fine-optimized program using the LLVM linker. Such fine-optimized program is semantically equivalent with the original program, and is treated as inputs of symbolic execution engines instead of the original one, so as to accelerate symbolic execution.

To evaluate LEO, we conduct an empirical study on KLEE using the widely used GNU Coreutils programs [57, 86, 33, 15]. Our experimental study shows that compared with two default settings of KLEE (i.e., symbolic execution without any code transformations – turning off all compiler optimizations, and symbolic execution turning on all compiler optimizations), LEO achieves the highest line coverage in 50/68 programs, indicating its great performance on accelerating symbolic execution. In particular, compared with symbolic execution without any code transformations (i.e., turning off all compiler optimizations), the average improvement rate of LEO on all programs is 88.92% in terms of line coverage, demonstrating that code transformation is indeed a promising direction to accelerate symbolic execution. Moreover, compared with symbolic execution turning on all compiler optimizations, the average improvement rate of LEO on all programs is 46.48% in terms of line coverage, indicating that effectively tuning compiler optimizations is a successful exploration in this direction and our machine-learning based approach is able to predict better compiler-optimization settings for accelerating symbolic execution. Furthermore, the compiler optimizations recommended by LEO with some specified training symbolic execution time (e.g., the default 10-minute) can always significantly outperform the default settings of KLEE in most cases even when the testing symbolic execution time increases.


```

1 int fun2(int N, int h[10]){
2   int i;
3 ---for(i=0;i<N-2;++i){
4 ---  if(i%2==0) h[i] = 1;
5 ---  else h[i]=0;
6 ---}
7   for(i=0;i<N;++i) {
8     h[i]=2*i;
9   }
10  int sum=0;
11  for(i=0;i<N;++i)
12    sum+=h[i];
13  return sum;
14 }

```

(a) Acceleration.

```

1 int fun1(int M, int g[10]) {
2   int i;
3 ---for(i=0;i<M;++i){
4   +++for(i=M-2;i<M;++i) {
5     g[i]=i*i;
6   }
7   for(i=0;i<M-2;++i) {
8     g[i]=0;
9   }
10  int sum=0;
11  for(i=0;i<M;++i)
12    sum+=g[i];
13  return sum;
14 }

```

(b) Deceleration.

■ **Figure 1** Motivating examples.

The contributions of this paper are summarized as follows.

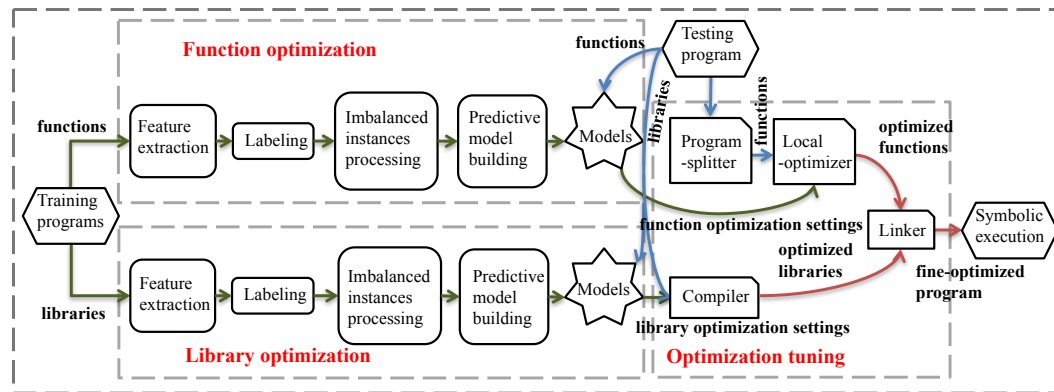
- The first approach to accelerating symbolic execution via machine-learning based compiler optimization tuning for code transformation.
- An implementation of the proposed approach, including program-splitter and local-optimizer components, enabling the learnt compiler optimization settings for different code portions.
- An extensive study on GNU Coreutils programs demonstrating the performance of LEO on accelerating symbolic execution as well as the contributions of various components of LEO.

2 Motivation

In this section, we use two examples of aggressive dead code elimination (ADCE) to illustrate the motivation of this work, i.e., tuning compiler optimization can accelerate symbolic execution. ADCE is a compiler optimization that assumes all instructions are dead unless they are proven not and tries to eliminate dead statements within loop computations. This optimization can accelerate program concrete execution but has different impacts on symbolic execution. The first example is shown in Figure 1a, where the code with marks is the code transformed through the compiler optimization. The transformation removes the first redundant loop as marked, and accordingly simplifies path conditions, which facilitates symbolic execution. As a result, symbolic execution after optimization requires only 11 queries², while symbolic execution before optimization requires 54 queries.

Figure 1b presents another example on ADCE. Contradictory to the observation in Figure 1a, the optimization used in Figure 1b decelerates symbolic execution. More specifically, the transformation tries to avoid redundant computations by complicating the starting condition of the first loop (i.e., at Line 3). That is, turning on this compiler optimization increases the complexity of the path conditions, which enhances the difficulty of constraint solving in symbolic execution. As a result, symbolic execution before optimization requires

² Query is a concept of SMT constraint solving. More queries tend to decrease the efficiency of symbolic execution.



■ **Figure 2** Overview of LEO.

48 queries, while it requires 107 queries after optimization, significantly aggravating the efficiency problem of symbolic execution. Combining the observations from Figures 1a and 1b, a compiler optimization can behave differently, e.g., accelerate or decelerate symbolic execution, making it not proper to give a unified compiler optimization setting for all programs. Therefore, this paper targets learning how to tune these compiler optimizations for each individual program to accelerate symbolic execution via code transformation.

From Figures 1a and 1b, the transformation performed by compiler optimizations actually occurs on some code portions rather than the whole program. For example, the transformation in Figure 1a occurs at Lines 3-6, and the transformation in Figure 1b occurs at Lines 3. That is, the transformation actually occurs at fine granularities (e.g., statements and functions) rather than at coarse granularities (e.g., the whole program). If a compiler optimization is uniformly set at coarse granularities, it is hard to guarantee the efficiency of symbolic execution. For example, if a large program consists of the two functions in Figure 1a and Figure 1b, it is hard to tell whether the optimization, ADCE, accelerates the symbolic execution of the whole program because such an optimization has opposite influence on the two functions. That is, to accelerate symbolic execution, compiler optimizations should be tuned at fine granularities, e.g., the function level, rather than at coarse granularities. On the other side, it is costly to tune compiler optimizations at much finer granularities (e.g., the statement level) due to the extra efforts on compiler optimization tuning. Therefore, *in this paper, we use the function level as a compromise and tune compiler optimizations for symbolic execution at the **function** level.*

3 Approach

To accelerate symbolic execution via code transformation, we propose the first approach to tuning compiler optimizations at the function level based on machine learning. The key insight of our approach is that program code portions with certain features (e.g., structure or complexity features) or combinations of features are inherently more likely to be transformed to easy-to-analyze programs by certain compiler optimizations. Besides the implemented source functions, a program may use API functions of some libraries, and thus it is necessary to learn how to set compiler optimizations for these libraries as well. However, the libraries are usually so large that splitting libraries into functions and tuning compiler optimizations for each library function incur huge costs, and thus LEO predicts the settings of compiler optimizations for libraries in a way different from what it does for functions. That is, LEO

divides a program into source-code functions and libraries, and predicts their settings of compiler optimizations separately.

Figure 2 presents the overview of LEO. It first trains a predictive model for each optimization to predict whether the compiler optimization should be turned on for a function (see Section 3.1), and then trains a predictive model for each compiler optimization to predict whether the compiler optimization is turned on for libraries (see Section 3.2). Based on the prediction results, LEO tunes the settings of compiler optimizations for the program under test, and implements the program-splitter and local-optimizer components to facilitate compiler optimization settings for each code portion (see Section 3.3). Note that although the general idea of LEO applies to various symbolic execution engines and compilers, in this work, we present LEO based on the KLEE symbolic execution engine [15] and its underlying LLVM compiler infrastructure [55].

3.1 Function Optimization

In function optimization, LEO first collects a set of training instances from source functions by extracting their features and identifying their labels, and then builds a predictive model based on these training data for each compiler optimization.

3.1.1 Feature Extraction

To predict whether a compiler optimization can facilitate symbolic execution for a function, the identified features from source functions should characterize how compiler optimizations influence the efficiency of symbolic execution. Therefore, we identify features from two aspects: path exploration and constraint solving, which are main reasons for the efficiency problem of symbolic execution [18]. From the aspect of path exploration, we use a group of features relevant to *program structure*, e.g., the number of basic blocks with one/two/more than two successor(s), the number of edges in the control flow graph, and the number of conditional branches. From the aspect of constraint solving, we use a group of features relevant to *program complexity*, e.g., the number of references (def/use) of static/extern/local variables, the number of instructions that do pointer arithmetic, and the number of indirect references via pointers. In particular, prior work on compiler optimizations for program concrete execution [38] has already recognized some characteristics of a program that are related to compiler optimizations. Here we use all these characteristics as the features of LEO because these features are relevant to either path exploration or constraint solving. Details about our features can be found in the homepage of LEO.

For each function, LEO extracts the values of these features, which are represented by a vector whose elements are numeric. As these features may have different value ranges, LEO normalizes each element's value into the range [0,1] using the min-max normalization [48] so as to adjust values measured on different scales into a common scale. Supposed that the set of training instances (i.e., functions) is denoted as $P = \{p_1, p_2, \dots, p_n\}$, the set of vectors extracted from P is denoted as $V = \{v_1, v_2, \dots, v_n\}$, the set of elements in a vector is denoted as $E = \{e_1, e_2, \dots, e_m\}$, and the value of the element e_j in the vector v_i before normalization is denoted as x_{ij} , then the value of the element e_j in the vector v_i after normalization is denoted as x_{ij}^* , Formula 1 presents the min-max normalization on x_{ij} , where $1 \leq i \leq n$ and $1 \leq j \leq m$.

$$x_{ij}^* = \frac{x_{ij} - \min(\{x_{kj} | 1 \leq k \leq n\})}{\max(\{x_{kj} | 1 \leq k \leq n\}) - \min(\{x_{kj} | 1 \leq k \leq n\})} \quad (1)$$

3.1.2 Labeling

LEO is designed to build a predictive model for a compiler optimization, characterizing how to accelerate symbolic execution through the compiler optimization setting. Therefore, the label for each training instance is defined as the setting of a compiler optimization that accelerates symbolic execution. In other words, a label of a training instance (i.e., function) refers to whether a compiler optimization should be turned on or off.

For any training instance (i.e., function), LEO labels based on the comparison between its symbolic execution efficiency with the compiler optimization turned on and turned off. Same as existing work [33, 15, 87], symbolic execution efficiency is measured by line coverage achieved by the generated test inputs within time limit. That is, within time limit, if line coverage achieved when turning on this compiler optimization is higher than that when turning off it, the instance label for this compiler optimization is “turning on”. Otherwise, the label is “turning off”.

It is hard to learn whether a compiler optimization should be turned on or off for a *function*, since symbolic execution takes the whole program rather than each function as input. To relieve this issue, LEO estimates the label of each function by analyzing the line coverage of the whole program instead. More specifically, LEO first collects line coverage of the whole program, i.e., which line of code is covered by the test inputs generated through symbolic execution, then determines the line coverage of each function by analyzing the distribution of line coverage. Finally, for each function, LEO compares its line coverage between symbolic execution with the optimization on and that with the optimization off to set the label.

3.1.3 Imbalanced Instance Processing

Through the steps introduced by Sections 3.1.1 and 3.1.2, we collect a set of training instances with features and labels. Based on the prior work [33], some compiler optimizations help accelerate symbolic execution in most cases but some other compiler optimizations make symbolic execution slower in most cases. That is, for a compiler optimization, its number of training instances whose labels are turned on may be greatly different from its number of training instances whose labels are turned off, which can incur the imbalanced data problem.

As the imbalanced problem may have serious impact on the accuracy of classification [23, 21], LEO uses over-sampling strategy to relieve the impact of imbalanced instances in optimization prediction. Here we choose over-sampling strategy rather than other strategies (e.g., under-sampling) because it is costly to collect a large number of training instances³. In particular, LEO uses SMOTE [22], because SMOTE over-samples the minority class by creating synthetic examples rather than by over-sampling with replacement [22]. More specifically, for each instance in the minority class, SMOTE creates synthetic examples along the line segments joining the instance and its k nearest neighbors by regarding all instances as points in space. Based on the amount of over-sampling required, neighbors are randomly chosen from the k nearest neighbors, and then one instance is created on each line segment.

³ Collecting a training instance requires feature extraction and labeling. Moreover, to label each instance, each program has to be executed twice in symbolic execution, including turning on the compiler optimization and turning off the compiler optimization.

3.1.4 Predictive Model

For each compiler optimization, LEO builds a predictive model through machine learning. In particular, LEO adopts the SMO algorithm, which is used to solve the quadratic programming problem in the training of Support Vector Machines (abbreviated as SVM) [69] and regarded as the fastest for linear SVM and sparse data sets. Note that although LEO is implemented based on SMO, it is not specific to this machine learning algorithm and we investigate the impact of machine learning algorithms in Section 4.6.4.

3.2 Library Optimization

As the libraries are usually so large that splitting libraries into functions and tuning compiler optimizations for each library function incur huge cost. Therefore, we predict compiler optimizations for libraries in a different way. More specifically, LEO regards the functions of libraries as a whole by building a predictive model for libraries used in a program (rather than each function). Similar to function optimization prediction, LEO predicts compiler optimizations for libraries as follows.

First, LEO defines a set of new features that characterize how compiler optimizations accelerate symbolic execution for libraries. Since library functions are relatively fixed in implementation and repeatedly used by various client code, it is not necessary to collect detailed features about each library function separately. Instead, knowing how compiler optimizations impact programs that used a library function before, can help predict how compiler optimizations impact the current program using that library function. Therefore, LEO directly uses whether each individual library function is called by a program as features of library optimization. That is, for each training instance (i.e., a program), LEO identifies the called library functions and uses 1/0 to represent a library function is/isn't called. As the values of these features are all 0 or 1, normalization is not necessary.

Second, LEO labels each training instance. An instance label is whether a compiler optimization should be turned on or off for the libraries used in a program. Similar to the process of function optimization prediction, LEO determines a label by comparing the line coverage of the whole program achieved when turning on the compiler optimization and that achieved when turning off the compiler optimization within time limit⁴.

Finally, based on the collected training data, LEO builds a predictive model for each compiler optimization using also SMO. Note that LEO also uses SMOTE to filter the impact of the imbalanced problem in library optimization prediction.

3.3 Optimization Tuning

Following Sections 3.1 and 3.2, LEO learns the settings of all compiler optimizations for a program, including each source-code function and the related libraries. However, as symbolic execution engines do not support various settings on different source-code functions of a program, LEO provides such fine-granularity optimization tuning by implementing two components: program-splitter and local-optimizer.

LEO first adopts the learnt settings of compiler optimizations for libraries by compiling the libraries individually. Then LEO splits the whole program into multiple files, each of which is only a function of the program, and adopts the learnt settings of compiler

⁴ As the features of a training instance are directly related to libraries and optimized libraries also contribute to the line coverage of the program, LEO approximately uses whether the line coverage of the program is improved when turning on a compiler optimization as the label for library optimization prediction, to save the cost of labeling.

optimizations for each file (i.e., function). Finally, LEO integrates the multiple optimized files and optimized libraries into a fine-optimized program using the LLVM linker, and analyzes this program rather than the original target program through symbolic execution. Due to program complexity, implementing the program-splitter and local-optimizer is an important technical challenge in LEO. In the following subsections, we first present the details on how to split a program into multiple function-level files in Section 3.3.1, and then present the details on how to optimize these files using learnt settings and integrate these optimized files and libraries in Section 3.3.2.

3.3.1 Program-Splitter

For any given program denoted as P_A , LEO splits it into function-level files $P_B = \{p_{b1}, p_{b2}, \dots, p_{bn}\}$, where p_{bk} refers to a function-level file ($1 \leq k \leq n$ and n is the total number of functions in P_A), via two stages: preprocessing stage and splitting stage. In the preprocessing stage, our approach preprocesses P_A and prepares the necessary materials, and in the splitting stage our approach splits P_A into function-level files based on these materials.

In the preprocessing stage, LEO first expands macro and removes comments to expediently transform P_A to P_B , and then prepares the following materials for the splitting stage:

- A common-symbol table, which contains the symbols of all common variables and functions in P_B ⁵, so as to solve the duplicate-name issue in link-time.
- A type-definition table, which records all definitions of existing types (e.g., structs) in P_B .
- A dependent table for each function in P_B , which records the declarations of its dependent functions and global variables.

In the splitting stage, our approach generates an individual file (denoted as p_{bk}) for each function (denoted as M_k) in P_B by the following steps:

- Putting the declarations of dependent functions and global variables into p_{bk} and using “extern” as their modifier, based on the dependent table of this function;
- Modifying the scope of the dependent functions and global variables, i.e., removing the “static” modifier, so that they can be used by the other files;
- Putting M_k into the file p_{bk} ;
- Putting all needed type definitions into p_{bk} referring to the type-definition table, based on all declarations in p_{bk} .

In particular, our approach records all global variables in an individual file so that all other files can use them.

3.3.2 Local-Optimizer

The optimizer of the KLEE symbolic execution engine applies all compiler optimizations together, but does not allow to turn on one compiler optimization or a subset of compiler optimizations. Therefore, we implement a local-optimizer by setting an interface that appoints which compiler optimizations are turned on. That is, we regard the names of compiler optimizations as parameters that are passed to the optimizer by the interface. Finally, LEO integrates all optimized files and libraries into a fine-optimized program using

⁵ In this paper, common variables and functions refer to the global variables and functions without “static” modifier.

the LLVM linker. When linking libraries, some symbols may have duplicate names, which will incur link errors. To solve this problem, our approach utilizes the common-symbol table generated in the program-splitter to remove the symbols whose scope is the current file from the symbol table of the executable.

4 Experimental Study

Our study addresses the following four research questions:

- **RQ1:** How does LEO perform on accelerating symbolic execution via code transformation?
- **RQ2:** How do different training and testing time limits impact LEO?
- **RQ3:** Do both function optimization and library optimization contribute to LEO?
- **RQ4:** How do different machine learning algorithms impact LEO?

Note that in our study, there are two types of time limits: training time limit and testing time limit. The former refers to the symbolic-execution time used for collecting training instances, and the latter refers to the symbolic-execution time used to analyze the programs under test.

4.1 Tools and Libraries

In our study, we use KLEE [15], one of the most widely used symbolic execution engines [15, 33, 87, 57]. KLEE is implemented using C++ based on the LLVM infrastructure, whose compiler provides dozens of compiler optimizations. The same as prior work [87, 33], we build KLEE with LLVM 2.9, which has 30 compiler optimizations integrated by KLEE. These optimizations are turned on through the command “-optimize” and turned off through the command “-disable-opt”, which are two default configurations of KLEE. In our study, we use similar KLEE options as the prior work [15]. Following the prior work [33, 30], we use the DFS search heuristic in KLEE so as to acquire more deterministic results, and disable the caching of KLEE since the caching contents can be different for different strategies, making it hard to check the actual impacts of different strategies. More discussion on the impact of search heuristics and caching can be found in Section 5.2.

We implement LEO’s machine-learning component by using the SMO algorithm provided by Weka 3.6.12⁶, whose *Puk* kernel is set with $\omega = 3$ and $\sigma = 1$ in this study based on a preliminary study on a small dataset.

In our study, we measure the performance with code coverage and fault detection rates achieved by the test inputs generated within the given testing time limit. For code coverage, we use line coverage, which is widely used to measure the effectiveness of symbolic execution [33, 15, 87, 86]. For fault detection rates, since real faults are usually small in number in practice and mutation faults have been widely recognized as suitable for simulating real faults in software testing experimentation [5, 51, 26, 25], following prior work [86, 57], we use mutation testing to simulate real faults and check the mutation scores, i.e., the proportion of killed mutants in all generated mutants. When collecting code coverage and mutation scores, we use widely-used and mature tools gcov⁷ and mutGen [5]. When calculating mutation scores, following prior work [86, 57], we regard the console outputs of the original program as test oracles. If there is any difference between the console outputs of the original program and the console outputs of a mutant for the same test inputs, we regard a mutant as killed.

⁶ <http://www.cs.waikato.ac.nz/ml/weka/>.

⁷ <http://ltp.sourceforge.net/coverage/gcov.php>

4.2 Subjects

Following previous work on symbolic execution [15, 33, 87, 57, 86, 62], we also use GNU Coreutils C programs as subjects, which implement different tools for Unix-like operating systems [15, 57]. In particular, we use 76 GNU Coreutils 6.11 programs⁸, whose total lines of source code (SLOC)⁹ are 39,752 linked with an internal library size of 49,710 SLOC and an external library size of 223,147 SLOC.

Furthermore, when measuring mutation scores, we use 40 programs in GNU Coreutils because the rest of programs cannot produce outputs under the study environment or their outputs are related to environment/context information (e.g., system time). Following prior work [86], for each program, we randomly select 100 mutants. If the total number of generated mutants is less than 100, we use all generated mutants instead.

4.3 Experimental Setup

We consider the following independent variables:

Compared Approaches. LEO is the first automated approach to tuning compiler optimizations for accelerating symbolic execution. Therefore, we compare LEO with only the default optimization configurations of KLEE, i.e., all compiler optimizations off (abbreviated as NO) and all compiler optimizations on (abbreviated as ALL). Here, NO is regarded as the baseline, representing the original symbolic execution without any code transformations for programs under test whereas ALL is regarded as a compared approach applying all available compiler optimizations to accelerate symbolic execution.

Time limits. As the GNU Coreutils programs are normally large and complicated, all paths of a program cannot be fully explored by symbolic execution during the acceptable period of time. Therefore, similar as prior work [15, 33], in the experiment we also limit the maximum execution time of KLEE and halt its execution when reaching the time limit. In particular, for the testing time limit, we set it to 10, 15, 20, 25, and 30 minutes, to investigate whether LEO always performs well regardless of testing time limits. We set the default training time limit to be 10 minutes in LEO. Moreover, we also study the impact of different training time limits on LEO. Due to the high cost of training, we first set the training time limit to be 10 minutes to 30 minutes with the step of 10 minutes. Then, we also add a 5-minute training time limit to better understand the trend of the impact of training time limit. That said, we set the training time limit to be 5, 10, 20, and 30 minutes.

Variants of LEO. To explore whether each component of LEO (i.e., function optimization and library optimization) contributes to LEO on accelerating symbolic execution, we adapt LEO by removing each component and compare the performance of the adapted LEO and the original LEO. In particular, LEO has four variants through such adaption, which are (1) LEO with all compiler optimizations for libraries turned on (denoted as LEO-Lall), (2) LEO with all compiler optimizations for libraries turned off (LEO-Lno), (3) LEO with all compiler optimizations for functions turned on (LEO-Fall), and (4) LEO with all compiler optimizations for functions turned off (LEO-Fno). That is, LEO-Lall and LEO-Lno are variants of LEO without library optimization prediction, LEO-Fall and LEO-Fno are variants of LEO without function optimization prediction.

⁸ We remove some programs from GNU Coreutils mainly because they can destroy our experimental data by generating dangerous test inputs.

⁹ Following prior work [87, 61, 54, 60], the SLOC in this paper are measured by cloc, which is accessible at <https://github.com/AlDanial/cloc>.

Machine learning algorithms. To investigate the impact of machine learning algorithms on LEO, we consider other five typical machine learning algorithms besides SMO – Alternating Decision Tree (abbreviated as ADT) [36], Bayesian Logistic Regression (BLR) [40], Multinomial Logistic Regression (MLR) [56], LogitBoost (LB) [37], and Random Forests (RF) [12]. In particular, we also use their implementations provided by Weka.

Following prior work on machine learning [20, 6], LEO is evaluated through leave-one-out cross-validation. That is, for each subject, we use the instances collected from the *remaining* 75 programs as the training data to build predictive models for compiler optimizations respectively, and use these predictive models to learn the settings of compiler optimizations for the specific subject. The training is conducted offline, and not included as overhead. Note that we use 68 of 76 programs as the testing programs in turn because the other 8 programs incur KLEE errors when using some predicted compiler optimization settings.

The dependent variables considered are line coverage and mutation scores, which have been widely used in prior studies on symbolic execution [57, 86, 33, 15, 87].

4.4 Verifiability

The experimental study is conducted on a workstation with eight-core Intel Xeon E5620 CPU (2.4GHz) with 24G memory, and Ubuntu 15.04 operating system. For ease of experiment replication, we release the tools and implementation used in our experiment as well as all the experimental data at the homepage of LEO¹⁰. The detailed results in the homepage allow for verification without running the experiment again. The open-source tools, the implementation of our experiment (including the source code and readme files), and the subjects and mutants are available, so that one can easily reproduce our experiment.

4.5 Threats to Validity

The threats to internal validity mainly lie in the tool supports and our own implementations. To reduce the threat from tool supports, we use the widely-used KLEE symbolic execution engine [15] and the LLVM compiler infrastructure [55]. Since LEO predicts compiler optimizations for each source-code function, it may discount the effect of inter-procedural compiler optimizations. In the future, we plan to utilize LEO to predict optimizations for *a subset of* functions rather than *a single* function to reduce this threat. It can also bring an additional benefit, i.e., reducing the cost of LEO for optimization prediction. Also, we use the mature tools, i.e., gcov and mutGen [5], to collect line coverage and generate mutants, respectively. To avoid implementation errors, the first two authors review the source code and experimental scripts, and we adopt the mature implementations of those machine learning algorithms used in our study, which are provided by Weka.

The threat to external validity mainly lies in the studied subjects. Although we use the widely-used GNU Coreutils programs [15, 33, 87, 57], they may not be representative of other programs. To reduce this threat, we will use more and larger subjects in the future. Note that our current subjects do not suffer from overfitting. The reason is that GNU coreutils was created by merging a lot of earlier GNU packages; even within the same package, programs differ in their implementation, creation time, and functionalities. Moreover, LEO optimizes at the function rather than program level. Regarding to the libraries, LEO predicts

¹⁰<https://github.com/JunjieChen/leo>.

optimizations of a library for a target program based on the actual library portions invoked and does not necessarily produce the same prediction results on the same libraries of different target programs, which is confirmed by our experimental data. Therefore, the library code does not have overfitting concerns as well.

The threats to construct validity lie in the measurement, the time limits, and the compared approaches. In this study, we measure the performance of LEO through only its acceleration effectiveness rather than its cost because LEO has little overhead¹¹. In particular, we choose the mostly used line coverage and mutation scores. The second threat comes from the time limit, including the training and testing time limits. To reduce these threats, we will repeat the experiment by using other time limits. The third threat lies in the compared approaches. As the first work on optimization prediction for symbolic execution, we use the state-of-the-art symbolic execution work KLEE and its compiler optimization support as the compared approaches (i.e., ALL and NO). There are also some other approaches that may be compared in the study, such as statically applying a subset of compiler optimizations for all the programs [84, 33]. However, according to the existing work [33], there is no unified compiler-optimization configuration guaranteeing the efficiency of symbolic execution for all programs. In particular, the experimental results in the existing work [33] have shown that the four different subsets of KLEE compiler optimizations that are designed based on their knowledge for symbolic execution perform almost the same as turning on all the optimizations (i.e., ALL). Therefore, statically applying a subset of compiler optimizations may not outperform LEO.

4.6 Results and Analysis

4.6.1 RQ1: Performance Comparison

Performance on line coverage. Table 1 lists the line coverage achieved by LEO and ALL/NO under the 10-minute testing time limit¹², where (✓), (○), (✗) represent that the approach achieves the highest, medium, lowest line coverage on the corresponding subject among LEO, ALL and NO, respectively. In particular, the last two rows of this table present the number of subjects where each approach achieves the best, medium, and worst results. From this table, the number of subjects where LEO achieves the best performance (i.e., 50) is much larger than that of ALL (i.e., 26) and NO (i.e., 13), and the number of subjects where LEO achieves the worst performance (i.e., 5) is much smaller than that of ALL (i.e., 13) and NO (i.e., 44). Based on these results, LEO is more effective than the baseline NO, demonstrating code transformation is indeed a promising direction to accelerate symbolic execution. Also, LEO is more effective than ALL, indicating that effectively tuning compiler optimizations is a good exploration in this direction and our machine-learning based approach indeed predicts better compiler-optimization settings specific to symbolic execution.

Figure 3 further shows the comparison results between LEO and ALL/NO under the 10-minute testing time limit, where we calculate the difference between the line coverage achieved by LEO and that achieved by ALL/NO, as the coverage improvement using LEO. In this figure, the y axis represents the coverage improvement using LEO and the x axis sorts the

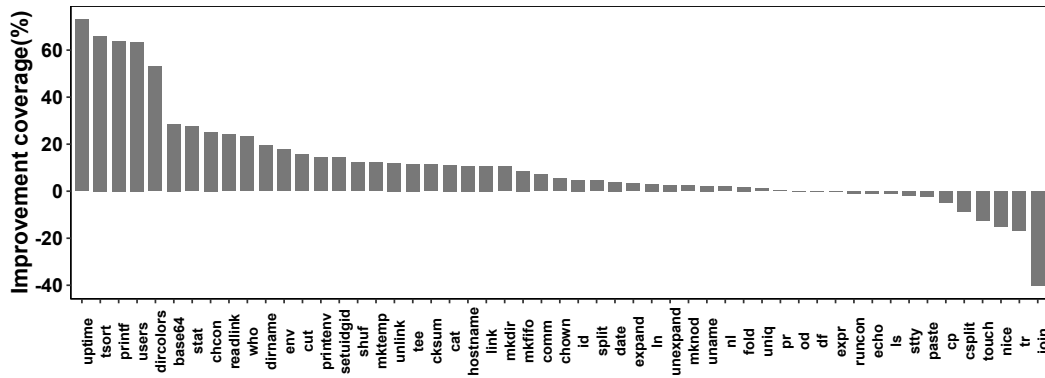
¹¹ Predictive models are built offline and they predict each optimization for each function or libraries very quickly (in seconds).

¹² If no otherwise specified, all training symbolic execution runs of LEO use the default 10-minute training time limit in the remainder of this paper.

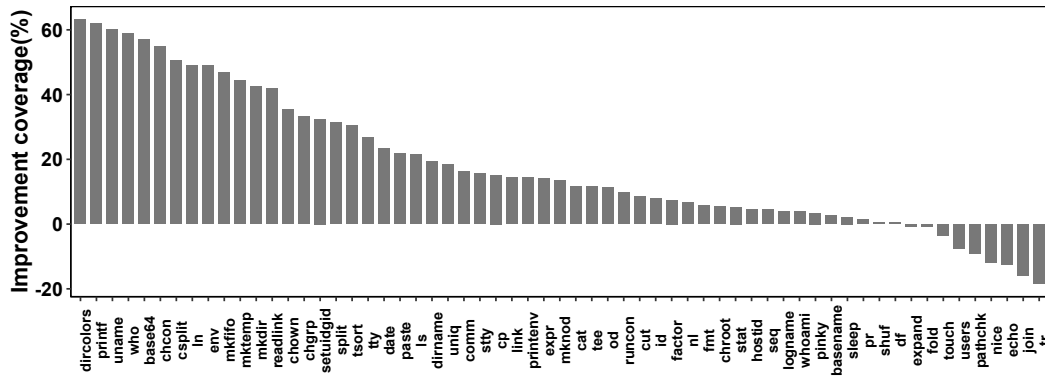
■ **Table 1** Line coverage achieved by LEO/ALL/NO within 10-minute testing time limit

Sub	LEO	ALL	NO	Sub	LEO	ALL	NO
base64	71.43(✓)	42.86(○)	14.29(✗)	basename	79.49(✓)	79.49(✓)	76.92(✗)
cat	66.38(✓)	55.17(○)	54.74(✗)	chcon	70.77(✓)	45.64(○)	15.90(✗)
chgrp	67.78(✓)	67.78(✓)	34.44(✗)	chown	65.59(✓)	60.22(○)	30.11(✗)
chroot	62.16(✓)	62.16(✓)	56.76(✗)	cksum	91.94(✓)	80.65(✗)	91.94(✓)
comm	78.57(✓)	71.43(○)	62.24(✗)	cp	41.46(○)	46.34(✓)	26.29(✗)
csplit	53.76(○)	62.57(✓)	3.30(✗)	cut	64.53(✓)	48.99(✗)	56.08(○)
date	48.05(✓)	44.16(○)	24.68(✗)	df	64.15(○)	64.42(✓)	63.61(✗)
dircolors	73.16(✓)	20.00(○)	10.00(✗)	dirname	93.55(✓)	74.19(○)	74.19(○)
echo	27.18(✗)	28.16(○)	39.81(✓)	env	100.00(✓)	82.22(○)	51.11(✗)
expand	42.38(○)	39.07(✗)	43.05(✓)	expr	48.22(○)	48.52(✓)	34.02(✗)
factor	71.64(✓)	71.64(✓)	64.18(✗)	fmt	65.83(✓)	65.83(✓)	60.19(✗)
fold	43.36(○)	41.59(✗)	44.25(✓)	hostid	63.64(✓)	63.64(✓)	59.09(✗)
hostname	67.86(✓)	57.14(✗)	67.86(✓)	id	32.03(✓)	27.34(○)	24.22(✗)
join	12.93(✗)	53.06(✓)	28.80(○)	link	75.00(✓)	64.29(○)	60.71(✗)
ln	78.35(✓)	75.26(○)	29.38(✗)	logname	56.00(✓)	56.00(✓)	52.00(✗)
ls	44.24(○)	45.33(✓)	22.70(✗)	mkdir	77.27(✓)	66.67(○)	34.85(✗)
mkfifo	82.98(✓)	74.47(○)	36.17(✗)	mknod	56.10(✓)	53.66(○)	42.68(✗)
mktemp	88.89(✓)	76.77(○)	44.44(✗)	nice	61.02(✗)	76.27(✓)	72.88(○)
nl	48.82(✓)	46.92(○)	42.18(✗)	nohup	77.63(✓)	77.63(✓)	77.63(✓)
od	40.65(○)	40.79(✓)	29.25(✗)	paste	66.84(○)	68.98(✓)	44.92(✗)
pathchk	46.97(○)	46.97(○)	56.06(✓)	pinky	83.33(✓)	83.33(✓)	79.91(✗)
pr	38.08(✓)	37.86(○)	36.64(✗)	printenv	77.14(✓)	62.86(○)	62.86(○)
printf	74.32(✓)	10.51(✗)	12.45(○)	pwd	20.34(✓)	20.34(✓)	20.34(✓)
readlink	96.00(✓)	72.00(○)	54.00(✗)	runcon	54.37(○)	55.34(✓)	44.66(✗)
seq	53.04(✓)	53.04(✓)	48.62(✗)	setuidgid	55.84(✓)	41.56(○)	23.38(✗)
shuf	59.88(✓)	47.67(✗)	59.30(○)	sleep	45.65(✓)	45.65(✓)	43.48(✗)
split	45.62(✓)	41.01(○)	14.29(✗)	stat	37.05(✓)	9.47(✗)	31.75(○)
stty	29.43(○)	31.32(✓)	13.77(✗)	tee	86.96(✓)	75.36(○)	75.36(○)
touch	56.25(✗)	68.75(✓)	59.72(○)	tr	22.15(✗)	39.15(○)	40.52(✓)
tsort	72.91(✓)	6.90(✗)	42.36(○)	tty	76.67(✓)	76.67(✓)	50.00(✗)
uname	79.55(✓)	77.27(○)	19.32(✗)	unexpand	47.42(✓)	44.85(✗)	47.42(✓)
uniq	64.32(✓)	63.24(○)	45.95(✗)	unlink	72.00(✓)	60.00(✗)	72.00(✓)
uptime	91.03(✓)	17.95(✗)	91.03(✓)	users	90.38(○)	26.92(✗)	98.08(✓)
who	83.09(✓)	59.71(○)	24.10(✗)	whoami	53.85(✓)	53.85(✓)	50.00(✗)
Best(✓)	50	26	13	Med(○)	13	29	11
Worst(✗)	5	13	44	–	–	–	–

subjects in their coverage improvement by removing those with zero coverage improvement. That is, any bar above 0 represents a subject whose LEO result is better than ALL or NO, whereas any bar below 0 represents a subject whose LEO result is worse. Besides, LEO achieves the same line coverage as ALL in 14 subjects, and as NO in 6 subjects. From this figure, the vast majority of bars are above 0. That is, LEO makes symbolic execution more efficient than both ALL and NO in most cases. Moreover, the improvement of LEO is usually larger than its decrement. In particular, the increased coverage for *uptime*, *tsort*, *printf* and *users* on ALL, as well as *dircolors*, *printf* and *uname* on NO are even more than 60%.



(a) LEO-ALL



(b) LEO-NO

Figure 3 Coverage improvement within 10-minute testing time limit

This is another empirical evidence that LEO does effectively accelerate symbolic execution on most subjects.

To further quantitatively measure the performance of LEO on accelerating symbolic execution, similar to previous work [24], we calculate the improvement rate of LEO in terms of line coverage for each program. It is calculated via Formula 2, where $Cov(LEO)$ represents the line coverage achieved by LEO and $Cov(ALL(orNO))$ represents the line coverage achieved by ALL or NO. The average line-coverage improvement rate of LEO compared with ALL on all subjects is 46.48% and that compared with NO is 88.92%, demonstrating the significant acceleration performance of LEO in terms of line coverage.

$$Rate_{Cov} = \frac{Cov(LEO) - Cov(ALL(orNO))}{Cov(ALL(orNO))} * 100\% \tag{2}$$

Note that in some cases LEO decelerates symbolic execution, e.g., *nice* and *tr*. We try to analyze the possible reasons and find that some compiler optimizations have coupling effect in fact. For instance, based on the comments of LLVM, the optimization “IndvarSimplify”¹³ should be performed after all the desired loop optimizations (e.g., the optimization “LoopRotation”). Currently, LEO learns each predictive model for each compiler optimization

¹³This optimization analyzes and transforms the induction variables into simpler forms suitable for subsequent analysis and transformation.

■ **Table 2** Mutation scores achieved by LEO/ALL/NO within 10-minute testing time limit.

Sub	LEO	ALL	NO	Sub	LEO	ALL	NO
base64	17.00 (✓)	11.00 (○)	4.00(✗)	basename	44.00 (✓)	44.00 (✓)	44.00(✓)
chcon	39.00 (✓)	18.00 (○)	7.00(✗)	cksum	10.00 (○)	9.00 (✗)	14.00(✓)
comm	20.00 (○)	14.00 (✗)	21.00(✓)	cut	19.00 (○)	18.00 (✗)	25.00(✓)
dircolors	46.00 (✓)	12.00 (✗)	45.00(○)	dirname	74.12 (○)	98.82 (✓)	74.12(○)
env	100.00 (✓)	62.20 (○)	48.78(✗)	expand	3.00 (○)	3.00 (○)	13.00(✓)
expr	100.00 (✓)	99.00 (○)	8.00(✗)	fold	3.00 (○)	3.00 (○)	9.00(✓)
hostid	60.87 (✓)	60.87 (✓)	34.78(✗)	link	37.00 (✗)	46.00 (✓)	39.00(○)
ln	99.00 (✓)	42.00 (○)	11.00(✗)	logname	62.50 (✓)	62.50 (✓)	32.50(✗)
mkfifo	51.28 (○)	50.00 (✗)	100.00(✓)	mknod	50.00 (✓)	46.00 (○)	26.00(✗)
nice	29.00 (✗)	48.00 (✓)	40.00(○)	nl	0.00 (○)	0.00 (○)	7.00(✓)
nohup	39.00 (✓)	39.00 (✓)	39.00(✓)	od	25.00 (○)	28.00 (✓)	3.00(✗)
paste	11.00 (○)	10.00 (✗)	23.00(✓)	pathchk	20.00 (○)	18.00 (✗)	24.00(✓)
printf	25.00 (✓)	4.00 (○)	1.00(✗)	pwd	7.00 (✓)	7.00 (✓)	7.00(✓)
readlink	66.67 (✓)	38.10 (✗)	47.62(○)	runcon	32.00 (✓)	27.00 (○)	25.00(✗)
setuidgid	27.00 (✓)	20.00 (○)	13.00(✗)	sleep	32.00 (✓)	21.00 (✗)	30.00(○)
split	12.00 (○)	16.00 (✓)	11.00(✗)	tee	31.00 (✓)	20.00 (✗)	29.00(○)
touch	23.00 (✗)	27.00 (✓)	25.00(○)	tr	4.00 (✗)	12.00 (○)	15.00(✓)
tsort	4.00 (✗)	9.00 (✓)	7.00(○)	tty	46.30 (✓)	44.44 (○)	24.07(✗)
unexpand	3.00 (○)	3.00 (○)	12.00(✓)	unlink	98.61 (✓)	58.33 (✗)	98.61(✓)
users	100.00 (✓)	100.00 (✓)	100.00(✓)	whoami	69.70 (✓)	69.70 (✓)	36.36(✗)
Best(✓)	22	14	16	Med(○)	13	15	9
Worst(✗)	5	11	15	–	–	–	–

individually. Neglect of such couple effects may impact the performance of LEO. Therefore, in the future we plan to improve LEO by learning predictive models considering the coupling effect of compiler optimizations, which can be learned/inferred through source code and documentation of these optimizations.

Performance on mutation score. Besides line coverage, Table 2 further shows the comparison of mutation scores. Similar with Table 1, in this table, (✓), (○), (✗) represent the approach achieves the highest, medium, lowest mutation scores. From this table, similarly, the number of subjects where LEO achieves the best mutation scores (i.e., 22) is larger than that of ALL (i.e., 14) and NO (i.e., 16), and the number of subjects where LEO achieves the worst mutation scores (i.e., 5) is smaller than that of ALL (i.e., 11) and NO (i.e., 15). This finding further confirms the performance of LEO in enhancing symbolic execution.

Similarly, to further quantitatively measure its performance, we also calculate the improvement rate of LEO in terms of mutation score for each program. It is calculated via the similar formula – Formula 3, where $Mut(LEO)$ represents the mutation score achieved by LEO and $Mut(ALL(orNO))$ represents the mutation score achieved by ALL or NO. The average mutation-score improvement rate of LEO compared with ALL on all subjects is 33.88% and that compared with NO is 149.11%, further demonstrating the significant acceleration performance of LEO in terms of mutation score.

$$Rate_{Mut} = \frac{Mut(LEO) - Mut(ALL(orNO))}{Mut(ALL(orNO))} * 100\% \quad (3)$$

■ **Table 3** Comparison within various testing time limits under the default training time limit.

Time (minutes)	#Best			#Worst		
	LEO	ALL	NO	LEO	ALL	NO
10	50	26	11	5	13	44
15	51	29	10	5	12	46
20	46	33	8	5	10	47
25	44	38	8	5	9	45
30	50	34	7	6	9	45

■ **Table 4** Statistics analysis on LEO and ALL/NO within various testing time limits under the default training time limit ($\alpha=0.05$).

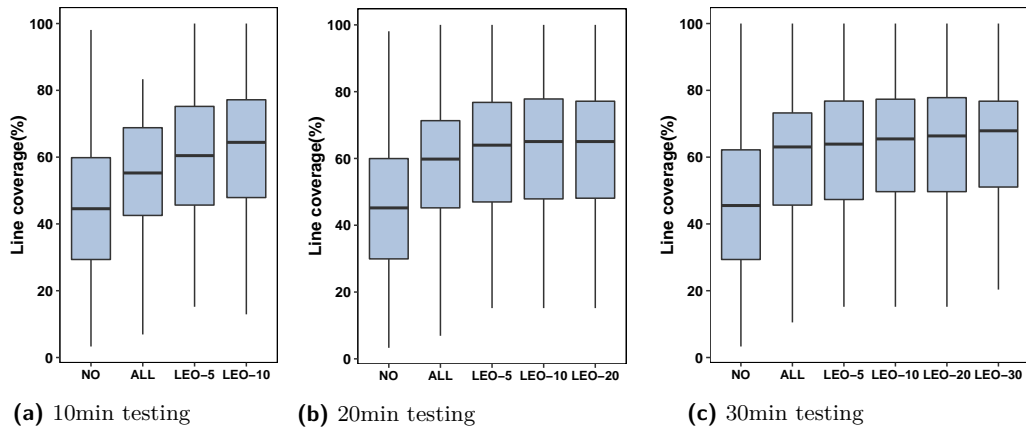
Time (minutes)		10	15	20	25	30
LEO v.s. ALL	Imp. rate(%)	46.48	39.01	37.80	23.85	18.39
	p-value	0.000(*)	0.001(*)	0.008(*)	0.065	0.029(*)
LEO v.s. NO	Imp. rate(%)	88.92	86.70	87.26	79.47	89.60
	p-value	0.000(*)	0.000(*)	0.000(*)	0.000(*)	0.000(*)

4.6.2 RQ2: Impact of Training and Testing Time Limits

Although label collection in LEO needs to fix time limit, in practical usage symbolic execution may set various time limits (i.e., testing time limit) based on different requirements. It is quite necessary to investigate whether LEO always works no matter which testing time limit is set. Therefore, we first explore the performance of LEO through symbolic execution in different testing time limits by using the predictive models learnt in default 10-minute training time limit, whose results are shown in Table 3. In this table, the first column lists various testing time limits and Columns 2-4 and 5-7 represent the number of subjects where the corresponding approach achieves the best and worst performance, respectively. From Table 3, the number of subjects where LEO achieves best performance is always much larger than that of ALL and NO, and the number of subjects where LEO achieves worst performance is always much smaller than that of ALL and NO. That is, LEO accelerates symbolic execution regardless of testing time limits.

To learn whether LEO outperforms ALL and NO significantly at various testing time limits, we further perform statistical analysis on their results. First, we analyze the population on line coverage achieved by each approach and find that the population of each approach follows the normal distribution by Kolmogorov-Smirnov test [63], which is the precondition of the paired sample T test. Then, we perform a paired sample T test (whose significant level α is 0.05), and the results are shown in Rows 3 and 5 of Table 4, where “*” demonstrates significant difference between the compared approaches. Moreover, in Table 4, Rows 2 and 4 refer to the average line-average improvement rates on all subjects. From this table, LEO significantly outperforms ALL in 4/5 testing time limit comparisons with line-coverage improvement rates ranging from 18.39% to 46.48%, and always significantly outperforms NO with line-coverage improvement rates ranging from 79.47% to 89.60%.

Besides, from Tables 3 and 4, the smaller the gap between the default training time limit and the used testing time limit, the better LEO tends to perform compared with ALL and NO (especially ALL). This observation is as expected due to two possible reasons. First, given sufficient time, symbolic execution can always achieve high line coverage for a subject.



■ **Figure 4** Trend of performance of LEO with different training time limits.

30 minutes may be already long enough for symbolic execution of individual GNU Coreutils programs, especially the transformed programs using compiler optimizations. Therefore, symbolic execution will achieve similar (high) line coverage and become saturate eventually, no matter what the settings of compiler optimization are (especially for LEO and ALL). Second, based on the theory of machine learning [82], the time limit in training and testing staying consistent tends to achieve the best effectiveness. In our study, even though the training time limit of LEO is inconsistent with the testing time limit, LEO still improves the efficiency of symbolic execution. That said, if LEO sets training time limit longer than 10 minutes, its accelerating effectiveness may be more obvious when being applied to symbolic execution with these longer testing time limits.

Therefore, we further explore the impact of different training time limits on LEO. More specifically, we study whether the performance of LEO within these longer testing time limits becomes better when using longer training time limits. Figure 4 shows the performance trends of LEO whose training time limit is gradually close to the testing time limit, where the line in each box represents the median line coverage and LEO- X (i.e., $X = 5, 10, 20$, and 30) refers to LEO with X -minute *training* time limit. Figures 4a, 4b, and 4c present the trends in 10-minute, 20-minute, and 30-minute *testing* time limit, respectively. From each subfigure in Figure 4, with the training time limit being closing to the specific testing time limit, LEO indeed achieves better performance, confirming our hypothesis. Also, we find that LEO always performs better than ALL and NO no matter which training time limit is used.

Furthermore, Table 5 further shows more details about the impacts of training time limits. The comparison results include the number of subjects where LEO with various training time limits achieve best and worst performance, and the average line-coverage improvement rates compared with ALL and NO. For each row in this table, reading values from left to right, we find that with the training time limit being close to the specific testing time limit, the number of subjects where LEO achieve best performance becomes larger, the number of subjects where LEO achieves worst performance becomes smaller, and the average line-coverage improvement rates of LEO mostly becomes better. That is, when the training time limit in LEO is close to the testing time limit, LEO achieves better performance on accelerating symbolic execution.

Overall, LEO mostly significantly accelerates symbolic execution in various testing time limits with our default training time limit. Furthermore, when having more sufficient training time, LEO tends to perform better for longer testing time limits.

■ **Table 5** Comparison within various training time limits.

Training time (minutes)		5	10	20	30
Testing-10min	#Best	46	50	—	—
	#Worst	6	5	—	—
	Imp. rate(%) (v.s. ALL)	39.31	46.48	—	—
	Imp. rate(%) (v.s. NO)	79.18	88.92	—	—
Testing-20min	#Best	45	46	47	—
	#Worst	6	5	5	—
	Imp. rate(%) (v.s. ALL)	32.04	37.80	38.23	—
	Imp. rate(%) (v.s. NO)	81.66	87.26	87.92	—
Testing-30min	#Best	47	50	51	51
	#Worst	8	6	6	4
	Imp. rate(%) (v.s. ALL)	15.98	18.39	18.51	18.51
	Imp. rate(%) (v.s. NO)	83.51	89.60	89.81	89.69

■ **Table 6** Comparison between LEO and its variants.

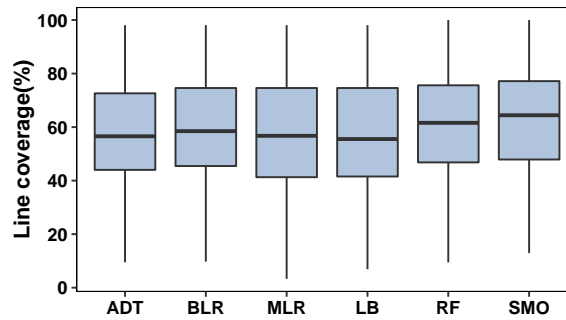
Approach	LEO v.s.			
	LEO-Lall	LEO-Lno	LEO-Fall	LEO-Fno
#Win	36	50	7	7
#Lose	17	11	0	2
Avg. Improvement (%)	9.89	16.51	3.89	1.28

4.6.3 RQ3: Contribution of Function/Library Optimization

Table 6 shows the comparison between LEO and its four variants. In this table, the second/third row presents the number of subjects where LEO achieves higher/lower line coverage than its variants within 10-minute testing time limit. The last row presents the average coverage improvement through LEO compared with its variants. Note that we do not list the number of subjects that the two compared approaches perform equally, i.e., achieve the same line coverage. From this table, LEO performs much better than its four variants since the number of subjects LEO performing better is always larger than the number of subjects it performing worse. Therefore, both function optimization and library optimization are indispensable.

Furthermore, the difference between “#Win” and “#Lose” in the second and third columns (i.e., results of LEO without library optimization prediction) is much larger than the following two columns (i.e., results of LEO without function optimization prediction). Moreover, the average coverage improvement of LEO compared with LEO without library optimization prediction (i.e., LEO-Lall and LEO-Lno) is also larger than that of LEO compared with LEO without function optimization prediction (i.e., LEO-Fall and LEO-Fno). That is, library optimization is more important than function optimization for LEO in accelerating symbolic execution. The main reason is that the studied GNU Coreutils programs usually use a large portion of library code (Section 4.2).

These results tell us another promising direction for further improving LEO. We have known that library optimization is more important for LEO. Currently, LEO achieves such great performance through simply taking all the libraries used in the program as a whole by predicting unified settings on compiler optimizations for them. If library optimization can be



■ **Figure 5** Comparison on machine learning algorithms.

better dealt with, LEO are quite likely to be improved on accelerating symbolic execution. In the future, we plan to learn how to further fine-tune compiler optimizations for libraries, e.g., splitting libraries into several sets of functions (which is more coarse granularity than each single function but makes it more efficient), or analyzing open-source repositories such as Github. Since libraries are widely used in software development, we believe that, like existing library researches (e.g., building a summary for library code to accelerating the analysis of client code [80]), researches on better transforming libraries for accelerating symbolic execution are also worthy and should attract more attentions.

4.6.4 RQ4: Impact of Machine Learning Algorithms

Figure 5 shows the results of LEO with various machine learning algorithms, to investigate the impact of machine learning algorithms on LEO. Actually, our approach using any of machine learning algorithms outperforms ALL and NO. From this figure, our approach using SMO performs slightly better than our approach using other machine learning algorithms, e.g., the top, median and bottom of SMO in box-plot are all higher than those of all other algorithms. Therefore, LEO achieves stably good acceleration performance for all studied machine learning algorithms, and SMO is a better choice.

5 Discussion

In this section, we first discuss the new direction that LEO opens for symbolic execution acceleration, and then discuss the impact of search heuristics and caching on LEO.

5.1 Promising Direction

Our work demonstrates the significant performance of code transformation on accelerating symbolic execution, indicating a promising direction for accelerating symbolic execution. Moreover, learning to tune existing compiler optimizations is a good exploration to accelerate symbolic execution in this direction. It can be further studied from the following aspects:

First, it is promising to design new code transformations specific to symbolic execution. As code transformations designed for symbolic execution scarcely exist, and it is very difficult to design such code transformations due to lack of knowledge in this direction, in this paper LEO accelerates symbolic execution by borrowing the knowledge of code transformation for *concrete* execution. Besides, LEO can be used as a light-weight approach to accelerating symbolic execution via code transformation. The results of LEO (e.g., the predictive model)

■ **Table 7** Line coverage achieved by LEO/ALL/NO with random search heuristic and caching within 10-minute testing time limit.

Sub	LEO	ALL	NO	Sub	LEO	ALL	NO
chown	83.87(✓)	82.80(○)	82.80(○)	cp	48.24(✗)	49.05(✓)	48.78(○)
csplit	68.62(○)	64.59(✗)	75.60(✓)	date	85.06(✓)	80.52(✗)	82.47(○)
echo	87.38(✓)	79.61(○)	79.61(○)	fmt	71.16(○)	66.77(✗)	75.86(✓)
id	60.94(✓)	60.16(○)	60.16(○)	ln	82.99(✓)	76.80(✗)	77.84(○)
ls	53.93(✓)	50.34(✗)	53.46(○)	nice	96.61(✓)	94.92(✗)	96.61(✓)
nl	86.26(✓)	83.89(○)	78.20(✗)	od	86.08(✓)	86.08(✓)	84.11(✗)
paste	92.51(✓)	92.51(✓)	92.51(✓)	pr	60.93(○)	60.60(✗)	61.15(✓)
printenv	100.00(✓)	100.00(✓)	100.00(✓)	pwd	20.34(✓)	20.34(✓)	20.34(✓)
runcon	66.99(✓)	66.99(✓)	66.99(✓)	stat	63.23(✓)	57.38(✗)	62.40(○)
touch	76.39(✗)	77.08(✓)	77.08(✓)	tr	55.24(○)	55.69(✓)	54.32(✗)
Best(✓)	14	8	9	Med(○)	4	4	8
Worst(✗)	2	8	3	–	–	–	–

can also provide knowledge to facilitate the design of code transformation specific to symbolic execution. That is, LEO can be regarded as a necessary step to code transformation for symbolic execution. Furthermore, when code transformations for symbolic execution are available, we can also use LEO to tune these transformations to achieve best acceleration performance due to the generality of our machine-learning based approach.

Second, code transformation can be combined with other symbolic execution optimization techniques (e.g., parallel and incremental symbolic execution). Code transformation manipulates the program under test by regarding symbolic execution as a black box, and thus it is orthogonal to other symbolic execution optimization techniques. For instance, through code transformation, a program is transformed into an easy-to-analyze target program, and then various optimized symbolic execution techniques can be applied to this new target program, making the analysis more easier. That is, LEO can further improve all the existing refined symbolic execution techniques because of the orthogonality.

5.2 Impact of Random Search Heuristic and Caching

The default KLEE random search heuristic and caching is disabled in our experimental study because the random search heuristic can bring much randomness and non-determinism and the caching contents can be different for different strategies, making it hard to evaluate the effectiveness of LEO. However, as random search heuristic and caching are widely used for symbolic execution, it is still interesting to know the performance of LEO with random search heuristic and caching on. Therefore, we conduct a preliminary study on 20 randomly-chosen GNU Coreutils programs with the similar setting as Section 4, but using random search heuristic and caching on. Table 7 shows the line coverage achieved by LEO, ALL, and NO with random search heuristic and caching. In particular, we repeat the experiments 5 times to reduce the impact of nondeterminism. From this table, with random search heuristic and caching, the number of subjects where LEO achieves the best performance (i.e., 14) is still larger than that of ALL (i.e., 8) and NO (i.e., 9), and the number of subjects where LEO achieves the worst performance (i.e., 2) is also still smaller than that of ALL (i.e., 8) and NO (i.e., 3). Furthermore, most subjects keep the same rankings with the results in Table 1. For example, LEO always performs the best for *chown* no matter whether using random search heuristic and caching. In particular, the main difference between using and not using random

search heuristic and caching is that when using them, the difference of LEO, ALL, and NO becomes smaller than that when not using them, no matter which technique performs the best. This is because random search heuristic and caching make symbolic execution more efficient and thus LEO, ALL, and NO achieve the similar (high) line coverage under the default 10-minute testing time limit. Therefore, *the setting with random search heuristic and caching has the orthogonal impact on accelerating symbolic execution with LEO*. That is, it further confirms that LEO, which optimizes *the code under test*, accelerates symbolic execution from an orthogonal dimension with techniques optimizing *symbolic execution itself*.

6 Related Work

In this section, we present the related work on both the symbolic execution and code transformation areas.

6.1 Symbolic Execution

Symbolic execution [29, 52, 17], is a systematic technique for generating program test inputs based on exploring all possible program paths, which has been recognized as one of the most costly testing methodologies. To improve the efficiency and effectiveness of symbolic execution, a huge amount of research effort [74, 81, 15, 28, 45, 50, 43, 70, 71, 34, 65, 58, 68, 79, 67, 83, 13, 44, 41, 85, 49, 11, 75, 3, 73, 67, 39, 59, 10, 35, 72] has been dedicated to the area, and more details on symbolic execution can be found in a recent survey [18]. To reduce the cost of symbolic execution, variants of symbolic execution have been proposed, e.g., concolic execution [42, 74] and execution-based testing [15, 16], which combine concrete execution with symbolic execution. Some researchers also proposed various techniques to accelerate the path exploration in symbolic execution. One specific approach is distributed symbolic execution where the path exploration is distributed among different workers [78, 77]. Since real-world programs usually consist of various sub-modules, a number of techniques have also been proposed to use the compositional approach to speed up symbolic execution [13, 44, 43]. Furthermore, Researchers have also proposed techniques to prune the search space of symbolic execution [79, 88].

Different from the above previous work on symbolic execution, our work accelerates symbolic execution via another orthogonal dimension – manipulating the programs under test via code transformation. Here we discuss closely related work applying code transformation to symbolic execution. Anand et al. [4] applied code transformation based on type-dependence analysis to help users identify problematic cases for symbolic execution and then the users can manually solve the problem. Dong et al. [33] showed that compiler optimizations could be harmful for symbolic execution via empirical study. Cadar [14] pointed out the potential direction of transforming program under test for better symbolic execution. Perry et al. [66] proposed code transformation rules specific to array operations to simplify constraints involving arrays for symbolic execution. Wagner et al. [84] and Converse et al. [31] mainly focused on reducing path exploration by simplifying program control-flow, but the generated tests may fail to cover the original program paths. In contrast, our work provides a general and fully automated machine-learning-based solution for accelerating symbolic execution based on all possible transformations.

6.2 Code Transformation

Compiler optimization is a typical and mature approach of code transformation [2, 32]. Besides, testability transformation [46], another code transformation approach, has been proposed to speed up search-based test generation [8, 53, 7], but cannot be directly applied to symbolic execution. Here we mainly review the work on compiler optimization [2, 32, 38, 19, 47, 1, 64, 9] since our work accelerates symbolic execution through this type of code transformation. Traditional work on compiler optimization focused on defining new optimizations and exploring their impacts on program concrete execution [2], whereas recently researchers focused on choosing the most suitable set of optimizations for general or specific target programs on concrete execution. More specifically, iterative compilation [32, 38, 19], a search-based approach that explores the compiler optimization space by iteratively compiling for single optimization objective (e.g., performance or code size) or multiple objectives [47]. Different from previous work searching for optimal compiler optimizations for program *concrete* execution, this paper presents the first work on predicting optimal compiler optimizations for *symbolic* execution based on machine learning.

7 Conclusion

Compiler optimization is a typical code transformation approach, which is firstly proposed to accelerate program concrete execution. In this paper, we present LEO, the first machine-learning-based approach to accelerating symbolic execution through tuning compiler optimizations. More specifically, LEO predicts compiler optimizations for source-code functions and libraries separately and applies the learnt optimization settings by program-splitter and local-optimizer. From our empirical study, compared with the default turning on/off all compiler optimizations in KLEE, LEO achieves the best acceleration performance in 50/68 GNU Coreutils programs and its average improvement rate on all programs is 46.48%/88.92% in terms of line coverage, with the default training/testing time limit. Furthermore, LEO consistently outperforms KLEE default settings with various training/testing time limits, and tends to perform the best when training and testing time limits are close.

References

- 1 F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. P. O'Boyle, J. Thomson, M. Toussaint, and C. K. I. Williams. Using machine learning to focus iterative optimization. In *CGO*, pages 295–305, 2006.
- 2 Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. *Compilers, Principles, Techniques*. Pearson Education, Inc., 1986.
- 3 Aws Albarghouthi, Arie Gurfinkel, Ou Wei, and Marsha Chechik. Abstract analysis of symbolic executions. In *CAV*, pages 495–510, 2010.
- 4 Saswat Anand, Alessandro Orso, and Mary Jean Harrold. Type-dependence analysis and program transformation for symbolic execution. In *TACAS*, pages 117–133, 2007.
- 5 J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *ICSE*, pages 402–411, 2005.
- 6 Tien-Duy B Le, David Lo, Claire Le Goues, and Lars Grunske. A learning-to-rank based fault localization approach using likely invariants. In *ISSTA*, pages 177–188, 2016.
- 7 André Baresel, David Binkley, Mark Harman, and Bogdan Korel. Evolutionary testing in the presence of loop-assigned flags: A testability transformation approach. In *ISSTA*, pages 108–118, 2004.

- 8 David W. Binkley, Mark Harman, and Kiran Lakhotia. Flagremover: A testability transformation for transforming loop-assigned flags. *TOSEM*, 20(3):12:1–12:33, 2011.
- 9 François Bodin, Toru Kisuki, Peter Knijnenburg, Mike O’Boyle, and Erven Rohou. Iterative compilation in a non-linear optimisation space. In *PFDC*, 1998.
- 10 Pietro Braione, Giovanni Denaro, and Mauro Pezzè. Enhancing symbolic execution with built-in term rewriting and constrained lazy initialization. In *FSE*, pages 411–421, 2013.
- 11 Pietro Braione, Giovanni Denaro, and Mauro Pezzè. Symbolic execution of programs with heap inputs. In *FSE*, pages 602–613, 2015.
- 12 Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.
- 13 William R Bush, Jonathan D Pincus, and David J Sielaff. A static analyzer for finding dynamic programming errors. *SPE*, 30(7):775–802, 2000.
- 14 Cristian Cadar. Targeted program transformations for symbolic execution. In *FSE*, pages 906–909, 2015.
- 15 Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, pages 209–224, 2008.
- 16 Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. Exe: Automatically generating inputs of death. *TISSEC*, 12(2):10, 2008.
- 17 Cristian Cadar, Patrice Godefroid, Sarfraz Khurshid, Corina S Păsăreanu, Koushik Sen, Nikolai Tillmann, and Willem Visser. Symbolic execution for software testing in practice: preliminary assessment. In *ICSE*, pages 1066–1071, 2011.
- 18 Cristian Cadar and Koushik Sen. Symbolic execution for software testing: three decades later. *CACM*, 56(2):82–90, 2013.
- 19 John Cavazos, Grigori Fursin, Felix Agakov, Edwin Bonilla, Michael F. P. O’Boyle, and Olivier Temam. Rapidly selecting good compiler optimizations using performance counters. In *CGO*, pages 185–197, 2007.
- 20 Gavin C Cawley and Nicola LC Talbot. Efficient leave-one-out cross-validation of kernel fisher discriminant classifiers. *Pattern Recognition*, 36(11):2585–2592, 2003.
- 21 Nitesh V Chawla. Data mining for imbalanced datasets: An overview. In *Data Min. Knowl. Discov.*, pages 853–867. Springer, 2005.
- 22 Nitesh V. Chawla, Kevin W. Bowyer, Lawrence O. Hall, and W. Philip Kegelmeyer. Smote: synthetic minority over-sampling technique. *JAIR*, pages 321–357, 2002.
- 23 Nitesh V. Chawla, Nathalie Japkowicz, and Aleksander Kotcz. Editorial: Special issue on learning from imbalanced data sets. *SIGKDD Explor*, 6(1):1–6, 2004.
- 24 Junjie Chen, Yanwei Bai, Dan Hao, Yingfei Xiong, Hongyu Zhang, and Bing Xie. Learning to prioritize test programs for compiler testing. In *ICSE*, pages 700–711, 2017.
- 25 Junjie Chen, Yanwei Bai, Dan Hao, Lingming Zhang, Lu Zhang, and Bing Xie. How do assertions impact coverage-based test-suite reduction? In *ICST*, pages 418–423, 2017.
- 26 Junjie Chen, Yanwei Bai, Dan Hao, Lingming Zhang, Lu Zhang, Bing Xie, and Hong Mei. Supporting oracle construction via static analysis. In *ASE*, pages 178–189, 2016.
- 27 Junjie Chen, Wenxiang Hu, Dan Hao, Yingfei Xiong, Hongyu Zhang, Lu Zhang, and Bing Xie. An empirical comparison of compiler testing techniques. In *ICSE*, pages 180–190, 2016.
- 28 Maria Christakis, Peter Müller, and Valentin Wüstholz. Guiding dynamic symbolic execution toward unverified program executions. In *ICSE*, pages 144–155, 2016.
- 29 Lori A. Clarke. A system to generate test data and symbolically execute programs. *TSE*, SE-2(3):215–222, 1976. doi:10.1109/TSE.1976.233817.
- 30 Hayes Converse, Oswaldo Olivo, and Sarfraz Khurshid. Non-semantics-preserving transformations for high-coverage test generation using symbolic execution. In *ICST*, pages 241–252, 2017.

- 31 Hayes Elliott Converse, Oswaldo Olivo, and Sarfraz Khurshid. *Non-semantics-preserving transformations for higher-coverage test generation using symbolic execution*. PhD thesis, The University of Texas at Austin, 2017.
- 32 Keith D Cooper, Alexander Grosul, Timothy J Harvey, Steve Reeves, Devika Subramanian, Linda Torczon, and Todd Waterman. Exploring the structure of the space of compilation sequences using randomized search algorithms. *J Supercomput*, 36(2):135–151, 2006.
- 33 Shiyu Dong, Oswaldo Olivo, Lingming Zhang, and Sarfraz Khurshid. Studying the influence of standard compiler optimizations on symbolic execution. In *ISSRE*, pages 205–215, 2015.
- 34 Ikpeme Erete and Alessandro Orso. Optimizing constraint solving to better support symbolic execution. In *ICSTW*, pages 310–315, 2011.
- 35 Antonio Filieri, Corina S. Păsăreanu, Willem Visser, and Jaco Geldenhuys. Statistical symbolic execution with informed sampling. In *FSE*, pages 437–448, 2014.
- 36 Yoav Freund and Llew Mason. The alternating decision tree learning algorithm. In *ICML*, pages 124–133, 1999.
- 37 Jerome Friedman, Trevor Hastie, Robert Tibshirani, et al. Additive logistic regression: a statistical view of boosting (with discussion and a rejoinder by the authors). *Ann. Stat.*, 28(2):337–407, 2000.
- 38 Grigori Fursin, Yuriy Kashnikov, Abdul Wahid Memon, Zbigniew Chamski, Olivier Temam, Mircea Namolaru, Elad Yom-Tov, Bilha Mendelson, Ayal Zaks, Eric Courtois, Francois Bodin, Phil Barnard, Elton Ashton, Edwin Bonilla, John Thomson, Christopher K. I. Williams, and Michael O’Boyle. Milepost gcc: Machine learning enabled self-tuning compiler. *IJPP*, 39(3):296–327, 2011.
- 39 Jaco Geldenhuys, Matthew B. Dwyer, and Willem Visser. Probabilistic symbolic execution. In *ISSTA*, pages 166–176, 2012.
- 40 Alexander Genkin, David D Lewis, and David Madigan. Large-scale bayesian logistic regression for text categorization. *Technometrics*, 49(3):291–304, 2007.
- 41 Patrice Godefroid. Compositional dynamic test generation. In *POPL*, pages 47–54, 2007.
- 42 Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed automated random testing. In *PLDI*, pages 213–223, 2005.
- 43 Patrice Godefroid, Shuvendu K. Lahiri, and Cindy Rubio-González. Statically validating must summaries for incremental compositional dynamic test generation. In *Proceedings of the 18th International Conference on Static Analysis, SAS’11*, pages 112–128, Berlin, Heidelberg, 2011. Springer-Verlag. URL: <http://dl.acm.org/citation.cfm?id=2041552.2041564>.
- 44 Patrice Godefroid, Aditya V. Nori, Sriram K. Rajamani, and Sai Deep Tetali. Compositional may-must program analysis: Unleashing the power of alternation. In *POPL*, pages 43–56, 2010.
- 45 Shengjian Guo, Markus Kusano, and Chao Wang. Conc-ise: Incremental symbolic execution of concurrent software. In *ASE*, pages 531–542, 2016.
- 46 Mark Harman, André Baresel, David Binkley, Robert Hierons, Lin Hu, Bogdan Korel, Phil McMinn, and Marc Roper. Testability transformation—program transformation to improve testability. In *Formal methods and testing*, pages 320–344. Springer, 2008.
- 47 Kenneth Hoste and Lieven Eeckhout. Cole: Compiler optimization level exploration. In *CGO*, pages 165–174, 2008.
- 48 Y Kumar Jain and Santosh Kumar Bhandare. Min max normalization based data perturbation method for privacy protection. *IJRCCT*, 2(8):45–50, 2011.
- 49 Konrad Jamrozik, Gordon Fraser, Nikolai Tillmann, and Jonathan De Halleux. Augmented dynamic symbolic execution. In *ASE*, pages 254–257, 2012.

- 50 Jinseong Jeon, Xiaokang Qiu, Jonathan Fetter-Degges, Jeffrey S. Foster, and Armando Solar-Lezama. Synthesizing framework models for symbolic execution. In *ICSE*, pages 156–167, 2016.
- 51 René Just, Darioush Jalali, Laura Inozemtseva, Michael D Ernst, Reid Holmes, and Gordon Fraser. Are mutants a valid substitute for real faults in software testing? In *FSE*, pages 654–665, 2014.
- 52 James C King. Symbolic execution and program testing. *CACM*, 19(7):385–394, 1976.
- 53 Bogdan Korel, Mark Harman, S. Chung, P. Apirukvorapinit, R. Gupta, and Q. Zhang. Data dependence based testability transformation in automated test generation. In *ISSRE*, pages 245–254, 2005.
- 54 Tomasz Kuchta, Cristian Cadar, Miguel Castro, and Manuel Costa. Doccovery: Toward generic automatic document recovery. In *ASE*, pages 563–574, 2014.
- 55 Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–86, 2004.
- 56 Saskia Le Cessie and Johannes C Van Houwelingen. Ridge estimators in logistic regression. *Applied statistics*, pages 191–201, 1992.
- 57 You Li, Zhendong Su, Linzhang Wang, and Xuandong Li. Steering symbolic execution to less traveled paths. In *OOPSLA*, pages 19–32, 2013.
- 58 Daniel Liew, Cristian Cadar, and Alastair F. Donaldson. Symbooglix: A symbolic execution engine for boogie programs. In *ICST*, 2016.
- 59 Kasper Luckow, Corina S. Păsăreanu, Matthew B. Dwyer, Antonio Filieri, and Willem Visser. Exact and approximate probabilistic symbolic execution for nondeterministic programs. In *ASE*, pages 575–586, 2014.
- 60 P. D. Marinescu and Cristian Cadar. KATCH: High-coverage testing of software patches. In *FSE*, pages 235–245, 2013.
- 61 Paul Dan Marinescu and Cristian Cadar. High-coverage symbolic patch testing. In *SPIN*, pages 7–21, 2012.
- 62 Paul Dan Marinescu and Cristian Cadar. Make test-zesti: A symbolic execution solution for improving regression testing. In *ICSE*, pages 716–726, 2012.
- 63 Frank J Massey Jr. The kolmogorov-smirnov test for goodness of fit. *JASA*, 46(253):68–78, 1951.
- 64 Zhelong Pan and Rudolf Eigenmann. Fast and effective orchestration of compiler optimizations for automatic performance tuning. In *CGO*, pages 319–332, 2006.
- 65 Corina S Păsăreanu and Neha Rungta. Symbolic pathfinder: symbolic execution of java bytecode. In *ASE*, pages 179–180, 2010.
- 66 David M Perry, Andrea Mattavelli, Xiangyu Zhang, and Cristian Cadar. Accelerating array constraints in symbolic execution. In *ISSTA*, pages 68–78, 2017.
- 67 Suzette Person, Matthew B. Dwyer, Sebastian Elbaum, and Corina S. Păsăreanu. Differential symbolic execution. In *FSE*, pages 226–237, 2008.
- 68 Suzette Person, Guowei Yang, Neha Rungta, and Sarfraz Khurshid. Directed incremental symbolic execution. In *PLDI*, pages 504–515, 2011.
- 69 John Platt. Sequential minimal optimization: A fast algorithm for training support vector machines. Technical Report MSR-TR-98-14, Microsoft Research, April 1998.
- 70 Rui Qiu, Sarfraz Khurshid, Corina S Păsăreanu, Junye Wen, and Guowei Yang. Using test ranges to improve symbolic execution. In *NFM*, pages 416–434. Springer, 2018.
- 71 Rui Qiu, Corina S Păsăreanu, and Sarfraz Khurshid. Certified symbolic execution. In *ATVA*, pages 495–511. Springer, 2016.
- 72 Rui Qiu, Guowei Yang, Corina S. Păsăreanu, and Sarfraz Khurshid. Compositional symbolic execution with memoized replay. In *ICSE*, pages 632–642, 2015.

- 73 Eric F. Rizzi, Mathew B. Dwyer, and Sebastian Elbaum. Safely reducing the cost of unit level symbolic execution through read/write analysis. *SEN*, 39(1):1–5, 2014.
- 74 Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A concolic unit testing engine for C. In *FSE*, pages 263–272, 2005.
- 75 Koushik Sen, George Necula, Liang Gong, and Wontae Choi. Multise: Multi-path symbolic execution using value summaries. In *FSE*, pages 842–853, 2015.
- 76 Junaid Haroon Siddiqui and Sarfraz Khurshid. ParSym: Parallel symbolic execution. In *ICSTE*, pages V1–405–V1–409, 2010.
- 77 Junaid Haroon Siddiqui and Sarfraz Khurshid. Scaling symbolic execution using ranged analysis. In *OOPSLA*, volume 47, pages 523–536, 2012.
- 78 Matt Staats and Corina Păsăreanu. Parallel symbolic execution for structural test generation. In *ISSTA*, pages 183–194, 2010.
- 79 Kunal Taneja, Tao Xie, Nikolai Tillmann, and Jonathan de Halleux. express: guided path exploration for efficient regression test generation. In *ISSTA*, pages 1–11, 2011.
- 80 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 *POPL*, volume 50, pages 83–95, 2015.
- 81 Nikolai Tillmann and Jonathan De Halleux. Pex–white box test generation for .net. In *TAP*, pages 134–153. Springer, 2008.
- 82 Lisa Torrey and Jude Shavlik. Transfer learning. *Handbook of Research on Machine Learning Applications and Trends: Algorithms, Methods, and Techniques*, 1:242, 2009.
- 83 Willem Visser, Jaco Geldenhuys, and Matthew B. Dwyer. Green: Reducing, reusing and recycling constraints in program analysis. In *FSE*, pages 58:1–58:11, 2012.
- 84 Jonas Wagner, Volodymyr Kuznetsov, and George Candea. -overify: Optimizing programs for fast verification. In *HotOS XIV*. EPFL-CONF-186012, 2013.
- 85 Tao Wang, Abhik Roychoudhury, Roland HC Yap, and Shishir C Choudhary. Symbolic execution of behavioral requirements. In *PADL*, pages 178–192. Springer, 2004.
- 86 Xiaoyin Wang, Lingming Zhang, and Philip Tanofsky. Experience report: How is dynamic symbolic execution different from manual testing? a study on klee. In *ISSTA*, pages 199–210, 2015.
- 87 Edmund Wong, Lei Zhang, Song Wang, Taiyue Liu, and Lin Tan. Dase: Document-assisted symbolic execution for improving automated software testing. In *ICSE*, pages 620–631, 2015.
- 88 Guowei Yang, Corina S Păsăreanu, and Sarfraz Khurshid. Memoized symbolic execution. In *ISSTA*, pages 144–154, 2012.

Type Regression Testing to Detect Breaking Changes in Node.js Libraries

Gianluca Mezzetti

Aarhus University, Denmark
mezzetti@gmail.com

Anders Møller

Aarhus University, Denmark
amoeller@cs.au.dk

Martin Toldam Torp

Aarhus University, Denmark
torp@cs.au.dk

Abstract

The npm repository contains JavaScript libraries that are used by millions of software developers. Its semantic versioning system relies on the ability to distinguish between breaking and non-breaking changes when libraries are updated. However, the dynamic nature of JavaScript often causes unintended breaking changes to be detected too late, which undermines the robustness of the applications.

We present a novel technique, *type regression testing*, to automatically determine whether an update of a library implementation affects the types of its public interface, according to how the library is being used by other npm packages. By leveraging available test suites of clients, type regression testing uses a dynamic analysis to learn models of the library interface. Comparing the models before and after an update effectively amplifies the existing tests by revealing changes that may affect the clients.

Experimental results on 12 widely used libraries show that the technique can identify type-related breaking changes with high accuracy. It fully automatically classifies at least 90% of the updates correctly as either major or as minor or patch, and it detects 26 breaking changes among the minor and patch updates.

2012 ACM Subject Classification Software and its engineering → Software libraries and repositories

Keywords and phrases JavaScript, semantic versioning, dynamic analysis

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2018.7

Supplement Material ECOOP Artifact Evaluation approved artifact available at <http://dx.doi.org/10.4230/DARTS.4.3.8>

Funding This work was supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No 647544).

1 Introduction

The world's largest software repository, npm,¹ hosts 475 000 Node.js JavaScript packages as of January 2018 and is used by millions of software developers. Most packages are libraries,

¹ <https://www.npmjs.com/>



and many are frequently updated, so versioning is essential to ensure that the components of a software system are compatible and up-to-date. The npm system encourages the use of *semantic versioning* [25], which distinguishes between patch, minor and major version updates: patch and minor are incremental updates that are not supposed to break any library client, whereas major version updates do not have any restrictions.

Unfortunately, the distinction between breaking and non-breaking changes is not always clear, and it may be difficult for the library developer to decide how to increment version numbers when changes are released. The dynamic nature of the JavaScript ecosystem often causes library developers to erroneously believe that their updates cannot break any clients. A prominent example was the supposedly minor update of a package called *debug* from version 2.3.3 to 2.4.0 on December 14 2016. Due to a simple spelling error, all clients that tried to load the new version crashed immediately.² The bug was fixed within an hour, but *debug* was downloaded more than 27 million times in December 2016 alone, which means that it may still have affected thousands of installations.

To make matters worse, Node.js library interfaces are rarely specified precisely, so library developers and their clients may have different views on which aspects of the library are supposed to be internal to the library and which aspects the client code may rely on. As an example, the developers of the popular package *React* from version 15.3.2 to 15.4.0 reorganized the module `react/lib/ReactDOM` that was intended for internal use only, but numerous other packages used that module and therefore broke.³

In statically typed programming languages like Java, even without versioning systems, the type system is helpful for detecting many situations where an upgrade of a library causes an application to break. For example, if the type signature of a library method has changed, the application code no longer compiles. Using access modifiers enables library developers to encapsulate private parts of the library, so that internal data representations and operations can be changed without affecting client code. Additionally, annotations about deprecated functionality signal to the application developer that attention is needed. Java's binary compatibility conditions [13, Chapter 13] and tools like JAPICC [23] make it possible to detect type-related breaking changes in libraries even without involving the client code. In the world of JavaScript and npm, there is no static type system or compilation to binary code, so incompatibility issues are often not detected until runtime.

We distinguish between two main categories of breaking changes. *Type-related* breaking changes are modifications of a library that affect the presence or types of functions or other properties in the library interface. Such changes include renaming a public function, moving it to another location, or changing its type signature. Type-related breaking changes should evidently always be reflected as major version updates. As the library interface is partly defined by the library initialization code, initialization errors like the one in the *debug* example are generally categorized as type-related breaking changes. *Semantic* breaking changes are modifications that are not type-related but affect the library functionality in other ways that may cause clients to malfunction. This category is more blurry, as it depends on a semantic contract between the library and the clients. The type checker in Java detects type-related breaking changes, but not semantic ones. Our goal is to provide a mechanism that can similarly detect type-related breaking changes for Node.js JavaScript libraries, without requiring type annotations.

² <https://github.com/visionmedia/debug/issues/347>

³ <https://github.com/supnate/rekit/issues/16>

In this paper, we first present a preliminary study of real-world breaking changes in the npm repository. The study shows that breaking changes do occur at patch and minor updates, and that a significant portion of the breakage is type-related. Next, we propose a technique, called *type regression testing*, to automatically detect such type-related breaking changes, which we call *type regressions*, thereby gaining some of the benefits that are known from statically typed languages.

Our type regression testing technique is based on a novel dynamic type analysis that automatically learns relevant information about the API of a given library. The basic idea is quite simple: The npm repository makes it possible to identify packages that directly or transitively depend on the library of interest. (For example, the *lodash* library has more than 50 000 direct dependents.) By exercising the test suites for those packages, we can monitor the dynamic execution and construct a model of the library API. When the library implementation has been updated and a new version is about to be released, the test suites are run again, this time with the updated library to infer a new model of the new API. We then compare the new and the old API models using certain rules to identify breaking changes in the update. Importantly, we do not use the library's own test suite, but the test suites of the clients, because they are more likely to provide representative executions and only use the public parts of its API. Type regression testing *amplifies* the existing test suites: even if the tests do not fail, it can identify type-related changes in the interactions between the clients and the library.

Making this idea work in practice requires a suitable notion of models of library APIs, together with a mechanism for comparing models before and after the library implementations have been updated. The API modeling and the comparison mechanism need to be aligned with how JavaScript library developers usually organize their code and try to adhere to the semantic versioning guidelines.

In summary, the contributions of this paper are as follows.

- We first present a preliminary experimental study on the prevalence and the kinds of breaking changes in the npm repository. We find that at least 5% of all packages have been affected by a breaking change in a minor or patch update of a dependency, and that a majority of these breaking changes are due to changes in the public API of the package.
- We propose *type regression testing* as a mechanism for leveraging the preexisting test suites of npm packages that depend on a JavaScript library of interest, to learn models of the library API and detect likely breaking changes.
- At the core of the type regression testing mechanism, library APIs are modeled using *dynamic access paths* and types that provide information about how the library and the clients interact. We define precisely how these models are obtained and compared. The possible breaking changes are identified by *type regressions*: changes in the type signatures of the library APIs that are incompatible with the mutual expectations of client and library developers.
- We report results from an experimental evaluation of the approach on 12 of the most depended upon libraries in the npm system, demonstrating that it can detect type-related breaking changes with high accuracy. Our implementation, named `NOREGRETS`, classifies at least 90% of the updates correctly as either major or as minor or patch, and it also successfully identifies 26 breaking changes among the minor and patch updates. Moreover, in cases where a likely breaking change is detected, the warning message produced by the tool pinpoints the involved part of the library, which aids diagnosis.

2 A Preliminary Experimental Study of Breaking Changes

To understand how frequently library updates break client applications, and what the typical causes of the breaking changes are, we conducted a large experiment on npm package updates. For this experiment, we exploited the fact that many clients have test suites and that if a test fails after updating a dependency, then the failure indicates that the update contains a breaking change.

To serve as clients, a sample consisting of 4616 packages with test suites was randomly picked from npm. Most of these packages are intended to be used as libraries, but they still depend upon other libraries and have test suites as required for this study.

All npm packages include a configuration file called `package.json` where the package dependencies are specified. Each dependency specification consists of a package name and a versioning constraint. This constraint specifies the range of versions which the dependency must lie within. The npm system will always install the newest version of the dependency satisfying this versioning constraint. According to the semantic versioning guidelines, clients should use constraints that permit all minor and patch updates but no major updates. Thereby the client will automatically receive the bug fixes and other improvements introduced in minor and patch updates, but hopefully never break since breaking changes are only allowed in major updates.

For each dependency specified in the `package.json` file, we ran the test suites using each version of the dependency going from the oldest to the newest version satisfying the semantic versioning constraint. We removed versions which, although satisfying the semantic versioning constraint, would never have been installed by npm in practice since a newer version was already available at the point where the constraint was created. An update was flagged as potentially breaking whenever the test suite of the client went from all tests passing to at least one test failing after applying the update. We did not count at the granularity of individual test cases, since it is technically difficult to do because of nested and parameterized tests.

This amounts to 75 913 executions of test suites of the 4616 packages. Of these, 430 fail, meaning that breaking changes are detected. Whenever we had two versions of the same client appearing among the failures, we discarded the oldest one of them to avoid duplicates. We also discarded major updates and updates containing a pre-release identifier. A pre-release identifier is a hyphen followed by a tag added to the end of a version number used to indicate, for example, release candidates and beta versions. Updates containing a pre-release identifier should be treated as a major updates according to the semantic versioning principle. Furthermore, we chose to exclude 94 failures that could not be reproduced consistently due to flaky tests in the clients. With these filters, the total number of failures was reduced to 263 affecting 259 different clients. None of these 263 breaking changes should have appeared if semantic version had worked as intended and the library developers and the client developers had a common agreement on what is the public interface of the libraries!

Thus, at least 5% of the npm packages have experienced a breaking change due to a non-major dependency update. The actual number is likely much higher, because not many packages have test suites that are thorough enough to catch all breaking changes in libraries they depend on.

Next, we manually categorized the test failures as either *type-related*, *semantic*, or *unknown*, the latter for the cases where we could not determine the cause within 30 minutes. We consider any update that modifies the presence or types of modules, properties, function arguments, and function return values as type-related. Specifically, an update that relocates

a module to a different path typically causes attempts to load the module using the old path to fail. Any test failure that is not type-related is considered semantic. Thus, the type-related test failures roughly correspond to the kinds of errors that could be caught statically if using a language like Java with type checking instead of JavaScript.

As result we found that among the 263 failures, 176 were type-related, 37 were semantic, and the remaining 50 were marked as unknown. Some type-related breaking changes are easy to detect. In particular, sometimes simply attempting to load a library module fails, because it has been relocated or because its initialization code consistently crashes after an update. An example of the latter is the bug in the *debug* package mentioned in Section 1, which alone accounts for 101 of the failures. Even if we only count the occurrence of this bug once, we still find that at least 46% of the breaking changes are type-related.

This preliminary study motivates the need for tool support to detect breaking changes in Node.js libraries before the developers publish new versions of their libraries. It also justifies focusing on breaking changes that are type-related, which are more amenable to automated detection than the semantic ones.

To our knowledge the only similar tool is *dont-break*,⁴ which follows essentially the approach we have used in our preliminary study: it detects breaking changes simply by running the test suites of library clients each time a new version of the library is about to be released. Although this is a simple approach, it has an important limitation: The library developer presumably has no knowledge of the clients, let alone their tests, so it can be difficult for him or her to identify the relevant parts of the library whenever a client test fails. In contrast, type regression testing precisely pinpoints the involved changes made at the library interface, allowing the library developer to decide whether or not the breaking change is intended, without having any knowledge of the client code. Additionally, type regression testing can detect breaking changes that do not necessarily surface as failing client tests and can therefore also be viewed as a test amplification mechanism.

3 Motivating Example

Consider the following subtle change of the `isIterateeCall` method in the *lodash* library when upgraded from 3.2.0 to 3.3.0—a minor update that should not introduce breaking changes.

```
1 // lodash 3.2.0
2 function isIterateeCall(value, index, object) {
3   ....
4   return prereq && object[index] === value;
5 }

6 // lodash 3.3.0
7 function isIterateeCall(value, index, object) {
8   ....
9   var other = object[index];
10  return prereq && (value === value ? value === other : other !== other);
11 }
```

The variable `prereq` is computed in the same way in both versions. The important difference is that the `object[index]` property lookup is only executed when `prereq` is `true` in version 3.2.0, due to short-circuiting of `&&`, whereas it is always executed in version 3.3.0. If `index` is an object, then the `toString` method is implicitly called on `index` to coerce it to a string such

⁴ <https://www.npmjs.com/package/dont-break>

7:6 Type Regression Testing to Detect Breaking Changes in Node.js Libraries

that it can be used in the property lookup. However, it is possible that there is no `toString` method on `index` if, for example, the `index` value was created using `Object.create(null)`. Consequently, a type error exception will be thrown when the coercion is attempted.

The `isIterateeCall` method is not directly visible to the client code, but it is called internally by the public `merge` method that forwards one of its arguments as the `index` parameter of `isIterateeCall`. The `merge` method takes a target object and a variadic number of source objects and merges the properties of the source objects into the target object. The artificial library client in lines 13–18 witnesses the problem.

```
12 // client
13 var l = require('lodash');
14 var o1 = {};
15 var o2 = {};
16 var oBad = Object.create(null);
17 var o4 = {};
18 l.merge(o1, o2, oBad, o4); // Type error in lodash 3.3.0
```

The `oBad` object is passed as the `index` parameter to `isIterateeCall`, and a runtime type error appears when using *lodash* 3.3.0 but not when using the older version. The problem has been confirmed by the developers who fixed it in *lodash* 3.3.1 as mentioned in the changelog.⁵

Type regression testing automatically finds this problem as follows. First, it builds a model of the library API by observing the executions of the tests of clients, for both the old and the new library version. Second, it compares the two models to detect breaking changes. This particular breaking change is detected by observing that the *lodash* 3.3.0 model, unlike the one generated by version 3.2.0, requires the `toString` method to be present on the third argument of `merge`. This expectation is breaking since the third parameter is in a contravariant position and its type is more specific than before the update.

It is important to notice that type regression testing leverages and amplifies preexisting test suites. For this *lodash* bug, the breaking change is detected even though the execution of `merge` does not trigger the type error in any of the client tests.

Furthermore, even if one of the client tests had triggered the type error, that would only produce a stack trace indicating a problem in `isIterateeCall`, and manual effort would then be required to be connect this type error to the third argument of `merge`. In contrast, type regression testing identifies exactly the `toString` property on the third argument of `merge` as the source of the problem.

4 Overview

Type regression testing targets a specific use case: a library developer is ready to release a new library version and wants to know whether the new implementation introduces breaking changes. The workflow of type regression testing comprises two phases.

(1) Public API Discovery. In the first phase, all the packages with tests that depend on the old library version are retrieved from npm. The type regression in the motivating example can be spotted by using any client whose tests cause the invocation of the *lodash* `merge` function, for example *strong-params* at version 0.7.0.⁶ Then, a public API model of both the old and the new library version is built using an instrumented interpreter that runs the

⁵ <https://github.com/lodash/lodash/wiki/Changelog#v331>

⁶ <https://github.com/ssowonny/strong-params>

tests of all the collected dependents. A model π , which we formally define in Section 5, is a map from *dynamic access paths* to *types*.

Intuitively, a dynamic access path is an abstraction of the sequence of actions that are performed to obtain a value. Types are an abstraction over values that is more specific than the ordinary notion of runtime types in JavaScript. For example, the path $p = \text{require}(\text{lodash}).\text{merge}(3)_4.\text{toString}$ exhibits the regression in the example from Section 3. This path denotes any value obtained by accessing the `toString` property of the third argument passed to the `merge` function when `merge` is invoked with four arguments. No type is associated with this path in the execution of the tests of *strong-params* when using *lodash* 3.2.0, but a function is observed when switching to *lodash* 3.3.0. In the model π of the public API of *lodash* 3.2.0, we have $\pi(p) = \circ$, denoting the fact that no value has been observed for p . In the model π' of the public API of *lodash* 3.3.0, we instead have $\pi'(p) = \text{function} \wedge \text{object}$, which means that the value is an instance of `function` and `object` (formally $\pi'(p)$ is an intersection type). This phase is fully detailed in Section 5.

(2) Type Regression Detection. In the second phase, we compare the models π and π' obtained from the old and the new library version to report *type regressions*, which are indications of type-related breaking changes. Type regressions are detected by comparing $\pi(p)$ and $\pi'(p)$ for every dynamic access path p , using a notion of subtyping, which we define in Section 5.2. In the example above, the type regression is reported because the type `function` \wedge `object` is not a supertype of \circ . This phase is fully detailed in Section 6.

5 Public API Discovery

The workflow described in Section 3 requires two models of the public API to be built, one for the old and one for the new library version. It is important that the API models only capture the publicly available API, so that the comparison is not susceptible to changes in the private parts of the library where modifications are always allowed.

In statically typed languages, like Java, the package structure, class hierarchy, and access modifiers statically identify the public API of a library. In a dynamically typed language, such as JavaScript, a library module is initialized by the execution of the library module itself. Specifically, in Node.js, the library code for initialization of a library *foo* is executed the first time `require("foo")` is called. The object stored by the library code in the `module.exports` variable is the one returned by the call to `require`. However, this object only exposes the immediately accessible part of the public API, and the public API often contains much more functionality. For example, all methods of router objects in the *express* library only become accessible after invoking `Router()`.⁷ As this example illustrates, it is generally difficult to statically identify the public API, which is why we resort to dynamic analysis.

The model of the public API of a library is a map $\pi : \mathbf{Path} \rightarrow \mathbf{Type}$ assigning a type $t \in \mathbf{Type}$ to each dynamic access path $p \in \mathbf{Path}$. We now define dynamic access paths and types, and then we describe the discovery mechanism that builds π .

5.1 Dynamic Access Paths

Dynamic access paths, hereafter also shortened to paths, are used in the model to refer to values that are part of the interface between the client and the library. This mechanism

⁷ <http://expressjs.com>

7:8 Type Regression Testing to Detect Breaking Changes in Node.js Libraries

ignores the syntactical structure of the library code and only considers how the library is being accessed dynamically by the client code.

► **Definition 1** (Dynamic Access Path). A *dynamic access path* $p \in \mathbf{Path}$ is a possibly empty sequence of *actions* α that abstractly represents a set of values, as defined by the grammar below. We indicate integers by the letters i, j and strings (library names and property names) by the letter n .

$$\begin{aligned} p &::= \varepsilon \mid \mathbf{require}(n) \mid p\alpha && p \in \mathbf{Path} \\ \alpha &::= .n \mid ()_i \mid \mathbf{new}()_i \mid (j)_i \mid .* \end{aligned}$$

At runtime, values are associated with paths, and conversely, each path represents a set of values, as defined by recursion on the structure of paths:

- ε : the empty path ε denotes any value that is not used for modeling the library API.
- $\mathbf{require}(n)$: the objects returned by the `require` function when passing the library name n as argument.
- $p.n$: the values obtained when accessing a property of name n of an object denoted by the path p .
- $p()_i$: the values returned by calling a function denoted by the path p when the function is called with i arguments.
- $p_{\mathbf{new}}()_i$: the values returned by calling a function denoted by the path p as a constructor with i arguments (i.e., using the `new` keyword in the call).
- $p(j)_i$: the values of the j 'th argument passed to a function that is denoted by the path p and called with i arguments (similarly, $p_{\mathbf{new}}(j)_i$ represents values of constructor arguments).
- $p.*$: the elements of the arrays denoted by p .

► **Example 2.** The following listing shows a client that uses a hypothetical library `twice`, which has a single method `twice` that takes an object and returns an object with the property `res` that contains the doubled value of the `x` property of the argument object.

```
19 // Twice library
20 module.exports.twice = function(t) {
21   return { res : t.x * 2 };
22 }
```

```
23 // Client
24 var m = require("twice");
25 var a = {x: 42, y: 43};
26 var b = m.twice(a);
27 a.y + b.res;
```

Consider which parts of the `twice` library are part of its public API. The value of the property `b.res` must be part of the public API, because `b` is coming from the library and `res` is accessed by the client on line 27. Intuitively, the client expects the `res` property to be available on the return value of the call on line 26. This value is given the path $\mathbf{require}(\mathit{twice}).f()_1.\mathit{res}$. Likewise, the value of the property `a.x` is also part of the public API with the path $\mathbf{require}(\mathit{twice}).f(1)_1.x$. The `twice` method reads the `x` property, so it should be present on the argument passed to `twice`. The value of the property `a.y`, instead, is not part of the library API since it is never read by the `twice` method (even though `a` is passed to `twice`). We describe in detail our mechanism for distinguishing between public and private parts of the API in Section 5.3.

Note that a value can be given different paths during a single program execution, and multiple values can be denoted by the same path. We include the number of arguments in the function invocation action, to distinguish two invocations of the same function with different numbers of arguments. JavaScript developers often define variadic functions where the function behavior changes depending on the number of arguments. For example, consider the `map` function of *lodash*, which implements the standard higher-order map function. In *lodash*, the function implicitly chooses the identity function as its function argument if no other argument is supplied. Hence, it is beneficial for the precision of the API model to distinguish an invocation of `map` where a function argument is supplied from one where no function argument is supplied. We leave more complex forms of overloading to future work.

5.2 Types

We use a notion of types that extends the basic types of ECMAScript (strings, numbers, objects, etc.) with types that have a special meaning in Node.js, for example, arrays, sets, maps, event-emitters, and streams. We also include intersection types that are used to capture the prototype hierarchies of objects, as explained later. Union types are used to easily join different observations.

► **Definition 3 (Types).** A *type* $t \in \mathbf{Type}$ is a term in the following grammar:

$$\begin{aligned}
 t &::= \circ \mid b && t \in \mathbf{Type} \\
 b &::= b \vee b' \mid b \wedge b' \mid \mathbf{undefined} \mid \mathbf{string} \mid \mathbf{boolean} \mid \mathbf{number} \mid \mathbf{object} \mid \mathbf{function} \\
 &\quad \mid \mathbf{array} \mid \mathbf{set} \mid \mathbf{map} \mid \mathbf{event-emitter} \mid \mathbf{stream} \mid \mathbf{throws}
 \end{aligned}$$

To simplify the presentation, we only show a representative subset of the Node.js types. The type $b \vee b'$ is a union type, while $b \wedge b'$ is an intersection type. The type \circ has a special use: If a path p is ascribed the type \circ in a model (i.e., $\pi(p) = \circ$), then no values represented by the path have been observed during client test execution, therefore they are assumed not to be part of the public interface of the library. (Note that the special type \circ and the special path ε are both used for identifying the library API; \circ is used in the generated models, and ε is used for tagging values in our instrumented interpreter as explained in Section 5.3.) The special type `throws` is ascribed to functions that throw exceptions. Considering exceptions as a special type is unusual, but as we demonstrate in Section 7, it fits our setting well.

The function $type(v)$ gives the type of a runtime value v . The function assigns the corresponding type to primitive values, e.g., `string` to strings, but it does more for objects and functions to account for prototype inheritance: for example, the type of a function is `function` \wedge `object` because functions are also objects, and similarly, the type of a set is `set` \wedge `object`.

Types do not distinguish between boxed and unboxed primitive values and do not represent function types explicitly by an arrow type as usually done in type systems. As explained above, the parameters and return types of a function are instead expressed as different paths.

As mentioned in Section 4, we use subtyping to detect changes in the public API of a library that are breaking. The intuition, according to the Liskov substitution principle, is that subtyping should satisfy substitutability. Informally, if t' is a subtype of t , denoted $t' <: t$, then values of type t may be replaced with values of type t' without affecting the desirable properties of the program [21]. In our case, if a library method returns a value of type t in the old version and a value of type t' in the new version where $t' <: t$, then behavioral subtyping tells us that there is no type-related breaking change.

7:10 Type Regression Testing to Detect Breaking Changes in Node.js Libraries

JavaScript is a dynamic languages, with many different programming styles used by library developers, and there is no canonical notion of subtyping that perfectly fits all styles. This problem arises also in optionally typed languages, such as TypeScript, where the type checker has more than 10 different options that library developers can customize to better match their programming styles.⁸ The subtyping relation that we use in our implementation is defined below, although we envision that library developers may want to customize some of the rules when adopting the type regression testing technique.

► **Definition 4 (Subtyping).** The subtyping relation $<$: among types is the relation given by the reflexive, transitive closure of the following rules.

$$\begin{array}{c} b <: b \vee b' \qquad b' <: b \vee b' \qquad b \wedge b' <: b \qquad b \wedge b' <: b' \\ \\ \frac{b <: b'' \quad b' <: b''}{b \vee b' <: b''} \qquad \frac{b'' <: b \quad b'' <: b'}{b'' <: b \wedge b'} \qquad \text{object} <: \text{undefined} \qquad t <: \circ \end{array}$$

The rules for intersection and union types are standard. Note that the subtyping relation, because of union and intersection types, is not anti-symmetric [15]: for example $b \wedge b' <: b' <: b \wedge b'$ whenever $b' <: b$. Consequently, types are not an order under the subtyping but only a preorder. Therefore, the least upper bound (also called join), which is needed for inference and checking, is not unique. For this reason, we implicitly work on the quotient order constructed from the preorder whenever we use the join operator \sqcup . The rule $\text{object} <: \text{undefined}$ is motivated by the following example, and $t <: \circ$ is relevant for Example 9.

► **Example 5.** Consider the patch update of the `express` library from 3.0.1 to 3.0.2:

```
28 // express 3.0.1
29 app.use = function(route, fn){
30   ...
31   return this._router.route.apply(this._router, args);
32 }
33 // express 3.0.2
34 app.use = function(route, fn){
35   ...
36   this._router.route.apply(this._router, args);
37   return this;
38 }
```

The return type of `get` changes from `undefined` (the value returned by the `route` function is `undefined`) to `object` \wedge `function` (the type of the `this` value). The reason the developers of `express` introduced this modification was to enable cascading of method calls, i.e., the ability to write `app.use(...).use(...)`. This update is clearly non-breaking, and the subtyping rule for `undefined` ensures that type regression testing does not consider this as a breaking change for `express` because `function` \wedge `object` $<$: `object` $<$: `undefined`.

Note that, although unlikely, it is possible to write a client that relies on the fact that `use` returns `undefined` rather than a function, so that some library developers might prefer to be warned about a possible breaking change in this case. The simple example shows that it may be worthwhile to allow JavaScript developers to customize the subtyping rules.

⁸ <https://www.typescriptlang.org/docs/handbook/compiler-options.html>

5.3 Instrumented Interpreter

We will explain our approach for public API discovery through the rules of an instrumented interpreter on a subset of the JavaScript language. At the beginning of the execution of the client tests, the model π assigns the type \circ to all paths. As values are determined to be part of the public API, the corresponding path is assigned a new type using the *type* function. Eventually, at the end of the test execution, π will hold a model of the public API of the library, corresponding to the subset that the client has used in the tests.

It is important to notice that the model which we build may not exactly match what the library developer intended as the public API. For example, if the library developer states in the documentation that clients are not supposed to read a certain value, but some client does it anyway, then the value will be considered part of the public API. Without a formal and generally accepted mechanism for specifying and enforcing encapsulation, any client usage taking place in practice has to be regarded as legitimate.

We describe the evaluation of a representative subset of JavaScript language constructs in A-normal form, i.e., where every subterm has been evaluated to a value [11]. For simplicity, we focus on a core language and do not explain the instrumentation of binary operators, constructor invocations, writes to local variables, exceptions, functions with multiple parameters, etc. The purpose of the following definitions is to explain the API discovery mechanism as an instrumentation of an existing JavaScript interpreter; in particular we are here not interested in all the details of JavaScript semantics, which are described elsewhere [14, 6].

► **Definition 6** (A-Normal Forms). The A-normal forms considered are the ones below; v denotes values, and c denotes constants.

$e ::= c$	(literal constant)	$e \in \mathbf{Exp}$
x	(variable read)	
$v[v]$	(property access)	
$v[v] = v$	(property update)	
$v(v)$	(function call)	

The instrumented interpreter is defined by a big-step operational semantics. We assume that the original semantics uses judgments of the form $\langle e, \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle$, which associate an initial term e , an environment ρ , and a store σ with a resulting value v and a store σ' . The environment ρ maps variable identifiers to primitive values (strings, numbers, booleans, and undefined) or to store locations l . The store σ maps locations to objects and closures. Objects are similar to environments, mapping identifiers to primitive values or to store locations.

The instrumented semantics $\langle e, \rho, \sigma, \pi \rangle \Downarrow^I \langle v, \sigma', \pi' \rangle$ extends the semantics with the π, π' components where π' is the computed public API model. The initial model π , used at the beginning of the program execution, associates the type \circ with each path p . Similarly to information flow analyses, the instrumented interpreter uses tagged values v^p where $p \in \mathbf{Path}$ is a path, in place of the original values [5]. Values v^p and $v^{p'}$ are indistinguishable by JavaScript programs for any p, p' . Recall that ε is called the empty path and is intended to represent values that are not part of the public API.

The key idea about the use of tagged values is to preserve the following property: If a value v^p is read in the client code and has a nonempty path p , then the value has been retrieved through the use of the public library API. Vice versa, if a value v^p in the library code has a nonempty path, then the value has been passed to the library through the public library API.

7:12 Type Regression Testing to Detect Breaking Changes in Node.js Libraries

Our instrumentation ensures that values that are passed between library and client code through the public API are always assigned a nonempty path. Initially, the only value with a nonempty path is the value returned by `require` when the library is first loaded.

► **Example 7.** Suppose the library `foo` is used by a client as follows:

```
39 var lib = require(foo);
40 var x = new lib.c();
41 lib.m(x);
```

The function values of `lib.m` and `lib.c` are tagged with `require(foo).m` and `require(foo).c`, respectively. The object value of `x` is initially assigned the path `require(foo).cnew()o` because it is a value returned by the library constructor `c`. Assume the object has properties that are written by `c` and read by `m`, and are not accessed by the client code. Such properties should be considered private to the library, so that changes to them in a library update are not treated as type regressions. This scenario is detected by our instrumented interpreter by observing that `x` already has a nonempty path when it is passed to `m`. Therefore, the path of `x` is emptied (set to ε) to avoid the object being considered part of the public API.

In general, whenever a value crosses the API a second time, the path of that value is set to ε , so that its usages are not recorded as being part of the public API.

The semantic rules for the instrumented interpreter are shown in Figure 1. We use the notation $\{p \mapsto t\}$ meaning the model that maps the path p to the type t and all other paths to \circ . The join $\pi \sqcup \pi'$ of two models π, π' is the pointwise join of the types for each path. By a slight abuse of notation, when joining the type \circ with another type we assume that it behaves as the unit, i.e. if $\pi(p) = \circ$ and $\pi'(p) = t$ then $(\pi \sqcup \pi')(p) = t$.

When evaluating a constant, it is tagged by the empty path (rule `CONST`). When evaluating a variable, the tagged value is looked up in the environment (rule `VAR`). The model is unaffected in both cases.

At property accesses, the value of a property needs to be retrieved from a target object, and the model should be updated to reflect the type of the property accessed whenever the target object is part of the public library API. The rule `ACCESS` uses an auxiliary function **Lookup** to handle the actual lookup of the property, possibly walking the prototype chain, to retrieve the value $v_r^{p_r}$ of the property $v_f^{p_f}$ of the object $v_t^{p_t}$ in the store σ . (We omit formal definitions of this and other auxiliary functions since they are not important for the tagging mechanism.) The rule distinguishes three cases for the path p_r' that is used as tag for the resulting value v_r . Assume the property access occurs in the client code. In the first case, p_t is nonempty and p_r is empty, meaning that the object v_t comes from the library, and the library code that wrote the property value v_r to the object did not obtain that value from the client, so in this case the path p_r' is set to $p_t\alpha$. The recorded action α is $.v_f$ where v_f is the name of the property accessed, unless v_t is an array, in which case the special array access action $.*$ is used. In the second case, p_t and p_r are both nonempty. This means that we are accessing a property of a library object, and the library code that wrote the property value to the object obtained that value from the client, so we are in a situation similar to Example 7, and we set p_r' to ε accordingly. In the third case, p_t is empty, which means that the object v_t comes from the client, so we simply keep the existing path p_r for the resulting value. If the property access instead occurs in the library code, the reasoning is the same, but with the roles of client and library swapped. In either case, the π component is updated to reflect that a value of type $type(v_r)$ has been observed for the path p_r' .

At property updates, a value is assigned as property on a target object. The rule `UPDATE` uses the auxiliary function **Update** to perform the actual update of the property $v_f^{p_f}$ of the

$$\begin{array}{c}
\text{CONST} \\
\langle c, \rho, \sigma, \pi \rangle \Downarrow^I \langle c^\varepsilon, \sigma, \pi \rangle \\
\\
\text{VAR} \\
\frac{\rho(x) = v^p}{\langle x, \rho, \sigma, \pi \rangle \Downarrow^I \langle v^p, \sigma, \pi \rangle} \\
\\
\text{ACCESS} \\
\text{Lookup}(v_t^{p_t}, v_f^{p_f}, \sigma) = v_r^{p_r} \\
p_r' = \begin{cases} p_t \alpha & \text{if } p_t \neq \varepsilon \wedge p_r = \varepsilon \\ \varepsilon & \text{if } p_t \neq \varepsilon \wedge p_r \neq \varepsilon \\ p_r & \text{if } p_t = \varepsilon \end{cases} \quad \alpha = \begin{cases} .* & \text{if } v_t \text{ is an array} \\ .v_f & \text{otherwise} \end{cases} \\
\pi' = \pi \sqcup \{p_r' \mapsto \text{type}(v_r)\} \\
\hline
\langle v_t^{p_t} [v_f^{p_f}], \rho, \sigma, \pi \rangle \Downarrow^I \langle v_r^{p_r'}, \sigma, \pi' \rangle \\
\\
\text{UPDATE} \\
p_a' = \begin{cases} \varepsilon & \text{if } p_t \neq \varepsilon \\ p_a & \text{if } p_t = \varepsilon \end{cases} \\
\text{Update}(v_t^{p_t}, v_f^{p_f}, v_a^{p_a'}, \sigma) = \sigma' \\
\hline
\langle v_t^{p_t} [v_f^{p_f}] = v_a^{p_a'}, \rho, \sigma, \pi \rangle \Downarrow^I \langle v_a^{p_a'}, \sigma', \pi \rangle \\
\\
\text{CALL} \\
p_a' = \begin{cases} p_f(1)_1 & \text{if } p_f \neq \varepsilon \wedge p_a = \varepsilon \\ \varepsilon & \text{if } p_f \neq \varepsilon \wedge p_a \neq \varepsilon \\ p_a & \text{if } p_f = \varepsilon \end{cases} \\
\text{Call}(v_f^{p_f}, v_a^{p_a'}, \sigma, \pi) = \langle v_r^{p_r}, \sigma', \pi' \rangle \\
p_r' = \begin{cases} \text{require}(v_a) & \text{if } v_f = l_{\text{require}} \\ p_f(1)_1 & \text{if } v_f \neq l_{\text{require}} \wedge p_f \neq \varepsilon \wedge p_r = \varepsilon \\ \varepsilon & \text{if } v_f \neq l_{\text{require}} \wedge p_f \neq \varepsilon \wedge p_r \neq \varepsilon \\ p_r & \text{if } v_f \neq l_{\text{require}} \wedge p_f = \varepsilon \end{cases} \\
\pi'' = \pi' \sqcup \{p_a' \mapsto \text{type}(v_a)\} \sqcup \{p_r' \mapsto \text{type}(v_r)\} \\
\hline
\langle v_f^{p_f} (v_a^{p_a'}), \rho, \sigma, \pi \rangle \Downarrow^I \langle v_r^{p_r'}, \sigma', \pi'' \rangle
\end{array}$$

■ **Figure 1** Semantic rules of the instrumented interpreter.

object $v_t^{p_t}$ with the value $v_a^{p_a'}$ in the store σ , resulting in a new store σ' where the object property has been updated. The path p_a' of the value being assigned is selected as follows. If the path p_t of the object is nonempty, then we are intuitively sending a value across the boundary between client and library. If p_a is also nonempty, it means that the value now crosses the API boundary a second time, so we set p_a' to ε . In all other cases, we simply

preserve the existing path p_a . (In particular, this means that in the situation where p_t is nonempty and p_a is empty, p'_a becomes ε , which is the only sensible choice with our current language of dynamic access paths.)

The rule **CALL** models the instrumentation of function calls. We use the auxiliary function **Call** to perform the actual call to the function $v_f^{p'_a}$ with argument $v_a^{p'_a}$, store σ , and initial model π . The argument value v_a is tagged by the path p'_a , which is selected as follows. Assume, without loss of generality, that the call occurs in the client code. In the first case of the definition of p'_a , p_f is nonempty and p_a is empty, meaning that the function v_f comes from the library and the argument value v_a comes from the client. In this case, the path p'_a is set to $p_f(1)_1$, indicating that v_a is used as first argument to the library function p_f when executing the function body. (This generalizes naturally to functions with multiple arguments.) The two remaining cases follow the same reasoning as for p'_r in rule **ACCESS**. The auxiliary function **Call** returns a value $v_r^{p'_r}$, a updated store σ' , and an updated model π' . The path p'_r used to tag the resulting value v_r is decided depending on the the function v_f , its path p_f , and the path p_r of the returned value. If v_t is the **require** function, written $v_t = l_{\text{require}}$, then the path p'_r is **require**(v_a). Otherwise, the path is either set to ε (according to the principle already discussed according to which the values crossing the API boundary twice are given an empty path), $p_f()_1$ (if the call is to a library function and the resulting value came from the library side), or kept as p_r . The resulting model π'' collects the observations from the execution of the function and the types of the function argument and the resulting value.

As described above, the instrumented interpreter generates a model for each client test being run. A complete model for a library version is made by joining all the models of the client tests using the \sqcup operator.

6 Type Regression Testing

Every mapping in a model π establishes a sort of mutual expectation between the client and the library. The direction of such an expectation depends on the path structure. For example, if $\pi(\text{require}(lib)) = t$, then it is the client that expects an object of type t from the library lib . The same direction applies to properties accessed on the object returned from **require**. For example, if $\pi(\text{require}(lib).p) = t$, then the client expects that the library has a property p of type t . The direction of the expectation flips upon an argument action. For example, if $\pi(\text{require}(lib).p(j)_i) = t$, then it is the library that expects the client to pass a value of type t .

The direction of the expectation affects the direction of the subtyping to use when checking for type regressions in library updates, much like covariance and contravariance in standard type checking. For example, if $\pi(\text{require}(lib)) = t$ in version 1.0.0 and $\pi'(\text{require}(lib)) = t'$ in version 1.1.0, then we have to check that $t' <: t$. Symmetrically, if the type for $\pi(\text{require}(lib)(j)_i) = t$ in version 1.0.0 and it is $\pi'(\text{require}(lib)(j)_i) = t'$ in version 1.1.0, then we have to check that $t <: t'$ because the library is certainly allowed to relax its expectations on minor version upgrades but it cannot require a more specific type.

We use the relation $<:_p$ to ensure that the direction of the typing relation is correct. It is inductively defined on the structure of the path p . The direction is switched on every argument action, as with contravariance for traditional function subtyping. In the base case where the path is empty, the ordinary subtyping relation $<:$ is used:

$$\frac{t <: t'}{t <:_\varepsilon t'} \qquad \frac{\alpha = (j)_i \quad t' <:_p t}{t <:_p \alpha t'} \qquad \frac{\alpha \neq (j)_i \quad t <:_p t'}{t <:_p \alpha t'}$$

We can now define precisely how to detect type regressions. Note that type regressions are never reported for empty paths, because empty paths represent values that are not part of the public API of the library.

► **Definition 8** (Type Regression). Let π and π' be models of an old and a new library version, respectively. A nonempty path p exhibits a *type regression* whenever $\pi'(p) \not\prec_p \pi(p)$.

► **Example 9.** Continuing the example from Section 3, the path where the type regression is detected is $p = \text{require}(\text{lodash}).\text{merge}(3).\text{toString}$. In the old model $\pi(p) = \circ$, while in the new model $\pi'(p) = \text{function} \wedge \text{object}$. By definition, the path p exhibits a type regression: $\text{function} \wedge \text{object} \not\prec_p \circ$ because $\circ <: \text{function} \wedge \text{object}$ does not hold.

Note that if the opposite change was made to the library, such that `toString` is read in the old version of the library but not in the new version, then the rule $t <: \circ$ (see Definition 4) would prevent the type regression, as desired.

7 Evaluation

To evaluate the type regression testing technique, we developed a tool, `NOREGRETS`.⁹ The implementation consists of 1800 lines of TypeScript code and 6400 lines of Scala code. The TypeScript part implements the instrumentation described in Section 5, using ES6 proxies. The Scala part fetches test suites for the npm packages from GitHub and post-processes the generated models to detect type regressions as described in Section 6.

Our primary hypothesis is that the type regression testing technique can help a developer decide if an update should be marked as either a major update or as a minor or patch update. We test this hypothesis by considering the tool as a *binary classifier* [28, 29], that is, a decision procedure that, given an input (in our case, a library update), returns one of two possible outcomes (*breaking* or *non-breaking*). Our secondary hypothesis is that the tool improves the *dont-break* approach for finding breaking changes, by amplifying the ability of the client test suites to reveal breaking changes, and by providing accurate and meaningful type regression reports. These hypotheses lead to the following research questions:

RQ1 How accurate is `NOREGRETS` in the classification of library updates as breaking or non-breaking?

RQ2 How does `NOREGRETS` compare with the *dont-break* approach? Specifically:

1. How many breaking changes does `NOREGRETS` find, and how many of those cannot be detected by the *dont-break* approach?
2. How many of the type regressions reported by `NOREGRETS` are spurious?
3. When a failure is detected, is it easy to locate the root cause?

Benchmarks. We randomly selected 12 among the most depended upon libraries from npm¹⁰ as benchmarks, listed in Table 1. Since the development and release process of those libraries is usually under scrutiny of many skilled developers, they can be considered high-quality libraries: their changelogs are usually accurate, minor updates rarely introduce breaking changes, and major updates are usually reserved for those situations where breaking changes have been introduced.

⁹ `node.js` type regression tester

¹⁰ <https://www.npmjs.com/browse/depended>

■ **Table 1** Node.js libraries used in the experimental evaluation.

Benchmark	LOC	Minor/Patch	Major	Client Test Suites	Model Size
<i>debug</i> 2.0.0	226	19	1	63	33
<i>async</i> 2.0.0	1682	5	0	64	3316
<i>lodash</i> 3.0.0	5225	16	1	42	4661
<i>moment</i> 2.0.0	1041	31	0	5	5
<i>express</i> 3.0.0	1011	95	1	4	54
<i>chalk</i> 1.0.0	169	4	0	93	105
<i>bluebird</i> 3.0.0	4827	29	0	16	503
<i>react</i> 15.0.0	41 685	11	1	5	31
<i>commander</i> 2.0.0	370	12	0	4	7
<i>request</i> 2.0.0	626	98	0	9	13
<i>body-parser</i> 1.0.0	89	55	0	3	6
<i>q</i> 1.0.0	1152	9	1	8	277

For our experiments, we picked a recent major version of each library, together with all minor and patch updates up to the next major release. The main focus of these experiments is on minor and patch updates, which are the ones where the developers do not expect breaking changes, but we also include a few major updates to check that NoREGRETS is able to classify those as breaking.

Table 1 contains additional details on the selected libraries: the name and the major version of the library, the number of lines of code in the major version of the library, the number of minor/patch and major updates of the library considered, the number of clients with test suites, and the size of the public API (counted as the number of non-`o` paths in the inferred model, averaged over the different library versions). All the data collected comes from a snapshot of the npm repository taken in April 2017. To simplify the experiments we only consider clients that use the Mocha testing framework, and to reduce noise we omit test suites that do not succeed consistently on major releases.

Our open-source implementation of NoREGRETS and all benchmarks and experimental data are available at <http://brics.dk/noregrets>.

RQ1 (accuracy as binary classifier)

Since the benchmarks selected in this experiment are high-quality libraries, we assume that most of the minor and patch updates of our benchmarks are not introducing breaking changes, and that most of the major updates are introducing breaking changes. In this way, we can evaluate our tool by checking that it correctly classifies major and non-major updates as breaking and non-breaking, respectively.

NoREGRETS reports type regressions for only 36 out of 384 minor or patch updates, and for 4 of the 5 major updates. A few of the type regressions detected at minor updates are actual breaking changes being introduced by mistake, for instance the one shown in the motivating example and all the ones discussed for RQ2.1 later in this section. Moreover, NoREGRETS is in fact correct also for the major update that is not being reported: a manual inspection confirms that the update of *debug* to version 3.0.0 does not introduce any type-related breaking changes apart from removing functions that were already deprecated and therefore not used by any available clients. Even if we disregard the fact that some of the minor updates are actually breaking and some major updates are non-breaking, NoREGRETS is able to give accurate suggestions to library developers: *In at least 90% of the cases, NoREGRETS is able to correctly classify a library update as either major or as minor or patch.*

■ **Table 2** Breaking changes found.

Benchmark	Changelog	Test Failure	Synthetic Client
<i>debug</i> 2.0.0	0	1	0
<i>async</i> 2.0.0	0	3	0
<i>lodash</i> 3.0.0	2	0	0
<i>moment</i> 2.0.0	2	0	0
<i>express</i> 3.0.0	0	0	18
Total		26	

RQ2 (comparison with the *dont-break* approach)

To answer the second research question, we manually inspected each type regression reported by NOREGRETS on minor and patch updates. To reduce the time spent, we focused on 5 benchmarks (*debug*, *async*, *lodash*, *moment*, and *express*).

RQ2.1. We consider a type regression on a path as introducing a breaking change whenever (i) the developers reference the change in the changelog, (ii) the breaking change can be witnessed by a test failure of one of the selected clients, or (iii) we can construct a synthetic client that crashes because of the change. If none of these conditions are satisfied, then the type regression is classified as a false positive. The breaking changes in the second category are the only ones that can be identified by the *dont-break* approach. The synthetic clients in the third category may not be representative of typical clients, but they nevertheless witness breaking changes. Also, as demonstrated by our motivating example and by Example 10, breaking changes often cause problems exactly because clients use libraries in ways that the library developers did not anticipate.

► **Example 10.** An example of a breaking change in the first category is the one introduced by *moment* 2.5.1 where the library started using the method `hasOwnProperty`. Unfortunately, the method is only available on non-host objects in older browsers. It took until version 2.8.2 for developers to realize this fact and fix the problem.¹¹

The main results of our inspection of the reported regressions are shown in Table 2. The Changelog column contains the number of breaking changes that are confirmed by the changelog, and the Test Failure and Synthetic Client columns show how many are witnessed by a failing preexisting test or a synthetic client, respectively. *Remarkably*, NOREGRETS is able to find 26 breaking changes in minor updates of high-quality libraries. It is also notable that this is accomplished with few client tests; for example, the two breaking changes in *moment* are detected using only 5 client test suites.

Moreover, *only 4 of the breaking changes that we have found could also be identified by a test failure*, demonstrating that our approach amplifies client tests compared to the *dont-break* approach. Going back to Example 10, note that detecting the breaking change by the *dont-break* approach would not just require a test that triggers the invocation of `hasOwnProperty` on non-host objects, but the test should also be run in a specific browser. Instead, NOREGRETS reports a type regression indicating that the object passed to *moment* should have a function `hasOwnProperty`.

¹¹ <https://github.com/moment/moment/pull/1874>

RQ2.2. On the 5 benchmarks, NOREGRETS reports a total of 168 type regressions across 167 library updates. By manually inspecting these 168 reports we find that 96 indicate actual breaking changes. (Some reports have the same root cause, which is how we arrive at a total of 26 breaking changes.) Thus, the number of false positives is acceptable: on average around one warning is reported per library update, and the majority of the warnings indicate actual breaking changes. Moreover, in the situations where multiple warnings are reported at a library update, we find that investigating the cause of one warning often quickly shows that other warnings have the same cause and therefore can be dismissed with little effort.

► **Example 11.** In *lodash* 3.10.0, the developers changed the behavior of the public method `isPlainObject` to use a different heuristic for recognizing so-called plain objects. Quoting from the implementation: “In most environments an object’s own properties are iterated before its inherited properties”. The method was changed accordingly to inspect all properties of the given object using a for-in loop, which causes 44 type regressions to be reported, one for each property of the objects being passed to `isPlainObject`. We classified this case as a false positive: the type regressions do not identify a breaking change since the property values are not used for anything but equality checks. We only had to inspect one library code location to understand that the other type regression reports had the same cause.

In our inspection of the regression reports, we proceeded hierarchically by the length of the paths involved in type regressions. By doing so, in our classification, we only needed to inspect a total of 36 type regressions out of 168, to identify the root cause of the breaking change they were referring to or discard them as false positives.

As already mentioned, 26 of our investigations resulted in the identification of an actual breaking change. To give some examples, the type regressions for 5 of these 26 breaking changes are listed in the first first 5 rows of Table 3. The *lodash* and *moment* examples have already been explained in Section 3 and in Example 10. The *debug* example, which shows the type regression for the breaking change mentioned in Section 1, and the *async* example both involve the special `throws` type. In the *async* example, the call to `require` returns an object in version 2.0.1 but throws an exception in version 2.1.0 because the module has been moved. Notice that none of type rules in Section 5.2 explicitly involve the `throws` type. Thereby, it is a type error if a function either starts throwing exceptions or stops throwing exceptions after a library update, which the *async* example motivates well; moving a public module clearly affects the public API. Likewise in the *debug* example, a spelling error results in an exception being thrown when the *debug* module is loaded, which breaks the public API. The *express* example is similar to the *moment* and *lodash* examples: the property `readable` of the function argument is not accessed in version 3.14.0 of *express*, but it is accessed in version 3.15.0, demonstrating that the type signature of the library function has changed.

NOREGRETS also reported 72 type regressions that we categorized as false positives. Using the same approach as when investigating the true positives, we found that these false positives had 10 separate causes. Many of these were due to technical limitations of NOREGRETS rather than of the type regression testing technique itself. ES6 introduces default exports of objects and functions, where the default exports are automatically read by Node.js when reading properties of the required object, but from the perspective of dynamic access paths it still looks like the properties are read from the default object, e.g., `require(foo).default.p` instead of just `require(foo).p`. In the 2.1.2 patch update of the *async* library, it started to use default exports resulting in 3 reports about breaking changes that are false positives. These are only false positives because a fallback mechanism is included to handle old installations that do not support default exports. Therefore, it is fair to assume that an inconsiderate developer, who did not include a fallback mechanism, may still have benefited from these warnings.

■ **Table 3** Type regression examples.

Library Update	Path	Type Regression
<i>debug</i> 2.3.3 → 2.4.0	<code>require(debug)()₁</code>	<code>throws ^ object ↯: function ^ object</code>
<i>async</i> 2.0.1 → 2.1.0	<code>require(async/asyncify)</code>	<code>throws ^ object ↯: function ^ object</code>
<i>lodash</i> 3.2.0 → 3.3.0	<code>require(lodash).merge(3)₄.toString</code>	<code>o ↯: undefined</code>
<i>moment</i> 3.5.0 → 3.5.1	<code>require(moment)(1)₁.hasOwnProperty</code>	<code>o ↯: function ^ object</code>
<i>express</i> 3.14.0 → 3.15.0	<code>require(express)()₀(1)₂.readable</code>	<code>o ↯: boolean</code>
<i>lodash</i> 3.10.1 → 4.0.0	<code>require(lodash).pluck</code>	<code>undefined ↯: function ^ object</code>
<i>express</i> 3.21.2 → 4.0.0	<code>require(express).mime</code>	<code>undefined ↯: object</code>
<i>lodash</i> 3.10.1 → 4.0.0	<code>require(lodash).forEach(2)₃()₃</code>	<code>undefined ↯: throws ^ object</code>

Another cause of false positives is that `hasOwnProperty` is not being instrumented, which is due to limitations of ES6 proxies.

RQ2.3. For each breaking change detected by NOREGRETS, the type regression report contains the involved dynamic access path p and associated types $\pi(p)$ and $\pi'(p)$. In contrast, when the *dont-break* approach detects a breaking change, it only provides the failing client test, with no information about the interactions between the client code and the library.

Based on our experience with the *dont-break* approach in the preliminary study (Section 2) and the NOREGRETS approach in the experiment for RQ2.1, we find that type regression reports greatly simplify the investigations of the breaking changes. The library developer does not need to understand what the client tests are doing, and can focus exclusively on the changes in the library’s codebase that have resulted in the changes of the public API. Since NOREGRETS additionally records the call-stack at the point when a new type observation is created, dynamic access paths can easily be correlated with actions performed deep down in the private code of the library. A typical example is the one discussed in Section 3 where the path `require(lodash).merge(3)₄.toString` shows that the coercion performed in the private `isIterateeCall` function is actually performed on the third argument of the `merge` function, and such information is not available if using the *dont-break* approach.

The last three rows of Table 3 show examples of type regressions found by NOREGRETS in major updates. The first two examples show that the `pluck` function was removed from *lodash* in version 4.0.0 and that the `mime` property was removed from *express* in version 4.0.0. The last example, involving the `forEach` function of *lodash*, is a little more subtle. The `forEach` function takes 3 parameters in version 3.10.1, a collection, a callback function, which is applied to each element in the collection, and an object that is used as the `this` object in the callback. However, in version 4.0.0, the ability to set the `this` object of the callback is removed from the `forEach` function. After the update, reading a property of `this` in the callback causes a type error since `this` is now `undefined`.

Discussion

The experimental evaluation is based on relatively few client test suites, which limits the fraction of the public APIs that are modeled and thereby reduces NOREGRETS’s ability to detect breaking changes. The libraries used in the experiments have thousands of clients, but our current implementation uses a fairly simple technique to locate clients and retrieve their test suites. In particular, it currently does not look for clients on GitHub but only uses npm. Also, tests are rarely published together with packages on npm, so NOREGRETS requires that

the `packages.json` file for each client contains a URL to a GitHub repository and that this repository contains a git tag matching the client version that is required. Many clients do not include tags in their repositories, so we chose to discard those clients. In principle, this issue could be alleviated by comparing the source code in all the repository commits with the source of the client published on npm, but we leave that to future work. Additionally, as mentioned we exclude test suites that do not succeed consistently on major releases. This is a well-known problem: In a recent study of 373 popular JavaScript applications, 41 of the packages had tests that failed or froze, and 3 had build or deployment issues [10]. This problem could in principle be mitigated by using a more fine-grained approach where `NOREGRETS` looks at test failures at the granularity of single tests rather than entire test suites. Furthermore, the ES6 proxy mechanism used by `NOREGRETS` sometimes interferes with tests causing them to fail, so we have to disregard those too to avoid noise. This is a known problem with opaque ES6 proxies, which has already been addressed by Thiemann et al. [17]. We could similarly solve this problem by modifying Node.js such that proxies becomes transparent, but this is again a technical limitation of our current implementation that could be alleviated with further implementation work. Still, the experimental results obtained with our current proof-of-concept implementation suffice to demonstrate the potential of the type regression testing idea.

Another opportunity for improvements is to investigate extensions of our notion of types that could arguably enable `NOREGRETS` to better fit specific programming constructs. For example, Andreasen et al. [3] show that parametric polymorphism and recursive types could be beneficial to type JavaScript functional programming constructs used in practice. Other possible extensions include representations of tuple types, polymorphic functions, and variadic parameters. Although these are theoretically interesting ideas, and could easily be implemented in `NOREGRETS`, in our evaluation we have not yet encountered concrete use-cases to justify the technical effort to introduce them.

8 Related Work

Studies of npm. Several experimental studies have investigated the npm repository [10, 19]. A study on JavaScript repositories showed that regression testing is a common practice, with an average of 78% of the packages having at least one test [10]. Two studies focus on the structure of the npm dependency network [30, 18]. In one of the studies, it is shown that the mean number of direct dependencies is 6, and that this number seems to be growing rapidly [30]. The same study also showed that the percentage of packages that are depended upon by other packages is only 27.5%, and a few popular packages are widely used by other packages. This should not be seen as a threat to the general applicability of our technique. If a package has no dependencies, then it matters little that the packages developer adheres to the semantic versioning principle. Another study has shown that the number of transitive dependencies is 10 times the number of direct dependencies and confirms that the number of dependencies of packages is growing exponentially, with a 60% increase in 2016 [18].

Studies of library updates and breaking changes. Our preliminary study is the first published study of the prevalence of breaking changes in the npm repository. So far, research on breaking changes has focused on other ecosystems, mainly Java. An experimental study by Derr et al. [9], involving 203 developers, analyzed the reasons behind many Android applications using outdated libraries versions. More than 50% of the participants indicated that one of the reasons was to “prevent incompatibilities”. The authors developed a tool to compute the difference between the public API of two Java library versions, which they used

to show that as many as 39% of minor and patch updates should have been flagged as major, which justifies the skepticism about the guarantees of semantic versioning expressed by the 203 developers.

The study from Raemakers et al. [26] addressed the use and misuse of semantic versioning in the Maven repository for Java packages. They conclude that “one third of all releases introduces at least one breaking change, and that this figure is the same for minor and major releases, indicating that version numbers do not provide developers with information in stability of interfaces”, showing that breaking changes are prevalent in Maven repositories. A similar study by Jezek et al. [16] on 109 Java open-source libraries discovered that every library introduces at least one breaking change of the public API in non-major updates.

Other studies are concerning the relation between library updates and breaking changes for JavaScript libraries. Mirhosseini and Parnin [22] showed that breaking changes, understanding the implications of changes, and migration effort are among the top concerns of JavaScript developers. A small user study among npm package maintainers showed that package updates are mostly coordinated by personal communications between developers [7]. A follow-up study, comparing 8 npm library developers to Eclipse and R/CRAN developers, showed that “npm developers were more willing than developers of other platforms to perform breaking changes in the name of progress” [8]. A study of the prevalence of client side vulnerabilities in web applications also showed that, like in the Maven system, many applications are using outdated libraries [20]. Zerouali et al. also found that many dependencies in the npm system are outdated due to too strict version constraints, and conclude that developers are reluctant to update dependencies since they want to avoid incompatible changes [31]. Wittern et al. [30] showed that 29% of all package.json version constraints specify a fixed version, while 68% of the constraints allow either all minor and patch updates or just all patch updates. The remaining 3% are free ranging constraints that also allow major updates.

Type inference for dynamic languages. Dynamic inference of types for dynamic languages is a widely studied topic [2, 3, 27, 1, 24]. However, this paper is the first one to also distinguish between private and public parts of a library’s API. The use of runtime traces to learn types has already been exploited for Ruby [2], JavaScript [3, 27], and Dart [1]. One notable difference with our approach is that we do not ascribe types to syntactical elements of the programs, but instead to our notion of dynamic access paths. TypeDevil [24] uses a dynamic analysis to gather runtime type information where types are either primitive or records of types. Inconsistencies of the observed types are reported as potential bugs. Other forms of dynamic analysis for JavaScript are discussed in a recent survey [4].

Detection of breaking changes. The only other tool that also aims at detecting breaking changes in npm package updates is the *dont-break* tool. Unfortunately, we were not able to make *dont-break* work properly, but we applied the same methodology in the preliminary study as discussed in Sections 2 and 7.

Greenkeeper is a service that helps packages maintainers avoid introducing dependency updates that contain breaking changes.¹² Instead of using range-based dependency constraints that allow all minor and patch updates, packages that use Greenkeeper will fix each dependency to a specific version. Whenever a new version of a dependency is available, Greenkeeper will run the tests of the package with the updated dependency to verify that the update did not break anything.

¹²<https://greenkeeper.io/>

Java's binary compatibility conditions [13, Chapter 13] and tools like JAPICC [23] make it possible to automatically detect type-related breaking changes in Java libraries. A disciplined set of guidelines for upgrading library releases have also been developed within the IBM's System Object Model to guarantee binary compatibility [12].

9 Conclusion

We have shown that breaking changes do occur in minor and patch updates of npm packages and that the majority of the breaking changes are type-related. Furthermore, we have designed a novel technique called type regression testing that detects type-related breaking changes across library versions, by leveraging the test suites of the library's clients. Type regression testing uses an instrumented JavaScript interpreter to build a model of a library's API through dynamic observations of how the client tests interact with the library. The models use the notion of dynamic access paths to give types to the individual components of the library's API. Specific differences in the model across two library versions are identified as type regressions, indicating that a breaking change likely has occurred.

We have implemented type regression testing in the tool `NOREGRETS`. Our evaluation shows that `NOREGRETS` is capable of detecting 26 breaking changes in 167 minor and patch updates of 5 high quality npm packages, and most of those breaking changes could not have been detected by existing techniques. We also find that `NOREGRETS` reports only a small number of false positives, and that the reported type regressions make it easy for the developer to determine the causes of the breaking changes. Furthermore, `NOREGRETS` correctly classifies at least 90% of the updates as either major or as minor or patch.

References

- 1 Christoffer Quist Adamsen, Gianluca Mezzetti, and Anders Møller. Analyzing test completeness for dynamic languages. In *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, pages 142–153, 2016. doi:10.1145/2931037.2931059.
- 2 Jong-hoon (David) An, Avik Chaudhuri, Jeffrey S. Foster, and Michael Hicks. Dynamic inference of static types for Ruby. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 459–472, 2011. doi:10.1145/1926385.1926437.
- 3 Esben Andreasen, Colin S. Gordon, Satish Chandra, Manu Sridharan, Frank Tip, and Koushik Sen. Trace typing: An approach for evaluating retrofitted type systems. In *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy*, pages 1:1–1:26, 2016. doi:10.4230/LIPIcs.ECOOP.2016.1.
- 4 Esben Andreasen, Anders Møller, Liang Gong, Michael Pradel, Marija Selakovic, Koushik Sen, and Cristian-Alexandru Staicu. A survey of dynamic analysis and test generation for JavaScript. *ACM Computing Surveys*, 50(5):66:1–66:36, 2017.
- 5 Thomas H. Austin, Tim Disney, Alan Jeffrey, and Cormac Flanagan. Dynamic information flow analysis for featherweight JavaScript. Technical Report UCSC-SOE-11-19, UC Santa Cruz, 2011.
- 6 Martin Bodin, Arthur Charguéraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudziuniene, Alan Schmitt, and Gareth Smith. A trusted mechanised JavaScript specification. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 87–100. ACM, 2014.

- 7 Christopher Bogart, Christian Kästner, and James D. Herbsleb. When it breaks, it breaks: How ecosystem developers reason about the stability of dependencies. In *30th IEEE/ACM International Conference on Automated Software Engineering Workshops, ASE Workshops 2015, Lincoln, NE, USA, November 9-13, 2015*, pages 86–89, 2015. doi:10.1109/ASEW.2015.21.
- 8 Christopher Bogart, Christian Kästner, James D. Herbsleb, and Ferdian Thung. How to break an API: cost negotiation and community values in three software ecosystems. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, pages 109–120, 2016. doi:10.1145/2950290.2950325.
- 9 Erik Derr, Sven Bugiel, Sascha Fahl, Yasemin Acar, and Michael Backes. Keep me updated: An empirical study of third-party library updatability on Android. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 2187–2200, 2017.
- 10 Amin Milani Fard and Ali Mesbah. JavaScript: The (un)covered parts. In *2017 IEEE International Conference on Software Testing, Verification and Validation, ICST 2017, Tokyo, Japan, March 13-17, 2017*, pages 230–240. IEEE Computer Society, 2017.
- 11 Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI), Albuquerque, New Mexico, USA, June 23-25, 1993*, pages 237–247, 1993. doi:10.1145/155090.155113.
- 12 Ira R. Forman, Michael H. Conner, Scott Danforth, and Larry K. Raper. Release-to-release binary compatibility in SOM. In *OOPSLA '95, Proceedings of the Tenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications, Austin, Texas, USA, October 15-19, 1995*, pages 426–438. ACM, 1995.
- 13 James Gosling, Bill Joy, Guy L. Steele, Gilad Bracha, and Alex Buckley. *The Java Language Specification, Java SE 8 Edition*. Addison-Wesley Professional, 1st edition, 2014.
- 14 Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. The essence of JavaScript. In *ECOOP 2010 - Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings*, pages 126–150, 2010.
- 15 Atsushi Igarashi and Hideshi Nagira. Union types for object-oriented programming. *Journal of Object Technology*, 6(2):47–68, 2007. OOPS Track at the 21st ACM Symposium on Applied Computing, SAC 2006. doi:10.5381/jot.2007.6.2.a3.
- 16 Kamil Jezek, Jens Dietrich, and Premek Brada. How Java APIs break - an empirical study. *Information & Software Technology*, 65:129–146, 2015. doi:10.1016/j.infsof.2015.02.014.
- 17 Matthias Keil, Sankha Narayan Guria, Andreas Schlegel, Manuel Geffken, and Peter Thiemann. Transparent object proxies in JavaScript. In *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic*, pages 149–173, 2015. doi:10.4230/LIPIcs.ECOOP.2015.149.
- 18 Riivo Kikas, Georgios Gousios, Marlon Dumas, and Dietmar Pfahl. Structure and evolution of package dependency networks. In *Proceedings of the 14th International Conference on Mining Software Repositories, MSR 2017, Buenos Aires, Argentina, May 20-28, 2017*, pages 102–112. IEEE Computer Society, 2017.
- 19 Raula Gaikovina Kula, Ali Ouni, Daniel M. Germán, and Katsuro Inoue. On the impact of micro-packages: An empirical study of the npm JavaScript ecosystem. *CoRR*, abs/1709.04638, 2017. arXiv:1709.04638.
- 20 Tobias Lauinger, Abdelberi Chaabane, Sajjad Arshad, William Robertson, Christo Wilson, and Engin Kirda. Thou shalt not depend on me: Analysing the use of outdated JavaScript

- libraries on the web. *Proceedings of Network and Distributed System Security Symposium (NDSS)*, 2017.
- 21 Barbara Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, 1994. doi:10.1145/197320.197383.
 - 22 Samim Mirhosseini and Chris Parnin. Can automated pull requests encourage software developers to upgrade out-of-date dependencies? In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, pages 84–94, 2017. doi:10.1109/ASE.2017.8115621.
 - 23 Andrey V. Ponomarenko and Vladimir V. Rubanov. Backward compatibility of software interfaces: Steps towards automatic verification. *Programming and Computer Software*, 38(5):257–267, 2012.
 - 24 Michael Pradel, Parker Schuh, and Koushik Sen. TypeDevil: Dynamic type inconsistency analysis for JavaScript. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, pages 314–324, 2015. doi:10.1109/ICSE.2015.51.
 - 25 Tom Preston-Werner. Semantic versioning 2.0.0. <http://semver.org/>.
 - 26 Steven Raemaekers, Arie Van Deursen, and Joost Visser. Semantic versioning versus breaking changes: A study of the Maven repository. *Proceedings - 2014 14th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2014*, pages 215–224, 2014. doi:10.1109/SCAM.2014.30.
 - 27 Claudiu Saftoiu, Arjun Guha, and Shriram Krishnamurthi. Runtime type-discovery for JavaScript. Technical Report Brown University CS-10-05, 2010.
 - 28 Marina Sokolova and Guy Lapalme. A systematic analysis of performance measures for classification tasks. *Inf. Process. Manage.*, 45(4):427–437, 2009.
 - 29 Sofia Visa, Brian Ramsay, Anca L. Ralescu, and Esther van der Knaap. Confusion matrix-based feature selection. In *Proceedings of the 22nd Midwest Artificial Intelligence and Cognitive Science Conference 2011, Cincinnati, Ohio, USA, April 16-17, 2011*, pages 120–127, 2011. URL: <http://ceur-ws.org/Vol-710/paper37.pdf>.
 - 30 Erik Wittern, Philippe Suter, and Shriram Rajagopalan. A look at the dynamics of the JavaScript package ecosystem. In *Proceedings of the 13th International Conference on Mining Software Repositories, MSR 2016, Austin, TX, USA, May 14-22, 2016*, pages 351–361, 2016. doi:10.1145/2901739.2901743.
 - 31 Ahmed Zerouali, Eleni Constantinou, Tom Mens, Gregorio Robles, and Jesús M. González-Barahona. An empirical analysis of technical lag in npm package dependencies. In *New Opportunities for Software Reuse - 17th International Conference, ICSR 2018, Madrid, Spain, May 21-23, 2018, Proceedings*, volume 10826 of *Lecture Notes in Computer Science*, pages 95–110. Springer, 2018.

Targeted Test Generation for Actor Systems

Sihan Li

Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, USA
sihanli2@illinois.edu

Farah Hariri

Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, USA
hariri2@illinois.edu

Gul Agha

Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, USA
agha@illinois.edu

Abstract

This paper addresses the problem of targeted test generation for actor systems. Specifically, we propose a method to support generation of system-level tests to cover a given code location in an actor system. The test generation method consists of two phases. First, static analysis is used to construct an abstraction of an entire actor system in terms of a *message flow graph* (MFG). An MFG captures potential actor interactions that are defined in a program. Second, a *backwards symbolic execution* (BSE) from a target location to an “entry point” of the actor system is performed. BSE uses the MFG constructed in the first phase of our targeted test generation method to guide execution across actors. Because concurrency leads to a huge search space which can potentially be explored through BSE, we prune the search space by using two heuristics combined with a feedback-directed technique. We implement our method in TAP, a tool for Java *Akka* programs, and evaluate TAP on the *Savina* benchmarks as well as four open source projects. Our evaluation shows that the TAP achieves a relatively high target coverage (78% on 1,000 targets) and detects six previously unreported bugs in the subjects.

2012 ACM Subject Classification Software and its engineering → Software testing and debugging

Keywords and phrases actors, symbolic execution, test generation, static analysis

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2018.8

Acknowledgements This work is supported in part by the National Science Foundation under grants NSF CCF 14-38982 and NSF CCF 16-17401. The authors would like to thank Krishna Kura for his help on transforming the benchmarks to Java *Akka* and thank Nadeem Jamali and the anonymous reviewers for their helpful comments and suggestions.

1 Introduction

We address the problem of targeted test generation for actor systems. Recall that an actor is an autonomous, concurrent agent which communicates with other actors using asynchronous messages. Asynchronous message-passing and state encapsulation (isolation) in actors make it easier to understand the message flow and facilitate scalability. State encapsulation prevents low-level data races and atomicity violations. Asynchronous message-passing avoids syntactic deadlocks [6, 7]. As a result, actor languages and frameworks—such as Erlang [9], Salsa [40], Scala/Java *Akka* [4, 2], and Orleans [3]—have gained in popularity, and have been used for scalable applications (for example, see [1, 3]).



© Sihan Li, Farah Hariri, and Gul Agha;

licensed under Creative Commons License CC-BY

32nd European Conference on Object-Oriented Programming (ECOOP 2018).

Editor: Todd Millstein; Article No. 8; pp. 8:1–8:31

Leibniz International Proceedings in Informatics



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

A goal of testing programs is to detect violations of desired safety properties. Some safety properties such as “no dangling links” or “division by zero” are implicit. Others are explicitly stated in the form of assertions. Violations of safety properties happen if particular lines of the code can be reached with problematic data. Because concurrency leads to nondeterminism, figuring out if particular lines of the code can be reached is challenging. By taking advantage of the actor semantics, more effective testing tools may be developed. One approach [33, 22] is to combine *concolic testing* [34] with *partial order reduction* based on a macro-step actor semantics [8]. Unfortunately, given the very large number of potential message schedules in an actor system, concolic testing is sometimes ineffective in determining if a particular code location can be reached.

An alternate approach is to use a *targeted test generation* technique to try to generate tests that cover specific code locations.¹ Targeted test generation has the advantage that one does not explore paths leading to code locations that obviously cannot have problems. Previous research has developed techniques and tools based on symbolic execution for targeted test generation for sequential programs (e.g., [24, 18, 16, 25, 10, 15, 29, 13]).

In this paper, we propose a method for generating targeted tests for actor systems based on *backward symbolic execution* (BSE). The tests we generate are system-level test: they exercise a group of interacting actors rather than only an isolated actor. The goal is to find if a particular line can be reached through sending messages to the *entry point* of an actor system, where an entry point is a message handler of an actor which interacts with the external environment. In actor terminology, such actors are called *receptionists*. Each test consists not only of the messages received by each actor but also the order in which these messages are received. We start a BSE from the target code and explore only those paths that are relevant to reaching that target; the exploration continues until a feasible path to an entry point of the actor system is found.

In sequential programs, a call graph is used to guide the inter-procedural BSE [15, 29, 13]. In the actor context, we propose to use an abstraction of an actor system called *message flow graph* (MFG). An MFG captures interactions between actors and is useful to guide inter-actor BSE. We develop a sound whole-system static analysis to construct MFGs for actor systems.

One challenge in static MFG construction and BSE for actor systems is to handle actor operations such as message send/receive and actor creation. Even when an actor framework is written in a language like Java, analyses that treat these actor operations as normal methods would not work: if the actor semantics is ignored, BSE will explore the library methods that are used to implement an actor runtime. Because a library that implements an actor runtime contains complex multi-threading and networking code, symbolic execution would become infeasible (cf. [22]). In addition, a static analysis would not be able to establish connections between actors without understanding the meaning of these library methods. To solve this problem, we define formal semantic models of actor operations in both MFG analysis and BSE, and replace actual implementations of actor operations with the semantic models. Assuming that the actor library has been correctly implemented, we prevent our analysis from exploring the underlying library. This makes our analysis more efficient and thus scalable.

In general, it is computationally intractable to consider every possible message arrival schedule even if we explore only paths that are relevant to a single target. To efficiently navigate the search space, we use a depth-first search strategy combined with two heuristics

¹ Targeted test generation is sometimes called *directed* or *guided* test generation in the literature.

and a feedback-directed search technique. The depth-first strategy attempts to reach the entry point of the actor system as soon as possible. The two heuristics are as follows:

1. Each message handler is executed atomically so that search space is reduced due to the lack of the interleaved execution of message handlers. This heuristic applies the macro-step semantics in the Actor model [8], which follows from the fact that messages to a given actor are processed one at a time and that actors do not share state.
2. Low weights are assigned to transitions in BSE that introduce more actors to be explored, in order to avoid unnecessary explorations. The heuristic is based on the conjecture that most concurrency bugs may be triggered by considering interactions of a small number of actors. We do not have direct evidence of this conjecture. However, there is a previous finding that most concurrency bugs in multi-threaded programs may be triggered using only two threads [23].

As constraints are collected and solved, some paths turn out to be infeasible. In this case, we deduce an *unsatisfiable core*—a subset of the constraint clauses whose conjunction is unsatisfiable. Our feedback-directed technique uses these unsatisfiable cores to effectively drive BSE towards a feasible path. The technique is particularly useful in cases where BSE frequently hits infeasible paths.

We have implemented our method in a tool called TAP for the Java *Akka* framework [2], a popular library enabling actor-style programming in Java. However, our method can be applied to other actor frameworks or languages. We evaluate TAP on *Savina* [21], a set of 30 third-party actor benchmarks, as well as on four open source actor projects from GitHub. The evaluation results show that TAP is effective in covering targets, achieving 78% target coverage on a total of 1,000 targets. The heuristics and the feedback-directed technique together substantially improve the target coverage over random search. In addition, TAP detects six previously unreported bugs in the subjects, five of which are crash bugs caused by out-of-order message delivery.

The paper makes the following contributions:

- **The MFG concept and construction:** We introduce the MFG abstraction for actor systems and develop a sound static analysis to construct it.
- **Modeling of actor operations in BSE:** We formally define the full semantics of actor operations in BSE for actor systems.
- **Efficient path exploration:** We propose two search heuristics and a feedback-directed technique to efficiently navigate the generally huge search space in BSE of actor systems.
- **Implementation and evaluation:** We implement our method in TAP for Java *Akka*, and conduct evaluations on benchmarks and real-world projects that demonstrate TAP's effectiveness in target coverage and bug detection.

2 Background

We provide background on the Actor model and the Java *Akka* framework. We also describe the targeted test generation problem for actor systems in terms of the inputs and outputs.

2.1 The Actor model

In the Actor model [19, 6, 8], an actor is an agent of computation; it performs computations as a response to a message. An actor is characterized by an actor name, a local state, and behaviors. The actor name serves as the address of the actor in the system; it can be passed around to other actors so that they may send messages to it. The local state of an actor is

8:4 Targeted Test Generation for Actor Systems

```
1 public class Main {
2     public static void main(String[] args) {
3         ActorSystem system = ActorSystem.create("Banking");
4         ActorRef serverActor = system.actorOf(Server.props());
5         ActorRef clientActor = system.actorOf(Client.props(serverActor));
6     }
7 }
8 public class Client extends UntypedActor {
9     private double balance = 100;
10    private ActorRef server;
11    @Override
12    public void onReceive(Object message) {
13        if (message instanceof WithdrawMessage) {
14            double amount = ((WithdrawMessage) message).amount;
15            if (balance >= amount) {
16                balance -= amount;
17                server.tell(message);
18            }
19        } else if (message instanceof DepositMessage) {
20            double amount = ((DepositMessage) message).amount;
21            balance += amount;
22            server.tell(message);
23        }
24    }
25 }
26 public class Server extends UntypedActor {
27     private double balance = 100;
28     @Override
29     public void onReceive(Object message) {
30         if (message instanceof WithdrawMessage) {
31             double amount = ((WithdrawMessage) message).amount;
32             assert(balance >= amount);
33             balance -= amount;
34         } else if (message instanceof DepositMessage) {
35             double amount = ((DepositMessage) message).amount;
36             balance += amount;
37         }
38     }
39 }
```

■ **Figure 1** A simplified Bank Account example.

encapsulated within the actor – no external entity can change it directly. The only way to change the local state of an actor is to send it a message that triggers this actor to change its own state. Upon receiving a message, an actor can have the following three behaviors: (1) performing local computations (updating its local state), (2) sending messages to actors, or (3) creating new actors. Communication between actors is through *asynchronous* message passing – the sender does not block its computation waiting on the recipient to process the message, nor does it assume the order in which the recipient processes its incoming messages. Messages are immutable and processed by the recipient one at a time without interleaving. An actor system contains a group of actors. The subset of actors that can communicate with the external environment are called *receptionists*, and the other actors in the system are called internal actors.

2.2 Actors in Akka

Akka is a set of libraries for developing distributed and scalable systems on the Java Virtual Machine. It can be used in both Scala and Java. The core of *Akka* is the `akka-actor` library, which is an implementation of the Actor model. Figure 1 shows a simplified Bank Account example written using Java *Akka*. We use this example to illustrate important concepts of the Actor model in the context of *Akka*.

Actor Creation. To create actors, we need to first create the enclosing actor system (Line 3 in Figure 1), a container in which the actors run. Then we create actors that live in the system via the method `actorOf`. The example creates two actors: a client and a server (Lines 4-5). The `actorOf` method takes as input a configuration object (`props`) that specifies the options for creating an actor such as its type and arguments to its constructor, and returns an `ActorRef` object, which represent the address of the actor in the system. The `ActorRef` corresponds to the concept, actor name in the Actor model. Following the naming convention in *Akka*, we will use the terms actor reference and actor name interchangeably in this paper. Other actors can send a message to this `ActorRef`, and the actor identified by this `ActorRef` will receive this message. Note that other actors *cannot* directly access the local state of this actor (e.g., access fields, call instance methods) through the `ActorRef`.

Acquaintance Relations. Actor A knows of actor B if A has access to the actor reference (`ActorRef`) of B. At Line 5 in our example, we create a `clientActor` and pass it the `ActorRef` of the `serverActor` (now the client knows of the server and can send messages to it). The actor reference can also be sent as a message to inform other actors. Another type of acquaintance between actors is via receiving messages: when an actor receives a message, it can access the actor reference of the sender through the `getSender()` method. An actor can also get its own actor reference through the `getSelf()` method.

Sending and Processing Messages. Every actor must implement a message handler, the `onReceive` method. The `onReceive` method takes as input a message object, and is invoked upon receiving a message. Typically, different types of messages trigger different behaviors in the actor. For example, the `onReceive` of the `Client` actor (Lines 13-23) behaves differently on the `WithdrawMessage` and the `DepositMessage`. Messages are sent via calling the `tell` method on an `ActorRef` object (e.g., Line 17).

2.3 Problem Description

Actors model an *open system* – a system that may interact with its external environment. In order to preserve locality properties of actors, such interaction is through messages received by receptionist actors in the system and messages sent to external actors by actors in the system. Thus the entry points of the system are message handlers of receptionists. Examples of open systems in the real-world include Twitter, LinkedIn, Facebook Chat, and Halo 4, all of which have been implemented using actors.

The input to our problem includes: (1) the code under test, (2) a target code location, (3) a user defined set of receptionists of the system, and (4) a start configuration defining the initial acquaintance between actors. The output (if found) is a test case that covers the target. Such a test consists of messages sent to relevant actors as well as their arrival orders on each of these actors.

In our Bank Account example, the code under test is the `Client` and the `Server` actor classes; the receptionist is the `Client` actor as the client is the interface of the system for user interactions. The main method sets up the initial acquaintance that the client knows of the server. Suppose our target is the negation of the assertion at Line 32. One possible output test case that covers the target is as follows. The client receives a deposit message with the amount 50 and a withdraw message with the amount 120, in that order. Since the deposit message is received before the withdraw message, the condition at Line 15 is evaluated to true, and the client forwards both messages to the server. However, on the server side, the withdraw message somehow arrives before the deposit message, causing the assertion violation. This test case specifies the messages received by the client and the server

$$\begin{aligned}
\text{Class} &::= \text{class } C \text{ extends } C' \{ \overrightarrow{C''} f; K \overrightarrow{M} \} \\
\text{ActorClass} &::= \text{class } C \text{ extends } C' \{ \overrightarrow{C''} f; \mathbf{K} \mathbf{R} \overrightarrow{M} \} \\
K \in \text{Ctor} &::= C(\overrightarrow{C'} f) \{ \text{super}(\overrightarrow{f'}); \overrightarrow{\text{this.f''}} = \overrightarrow{f'''}; \} \\
\mathbf{R} \in \text{Receive} &::= \text{void onReceive}(C \mathbf{v}) \{ \overrightarrow{C'} \mathbf{v}'; \overrightarrow{\mathbf{s}} \} \\
M \in \text{Method} &::= C m(\overrightarrow{C'} v) \{ \overrightarrow{C'} v'; \overrightarrow{\mathbf{s}} \} \\
s \in \text{Stmt} &::= v = e;^\ell \mid \text{return } v;^\ell \mid \text{if } (e) \overrightarrow{\mathbf{s}} \text{ else } \overrightarrow{\mathbf{s}'};^\ell \mid \mathbf{v.send}(\mathbf{v}');^\ell \\
e \in \text{Expr} &::= v \mid (C) v' \mid v.f \mid v.m(\overrightarrow{v'}) \mid \text{new } C(\overrightarrow{v}) \mid v \text{ op } v' \mid \text{aref} \\
\text{aref} \in \text{ARef} &::= \text{create}(C.\text{class}, \overrightarrow{\mathbf{v}}) \mid \text{self} \mid \text{sender} \\
v \in \text{Var} &\text{ is a set of variable names} \\
f \in \text{FieldName} &\text{ is a set of field names} \\
C \in \text{ClassName} &\text{ is a set of class names} \\
m \in \text{MethName} &\text{ is a set of method names} \\
\ell \in \text{Lab} &\text{ is a set of labels} \\
\text{op} \in \{ +, -, *, /, <, >, ==, !=, \dots, \text{instanceof} \}
\end{aligned}$$

■ **Figure 2** An actor language extending Featherweight Java.

as well as the message receiving orders on both actors. For illustration purposes, we do not assume the first-in-first-out (FIFO) message delivery between a pair of actors in this example. Given FIFO message delivery, the two messages could be routed through different actors, still creating nondeterminism in the arrival order at the server.

Note that we must specify the receptionists of an actor system in our problem settings. This requirement enforces system-level testing because internal actors can only be tested through receptionists. In our example, to cover the target we have to send messages to the client in order to trigger messages sent to the server. If all actors were potential receptionists, then every actor may receive messages directly from the external environment. In this case, each actor may be tested individually with all possible message sequences and no interaction between actors need be considered. The benefit of considering external messages only to designated actors is that it constrains the generated tests to those which would realistically occur in an actor system. While this means that system-level testing is required, it eliminates consideration of tests based on arbitrary messages to individual actors that would never be sent in a realistic system.

3 Actor Language

To formally describe our method, we define a simplified actor language by extending Featherweight Java [20] and adding actor constructs to it. We choose the Featherweight Java language for its simplicity and for the fact that our tool targets Java *Akka*. The formalism in this paper largely follows the conventions in previous work [20, 35, 26]. The actor constructs in our language resemble the counterparts in Java *Akka*. Although there have been formalizations of actor languages [8, 32], our formalization of the language is closely coupled with our analysis, and includes more details such as data store and context, which are required to specify our analysis.

$$\begin{aligned}
\alpha &\in \text{ActorMap} = \text{ActorRef} \rightarrow \text{ActorState} \\
\text{msg} &\in \text{Message} = \text{ActorRef} \times \text{ActorRef} \times \text{Obj} \\
r &\in \text{ActorRef} \subset \text{Obj} \\
\varsigma &\in \text{ActorState} = \text{Stmt} \times \text{Stack} \times \text{Store} \times \text{CallStack} \times \text{Context} \\
st &\in \text{Stack} = (\text{Var} \rightarrow \text{Addr})^* \\
\sigma &\in \text{Store} = \text{Addr} \rightarrow \text{Obj} \\
o &\in \text{Obj} = \text{HContext} \times (\text{FieldName} \rightarrow \text{Addr}) \\
cs &\in \text{CallStack} = (\text{Stmt} \times \text{Context} \times \text{Addr})^* \\
a &\in \text{Addr} = (\text{Var} \times \text{Context}) \cup (\text{FieldName} \times \text{HContext}) \\
c &\in \text{Context} \text{ is an infinite set of regular contexts} \\
hc &\in \text{HContext} \text{ is an infinite set of heap contexts}
\end{aligned}$$

■ **Figure 3** Domains of actor maps and messages.

3.1 Syntax

Figure 2 describes the grammar of a simplified actor language. The language is in A-Normal form, where computations are syntactically sequentialized. For example, the statement $v = o.m(o.f)$ is transformed to two statements $v1 = o.f$; $v2 = o.m(v1)$ in A-Normal form. Such transformation brings our language closer to an intermediate language for simpler semantics definitions. Most of the notations in Featherweight Java are intuitive. We give a quick reminder of the less obvious conventions. A class declaration consists of a list of fields (we use an arrow to represent a list), a single constructor, and a list of methods. The constructor takes as input a list of arguments and assigns each argument to the corresponding field. Each statement in the language is assigned a distinct label. We augment Featherweight Java with *binary* expressions and *if* statements, which are later needed in the formalization of the BSE semantics. We omit the *loop* statement because loops are bounded and unrolled into *if* statements in our analysis. Such unrolling trades completeness for tractability and is standard practice in testing.

We now introduce actor constructs (highlighted in bold). Each actor class declaration must include exactly one **onReceive** method. This method takes a single input (message) and returns **void**. An actor creation operation **create**(*A.class*, \vec{v}) takes as input the class of the actor to be created *C*, followed by a list of arguments to the constructor of *C*, and returns the actor reference of the created actor. A message send operation $v.\text{send}(v')$;^ℓ sends the message v' to the actor reference v of the recipient actor.

3.2 Concrete Semantics

An instantaneous snapshot of an actor systems is called a *configuration*.² The semantics of our language is defined by a transition relation on configurations. A configuration is a tuple,

$$\langle \alpha \mid \mu \rangle$$

² Recall that actors are asynchronous: there is no unique global time. Thus an actor snapshot is with respect to some frame of reference, i.e., a causally consistent linearization of a partial order.

Actor Creation

$$\begin{aligned}
\langle \alpha \bullet r \mapsto (\llbracket v = \text{create}(C.\text{class}, \vec{v}'); \ell \rrbracket, st, \sigma, _) \mid _ \rangle &\Rightarrow_C \\
\langle \alpha \bullet r \mapsto (\text{succ}(\ell), st, \sigma', _) \bullet r' \mapsto \varsigma \mid _ \rangle, &\text{ where} \\
r' \text{ is fresh} \quad \sigma' = \sigma + [st(v) \mapsto r'] \quad o'_i = \sigma(st(v'_i)) \quad \varsigma = (\text{nil}, [], \sigma'', [], \text{nil}) \\
\vec{f} = \mathcal{F}(C) \quad a_i = (f_i, hc) \quad o = (hc, [f_i \mapsto a_i]) \quad \sigma'' = [a_{this} \mapsto o, a_i \mapsto o'_i, a_{self} \mapsto r']
\end{aligned}$$

Message Sending

$$\begin{aligned}
\langle \alpha \bullet r \mapsto (\llbracket v.\text{send}(v'); \ell \rrbracket, st, \sigma, _) \mid \mu \rangle &\Rightarrow_C \\
\langle \alpha \bullet r \mapsto (\text{succ}(\ell), st, \sigma, _) \mid \mu \bullet (r, \sigma(st(v)), \sigma(st(v'))) \rangle
\end{aligned}$$

Message Receiving

$$\begin{aligned}
\langle \alpha \bullet r \mapsto (\text{nil}, st, \sigma, cs, c) \mid \mu \bullet (r', r, o) \rangle &\Rightarrow_C \\
\langle \alpha \bullet r \mapsto (s, st', \sigma', cs', c') \mid \mu \rangle, &\text{ where} \\
c' \text{ is fresh} \quad o_0 = \sigma(a_{this}) \quad \llbracket \text{void onReceive}(C v) \{ \overrightarrow{C' v'}; \vec{s}' \} \rrbracket &= \text{rec}(cls(o_0)) \\
s = \text{car}(\vec{s}') \quad a = (v, c') \quad a'_i = (v'_i, c') \quad st' = \text{cons}([v \mapsto a, v'_i \mapsto a'_i], st) \\
cs' = \text{cons}((\text{nil}, c, \text{nil}), cs) \quad \sigma' = \sigma + [a \mapsto o, a_{sender} \mapsto r']
\end{aligned}$$

Self Reference

$$\langle \alpha \bullet r \mapsto (\llbracket v = \text{self}; \ell \rrbracket, st, \sigma, _) \mid _ \rangle \Rightarrow_C \langle \alpha \bullet r \mapsto (\text{succ}(\ell), st, \sigma + [st(v) \mapsto r], _) \mid _ \rangle$$

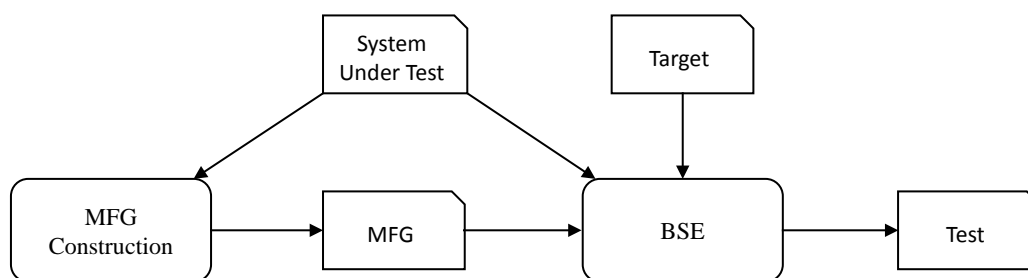
Sender Reference

$$\begin{aligned}
\langle \alpha \bullet r \mapsto (\llbracket v = \text{sender}; \ell \rrbracket, st, \sigma, _) \mid _ \rangle &\Rightarrow_C \\
\langle \alpha \bullet r \mapsto (\text{succ}(\ell), st, \sigma + [st(v) \mapsto \sigma(a_{sender})], _) \mid _ \rangle
\end{aligned}$$

■ **Figure 4** Concrete semantics for actor operations of the simplified actor language.

where α is an actor map that maps a finite set of actor references to actor states, and μ is a finite multi-set of pending messages. It is important to note that by modeling the pending messages as a multi-set, the order in which messages are sent is not preserved. As a result, our language semantics does not guarantee the FIFO message delivery between a pair of actors. We choose not to assume the FIFO message delivery in both the concrete language semantics and the BSE semantics in Section 5.1, because the FIFO semantics is not primitive in the Actor model [19, 6, 8]. However, one can easily accommodate the FIFO semantics in our models by replacing the multi-set with a data structure that preserves the message sending orders (e.g., a set of lists representing a sequence of messages, one list for each pair of a sender and a receiver). Since most real-world actor languages and frameworks guarantee the FIFO message delivery, we do implement the FIFO semantics in our tool.

The domains in a configuration are described in Figure 3. A message is a tuple consisting of the actor reference of the sender, the actor reference of the recipient, and the message content. An actor reference is an object that stores the location information of an actor. An actor state ς consists of a statement under execution, a data stack to store local variables, a data store of points-to relations, a call stack to track active method invocations, and a current execution context. A data stack st consists of a list of data frames, each of which maps local variables to addresses. A data store σ maps addresses to objects. A call stack cs consists of a list of call frames, and each call frame consists of the statement to return to,



■ **Figure 5** The overview of our two-phased test generation method.

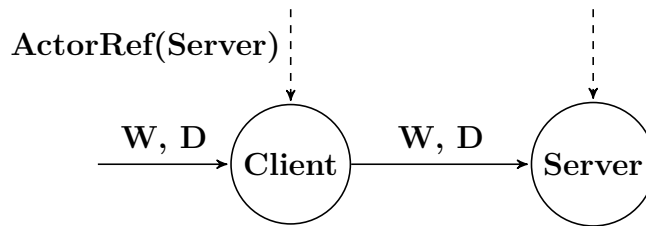
the context to restore, and the address to store the return value. An object o consists of a heap context and a list of fields. An address is a location that holds an object. An address a consists of either a local variable and its regular context (allocated for a local variable) or a field and its heap context (allocated for a field). In the concrete semantics, every dynamic object instance has a unique heap context, and every dynamic method call has a unique regular context.

We express the concrete semantics of our language as a transition relation (\Rightarrow_C) from one configuration to another. Figure 4 shows the semantics of actor operations only. The semantics of local computations in an actor is similar to the normal semantics of Java, and thus omitted. For simplicity, we use underscore $_$ in our transition rules, to represent the remaining states in a tuple that are neither used nor updated in the transition. We use standard functions $car, cdr, cons, list$ to manipulate lists, and define a number of helper functions: $succ$ returns the next statement given the label of the current statement, \mathcal{F} returns a list of field names for a given class, cls returns the class name of a given object, and rec returns the declaration of the `onReceive` method of a given class. We use the operator \bullet to add an element to a set, and the notation $+$ to insert or update (if existing) entries in a map. We use `nil` as the null value for every domain. A *fresh* value means that a new value is generated from the corresponding domain. The symbols a_{this}, a_{self} , and a_{sender} represent reserved addresses to store the *this* object, the actor reference of itself, and the actor reference of a sender, respectively.

The *Actor Creation* rule says that a new actor is created with a fresh reference r' in the system. The actor has an initial state, where the current statement is `nil`. The *Message Sending* rule defines the asynchronous semantics of sending messages. The new message is put in the set of pending messages μ , and the sending actor continues its execution. Note that messages are immutable so that there are no concurrent writes on messages. The *Message Receiving* rule says that an actor can receive a message only when it is ready (i.e., the statement is `nil`). Upon receiving the message, the `onReceive` method is invoked, and the message is no longer pending and thus removed from μ . After executing the `onReceive` method, the statement is set to `nil`, signifying that the actor is ready to receive a message again. The *Self Reference* rule says that the actor reference of the *this* object is assigned to a local variable v . Similarly, the *Sender Reference* rule says that the actor reference of the message sender is assigned to a local variable v .

4 Message Flow Graph Construction

Our test generation method operates in two phases as shown in Figure 5. In the first phase, we use static analysis to construct a *message flow graph* (MFG), an abstraction of an actor system that models potential interactions (i.e., actor creation and communication) between



■ **Figure 6** The MFG of the Bank Account example. The symbols W and D represent the withdraw message and the deposit message, respectively.

actors in the system. The input to our MFG analysis is the system under test including the code and the specified receptionists, and the output is an MFG of the system. In the second phase, we use BSE to generate a test that covers a given target. To generate tests that exercise multiple actors, BSE must go across actors. The MFG from the first phase is a key input that enables *inter-actor* BSE. After BSE reaches the entry of the message handler on an actor a , it queries the MFG to obtain actors that can send the required message to the actor a . Then BSE picks one potential sender, jumps to the exit of the message handler of the sender, and continues with the previous path constraint carried over. When a feasible path is found during the path exploration, we generate the test from the path constraint. We next explain the MFG construction and the BSE (in Section 5) in details.

An MFG is a directed graph between *abstract* objects, where an abstract object represents multiple concrete objects of the same class whose field values have been merged into a set. Specifically, a node in the MFG represents an abstract actor and a directed edge between two nodes means that the abstract actor represented by the source node either creates or sends a message to the abstract actor represented by the sink node. MFG edges are labeled with abstract constructor parameters for actor-creation edges and abstract messages for message-sending edges. Note that the MFG edges do not indicate the acquaintance between actors—it is possible that an actor a knows of another actor b , but there is no edge from a to b because a neither creates b nor sends a message to b .

An abstract object may be replaced by its class if there is only one abstract object per class. Figure 6 shows the MFG of the Bank Account example. There are two actors, `Client` and `Server`, in the graph. The symbols W and D represent the `WithdrawMessage` and the `DepositMessage`, respectively. Both actors are created (creation edges are represented with dashed arrows) by the external environment. The `Client` is initialized with an actor reference mapped to the `Server`, and it can send `WithdrawMessage` and `DepositMessage` to the `Server`. The `Client` is the only receptionist of the system and can receive `WithdrawMessage` and `DepositMessage` from the external environment.

To construct a MFG, we need to not only resolve the recipient of each message-sending site and the actor being created of each actor-creation site, but also pass along the messages and constructor parameters between actors. This is because the message and constructor parameters can affect the analysis of receiving actors. We use points-to analysis to compute the points-to sets for messages, constructor parameters, and actor references. In addition, we model the semantics of actor operations so that analysis information can be carried across actor boundaries. In particular, the actor creation operation conceptually creates two objects: an actor reference object and a corresponding actor object. Our analysis keeps track of such mappings to resolve the actor being created, and passes the points-to set of constructor parameters to this actor for instantiation.

$$\begin{aligned}
\omega \in \Omega &= ActorState \times Graph \times RefMap \\
\gamma \in RefMap &= ActorRef \rightarrow (ClassName \times Obj) \\
r \in ActorRef &\subset Obj \\
G \in Graph &= ActorRef \times ActorRef \times Type \rightarrow (\mathcal{P}(Obj))^* \\
Type &= \{\text{create}, \text{send}\} \\
\varsigma \in ActorState &= Stmt \times Stack \times Store \times CallStack \times Context \\
st \in Stack &= (Var \rightarrow Addr)^* \\
\sigma \in Store &= Addr \rightarrow \mathcal{P}(Obj) \\
o \in Obj &= HContext \times (FieldName \rightarrow Addr) \\
cs \in CallStack &= (Stmt \times Context \times Addr)^* \\
a \in addr &= (Var \times MethodName \times Context) \cup (FieldName \times HContext) \\
Context &= HContext = Lab
\end{aligned}$$

■ **Figure 7** State space of the small-step state machine.

Note that passing only the type of the message or constructor parameter between actors can result in unacceptable imprecision in our analysis. For example, a common case is that an actor reference \mathbf{r} is sent as a message to a recipient actor \mathbf{A} ; \mathbf{A} receives \mathbf{r} and then sends a message to \mathbf{r} . When resolving \mathbf{r} in \mathbf{A} , we only know that the type of \mathbf{r} is `ActorRef`, but we know nothing about the actor that lives in \mathbf{r} . Thus, we have to conservatively assume all actor classes in our system may live in \mathbf{r} , and add a message-sending edge from \mathbf{A} to every actor class. To avoid such imprecision, we need to pass along the points-to sets of messages and constructor parameters instead of their types. We next formally describe our analysis.

4.1 Analysis Semantics

We express the semantics of our analysis using small-step state machines, each modeling one abstract actor. Communication between actors is modeled by global states shared across state machines. The domain Ω of a state machine is defined in Figure 7. The reference map γ stores the mappings between actor references and the actors created in the system. The graph G records the actor-creation and message-sending events between actors. Specifically, G maps a tuple of a source actor reference, a sink actor reference, and an operation type to a list of points-to sets of messages or constructor parameters. Visually, an entry in the map can be seen as a directed edge, with the label being the list of points-to sets. The *ActorState* is similar to the one defined in concrete semantics of our language except that now the *ActorState* is an abstract state: the store maps an address to a set of objects rather than one object; the regular and heap contexts are a finite set of statement labels. An important design decision made by our analysis is that we create only one abstract actor object per actor class. That is, actors of the same class created in different sites are merged into one abstract actor object by merging the points-to sets of the corresponding fields. In this way, we only need to create one state machine per actor class, making our analysis faster and more scalable. The incurred imprecision can be refined by the BSE in phase II because our BSE distinguishes every concrete actor.

Actor Creation

$$\begin{aligned}
& ((\llbracket v = \text{create}(C.\text{class}, \vec{v}^{\rightarrow}); \ell \rrbracket, st, \sigma, _), G, \gamma) \Rightarrow_A ((\text{succ}(\ell), st, \sigma', _), G', \gamma'), \text{ where} \\
& (\gamma', r) = \text{getRef}(\gamma, C) \quad \sigma' = \sigma \sqcup [st(v) \mapsto \{r\}] \quad r' \in \sigma(a_{\text{self}}) \\
& G' = \text{merge}(G, [(r', r, \text{create}) \mapsto \text{list}(\sigma(st(v'_i)))]))
\end{aligned}$$

Message Sending

$$\begin{aligned}
& ((\llbracket v.\text{send}(v'); \ell \rrbracket, st, \sigma, _), G, _) \Rightarrow_A ((\text{succ}(\ell), st, \sigma, _), G', _), \text{ where} \\
& r \in \sigma(st(v)) \quad r' \in \sigma(a_{\text{self}}) \quad G' = \text{merge}(G, [(r', r, \text{send}) \mapsto \text{list}(\sigma(st(v')))]))
\end{aligned}$$

Message Receiving

$$\begin{aligned}
& ((\text{nil}, st, \sigma, cs, c), G, _) \Rightarrow_A ((s, st', \sigma', cs', c'), G, _), \text{ where} \\
& o_0 \in \sigma(a_{\text{this}}) \quad \llbracket \text{void onReceive}(C v) \{C' v'; \vec{s}^{\rightarrow}\} \rrbracket = \text{rec}(\text{cls}(o_0)) \quad s = \text{car}(\vec{s}^{\rightarrow}) \\
& (hc_0, _) = o_0 \quad c' = hc_0 \quad a = (v, \text{onReceive}, c') \quad a'_i = (v'_i, \text{onReceive}, c') \\
& st' = \text{cons}([v \mapsto a, v'_i \mapsto a'_i], st) \quad cs' = \text{cons}((\text{nop}, c, \text{nil}), cs) \quad r \in \sigma(a_{\text{self}}) \\
& O_r = \text{preds}(G, r, \text{send}, \gamma) \quad O \in \{\text{car}(G((r', r, \text{send}))) \mid r' \in O_r\} \\
& \sigma' = \sigma \sqcup [a \mapsto O, a_{\text{sender}} \mapsto O_r]
\end{aligned}$$

Self Reference

$$((\llbracket v = \text{self}; \ell \rrbracket, st, \sigma, _), _) \Rightarrow_A ((\text{succ}(\ell), st, \sigma \sqcup [st(v) \mapsto \sigma(a_{\text{self}})], _), _)$$

Sender Reference

$$((\llbracket v = \text{sender}; \ell \rrbracket, st, \sigma, _), _) \Rightarrow_A ((\text{succ}(\ell), st, \sigma \sqcup [st(v) \mapsto \sigma(a_{\text{sender}})], _), _)$$

■ **Figure 8** Abstract semantics for actor operations in MFG analysis.

The analysis semantics is defined by the transition relation $(\Rightarrow_A) \subset \Omega \times \Omega$. The analysis semantics of local computations is precisely the 1-object-sensitive points-to analysis [27]. We provide the transition rules for local computations in the appendix. Figure 8 describes transition rules for the actor operations. The *getRef* function checks if the given class C is in the value set of γ . If found, it returns itself and the key of the value. If not found, it adds an entry $r \mapsto (C, \text{nil})$ to γ , where r is fresh, and returns the updated γ' and r . Since only one abstract actor object is created per actor class, an actor class can appear in at most one tuple in the value set of γ . The *merge* function merges the labels of edges with the same source and sink. The *preds* function finds all predecessors of a given type for a node r in the graph G and returns the set of actor objects mapped by the predecessors in γ .

In the *Actor Creation* rule, instead of instantiating the actor object at the creation site, an actor-creation event is recorded and merged into the graph. Subsequently, when a state machine for this actor class is created, actor-creation events are used to instantiate the single abstract actor object for this class. Similarly in the *Message Sending* rule, a message-sending event is recorded and merged into the graph. The *Message Receiving* rule says that the *onReceive* method of the actor is invoked upon receiving a message. The graph G is queried to find the set of all possible senders O_r , and the set of all possible messages received by O . Note that when updating the call stack, we use **nop** instead of **nil** for the statement to return to. **nop** indicates no operation to be performed and stops the state machine. Otherwise, the state machine will not halt.

Algorithm 1: Iterative MFG construction.

```

Input  : An Actor system  $P$ , a raw graph  $G \in Graph$ , and an actor reference map
            $\gamma \in RefMap$ 
Output : A message flow graph of  $P$ 

1  $worklist \leftarrow []$     $factStore \leftarrow []$ 
2  $worklist.appendAll(\gamma.keySet())$ 
3 while  $worklist$  not empty do
4    $r \leftarrow worklist.removeFirst()$ 
5    $beforeFacts \leftarrow InEdges(r, G)$ 
6   if  $factStore[r] \neq beforeFacts$  then
7      $factStore[r] \leftarrow beforeFacts$ 
8      $\mathcal{M}_r \leftarrow CreateStateMachine(r, G, \gamma)$ 
9      $\mathcal{M}_r.execute()$ 
10     $worklist.appendAll(Successors(r, G))$ 
11  end
12 end
13 return CollapseToMFG( $\gamma, G$ )

14 Procedure CreateStateMachine( $r, G, \gamma$ )
15    $(C, \_) \leftarrow \gamma(r)$     $\vec{f} \leftarrow \mathcal{F}(C)$     $a_i \leftarrow (f_i, \ell_C)$ 
16    $o \leftarrow (\ell_C, [a_i \mapsto f_i])$  // actor allocation
17    $\gamma \leftarrow \gamma + [r \mapsto (C, o)]$  // ref map update
18    $\vec{O} \leftarrow [\emptyset, \dots, \emptyset]$  // a list of points-to sets
19   foreach  $(r', r'', create) \mapsto \vec{O}'$  in  $InEdges(r, G)$  do
20      $O_i \leftarrow O_i \cup O'_i$ 
21   end
22    $\sigma \leftarrow [a_{this} \mapsto \{o\}, a_i \mapsto O_i, a_{self} \mapsto \{r\}]$ 
23    $\omega_0 \leftarrow ((nil, [], \sigma, [], nil), G, \gamma)$ 
24   Create  $\mathcal{M}_r$  with the initial state  $\omega_0$ 
25   return  $\mathcal{M}_r$ 
26 End

```

4.2 MFG Construction Algorithm

Algorithm 1 shows our iterative algorithm to construct the MFG. The algorithm takes as input an actor system P , a raw graph G , and a reference map γ , and outputs an MFG graph. G and γ are initialized from the driver code that sets up the actor system. Initially, G contains actor-creation and message-sending events by the external environment, and γ contains the mappings for actors created by the external environment. For each actor class, one state machine is instantiated to model the abstract actor of this class. The algorithm maintains a *worklist* that keeps track of the abstract actors to be analyzed next as well as a *factStore* that stores the relevant data facts for each abstract actor. The data facts for an abstract actor are essentially the set of incoming edges of this actor node in G , and these facts affect the initial state of the state machine for this actor.

The algorithm starts with pushing the initial actors onto the *worklist* (Line 2), and iteratively analyzes these actors one at a time. Before the analysis, the algorithm computes the relevant data facts for this actor from G (Line 5). It then checks whether the facts are changed, by comparing the computed facts with the previous facts stored in *factStore*. If changed, the algorithm updates the facts for this actor in *factStore* (Line 7), analyzes this actor with these new facts by instantiating and running the state machine described in

Section 4.1 (Lines 8-9), and pushes all the successors of this actor node onto *worklist* (Line 10). Otherwise, the algorithm skips this actor because the execution of its state machine will yield the same result and will not change the global state G . This process continues until *worklist* is empty, indicating a fixed point is reached. The `CreateStateMachine` procedure is the only place where instantiations of abstract actors happen. The constructor parameters of multiple actor-creation edges are merged (Lines 18-21) and the results are used to initialize the fields of the abstract object (Line 22). Finally, the algorithm builds an MFG from G and γ by collapsing the abstract objects of nodes and labels into classes. If an object is an actor reference, we also encode the class of the underlying actor into the MFG.

4.3 Optimizations

Our analysis applies two lightweight yet effective optimizations to actor classes based on the code pattern in actor programs. Since actors often receive multiple types of messages and behave differently for each message type, a common code pattern in actors' `onReceive` methods is that an *if* statement is used at the top of its control flow to check the message type and process one type of message in one branch. In our running example, both the `Client` and the `Server` actors follow this pattern.

Our first optimization eliminates unreachable code based on the potential types of the message in our analysis. Specifically, we compute the potential types from the points-to set of the message and analyze only the branches of the top *if* statement that may be taken under these message types. Our second optimization is based on the idea that when a message must be of a certain type under some context, we can safely remove objects that are not an instance of this type from the points-to set of this message. The optimization works as follows: after entering a branch of the top *if* statement, we carry the corresponding type constraint of the message (obtained from the condition of the *if* statement) with our analysis. That is, whenever we query the points-to set of the message in this branch, an additional filter function $f : \mathcal{P}(Obj) \times ClassName \rightarrow \mathcal{P}(Obj)$ is applied to the original points-to set to filter out objects that are not an instance of the given type. Our evaluation shows that these optimizations significantly reduce the size of the MFGs.

Example. Let us illustrate the optimizations using the `Client` actor in Figure 1. Suppose that the points-to set of the `message` parameter in the `onReceive` method contains only one `DepositMessage` message. Based on the first optimization, we only need to analyze the second branch of the *if* statement (Lines 20 - 22) instead of the whole method. To illustrate our second optimization, we now suppose that the points-to set of the `message` parameter contains a `WithdrawMessage` message and a `DepositMessage` message. Then both branches of the *if* statement must be analyzed. When analyzing its first branch (Lines 14-18), we know that the `message` parameter must be of the type `WithdrawMessage`. With this type constraint, we can remove the `DepositMessage` message from the points-to set in this branch because it is not an instance of the type `WithdrawMessage`. Hence, we can conclude that at Line 17, `message` must point to a `WithdrawMessage` message rather than may point to a `WithdrawMessage` message or a `DepositMessage` message. Similarly, the optimization can be applied to the second branch as well.

5 Test Generation

In phase II, we use backward symbolic execution to generate tests for the target. BSE starts from the target, and performs a backward exploration, searching for a feasible path to the entry points of the system. Constraints over the execution are collected and used to generate

$$\begin{aligned}
\alpha \in ActorMap &= ActorRef \rightarrow ActorState \\
Event &= SendingEvent \cup CreationEvent \\
SendingEvent &= \widehat{ActorRef} \times \widehat{ActorRef} \times \widehat{Var} \times \widehat{Time} \\
CreationEvent &= \widehat{ActorRef} \times \text{ClassName} \times (\widehat{Var})^* \times \widehat{Time} \\
\varsigma \in ActorState &= LocalState \times \widehat{Time} \times Requests \\
\beta \in LocalState &= Stmt \times CallStack \times \widehat{Var} \\
cs \in CallStack &= (Stmt \times \widehat{Var} \times \widehat{Var})^* \\
Q \in Requests &= \widehat{Var} \times \widehat{ActorRef} \times \widehat{Time}
\end{aligned}$$

$\widehat{Var}, \widehat{ActorRef}, \widehat{Time}$ are sets of free variables in first order logic.

■ **Figure 9** State space of the backward symbolic execution.

the test. The generated test consists of the messages sent to relevant actors as well as the message receiving orders.

The semantics of BSE is formally defined as a transition relation \Rightarrow_S from one symbolic configuration to another symbolic configuration. A symbolic configuration is a tuple,

$$\langle \alpha \mid \mu \mid \phi \mid \chi \rangle$$

where α represents relevant actors in BSE and is a map from a finite set of actor references to actor states, μ is a finite set of pending events (including both actor creation and message sending events). ϕ is the path condition collected over the transitions, and χ is the set of external messages to the system. The domain of ϕ is the quantifier-free formulae in *first-order logic* (FOL) with equality. The domain of the remaining configuration is described in Figure 9. Note that $\widehat{Var}, \widehat{ActorRef}, \widehat{Time}$ are sets of free variables in FOL, which can hold values of primitives and references. A message-sending event consists of the actor reference of the sender, the actor reference of the recipient, the message, and the time when the message is sent. An actor-creation event consists of the actor reference of the actor being created, the type of the actor, and a list of constructor parameters, and the creation time. An actor state consists of a local state, the current local time of the actor, and a set of message requests.

Since BSE goes backwards, a message request under this context indicates that a certain message is required in order for the execution to reach this point, yet this message is not in the mailbox of that actor. For each message request, BSE attempts to find an actor that can send the corresponding message, and thus “fulfill” this request. The local state consists of the current statement, the call stack, and a variable representing the receiver object of the current method call. A message request consists of a message, an actor reference for the sender, and the time of receiving the message. The call stack consists of a list of call frames, and each call frame consists of the statement to return to, the variable of the return value, and the variable of the caller object. \widehat{Time} is a set of integer variables.

To describe the BSE semantics, we add two additional types of statements to our language as indicators of reaching the entry of a method. We use `entryR`; as the first statement for every `onReceive` method, and use `entry`; as the first statement for all other methods. For space considerations, the formal semantics of local computations in BSE is described in the appendix.

Actor Creation

$$\begin{aligned} \langle \alpha \bullet r \mapsto ((\llbracket v = \text{create}(C.\text{class}, \vec{v}^{\hat{t}}); \ell \rrbracket, _) , \hat{t}, _) \mid \mu \mid \phi \mid _ \rangle \Rightarrow_S \\ \langle \alpha \bullet r \mapsto ((\text{pred}(\ell), _) , \hat{t}', _) \mid \mu' \mid \phi' \mid _ \rangle, \text{ where} \\ \hat{t}', \hat{r}' \text{ are fresh} \quad \phi' = \phi[\hat{r}'/\hat{v}] \wedge \hat{t}' < \hat{t} \quad \mu' = \mu \cup \{(\hat{r}', C, \vec{v}^{\hat{t}}, \hat{t})\} \end{aligned}$$

Message Sending

$$\begin{aligned} \langle \alpha \bullet r \mapsto ((\llbracket v.\text{send}(v'); \ell \rrbracket, _) , \hat{t}, _) \mid \mu \mid \phi \mid _ \rangle \Rightarrow_S \\ \langle \alpha \bullet r \mapsto ((\text{pred}(\ell), _) , \hat{t}', _) \mid \mu' \mid \phi' \mid _ \rangle, \text{ where} \\ \hat{t}' \text{ is fresh} \quad \phi' = \phi \wedge \hat{t}' < \hat{t} \quad \mu' = \mu \cup \{(\hat{r}, \hat{v}, \hat{v}', \hat{t})\} \end{aligned}$$

Actor Entry-Existing Actor

$$\begin{aligned} \langle \alpha \bullet r \mapsto ((\llbracket \text{entry}_R; \ell \rrbracket, _) , \hat{t}, Q) \mid _ \mid \phi \mid _ \rangle \Rightarrow_S \\ \langle \alpha \bullet r \mapsto ((\text{nil}, _) , \hat{t}', Q') \mid _ \mid \phi' \mid _ \rangle, \text{ where} \\ \hat{t}' \text{ is fresh} \quad \phi' = \phi \wedge \hat{t}' < \hat{t} \quad \llbracket \text{void onReceive}(C'v')\{\overrightarrow{C''v''}; \vec{s}\} \rrbracket = \text{method}(\ell) \\ Q' = Q \cup \{(\hat{v}', r_{\text{sender}}, \hat{t})\} \end{aligned}$$

Actor Entry-New Actor

$$\begin{aligned} \langle \alpha \bullet r \mapsto ((\llbracket \text{entry}_R; \ell \rrbracket, _) , \hat{t}, Q) \mid _ \mid \phi \mid _ \rangle \Rightarrow_S \\ \langle \alpha \bullet r \mapsto ((\text{nil}, _) , \hat{t}', Q') \bullet r' \mapsto ((\text{nil}, [], \hat{v}_0^{\hat{t}'}, \hat{t}'', []) \mid _ \mid \phi' \mid _ \rangle, \text{ where} \\ \hat{t}', \hat{t}'', \hat{r}', \hat{v}_0^{\hat{t}'} \text{ are fresh} \quad \llbracket \text{void onReceive}(C'v')\{\overrightarrow{C''v''}; \vec{s}\} \rrbracket = \text{method}(\ell) \\ C \in \text{predCls}(AC(r)) \quad Q' = Q \cup \{(\hat{v}', r_{\text{sender}}, \hat{t})\} \quad \phi' = \phi \wedge \hat{t}' < \hat{t} \wedge \hat{t}'' < \hat{t} \end{aligned}$$

Messaging Event Matching-Internal

$$\begin{aligned} \langle \alpha \bullet r \mapsto (_, Q \bullet (\hat{v}, r_{\text{sender}}, \hat{t})) \mid \mu \bullet (\hat{r}', \hat{r}'', \hat{v}', \hat{t}') \mid \phi \mid \chi \rangle \Rightarrow_S \\ \langle \alpha \bullet r \mapsto (_, Q) \mid \mu \mid \phi' \mid \chi \rangle, \text{ where} \\ \phi' = \phi \wedge \hat{r} == \hat{r}'' \wedge \hat{v} == \hat{v}' \wedge r_{\text{sender}} == \hat{r}' \wedge \hat{t}' < \hat{t} \end{aligned}$$

Messaging Event Matching-External

$$\langle \alpha \bullet r \mapsto (_, Q \bullet (\hat{v}, r_{\text{sender}}, \hat{t})) \mid _ \mid \chi \rangle \Rightarrow_S \langle \alpha \bullet r \mapsto (_, Q) \mid _ \mid \chi \cup \{(\hat{r}, \hat{v})\} \rangle$$

Creation Event Matching

$$\begin{aligned} \langle \alpha \bullet r \mapsto ((\text{nil}, _, \hat{v}_0), \hat{t}, []) \mid \mu \bullet (r', AC(r), \vec{v}^{\hat{t}}, \hat{t}') \mid \phi \mid _ \rangle \Rightarrow_S \langle \alpha \mid \mu \mid \phi' \mid _ \rangle, \text{ where} \\ \vec{f} = \mathcal{F}(AC(r)) \quad \phi' = \phi \wedge \hat{r} == \hat{r}' \wedge \text{read}(\hat{v}_0, f_i) == \hat{v}_i \wedge \hat{t}' < \hat{t} \end{aligned}$$

OnReceive Looping

$$\begin{aligned} \langle \alpha \bullet r \mapsto ((\text{nil}, _) , _) \mid _ \rangle \Rightarrow_S \langle \alpha \bullet r \mapsto ((\text{last}(\vec{s}), _) , _) \mid _ \rangle, \text{ where} \\ \llbracket \text{void onReceive}(C'v')\{\overrightarrow{C''v''}; \vec{s}\} \rrbracket = \text{rec}(AC(r)) \end{aligned}$$

Self Reference

$$\langle \alpha \bullet r \mapsto ((\llbracket v = \text{self}; \ell \rrbracket, _) , _) \mid \phi \mid _ \rangle \Rightarrow_S \langle \alpha \bullet r \mapsto ((\text{pred}(\ell), _) , _) \mid \phi[\hat{r}/\hat{v}] \mid _ \rangle$$

Sender Reference

$$\langle \alpha \bullet r \mapsto ((\llbracket v = \text{sender}; \ell \rrbracket, _) , _) \mid \phi \mid _ \rangle \Rightarrow_S \langle \alpha \bullet r \mapsto ((\text{prev}(\ell), _) , _) \mid \phi[r_{\text{sender}}/\hat{v}] \mid _ \rangle$$

 ■ **Figure 10** Transition rules for actor operations in backward symbolic execution.

5.1 Semantics Of Actor Operations In BSE

Figure 10 shows the semantics of BSE for actor operations. We put a hat on a symbol to represent a free variable in ϕ . For example, we use \hat{v} in ϕ to represent the corresponding variable v is free. Note that for variables with the same name in different execution contexts, we create distinct variables in ϕ to represent them. The notation $\phi[\hat{v}'/\hat{v}]$ means that every occurrence of \hat{v} in ϕ is syntactically replaced by \hat{v}' . It is important to note that whenever such substitutions happen in ϕ , we also perform the corresponding substitutions in the rest of the symbolic configuration. For readability, we omit these subsequent substitutions in our transition rules. We use a number of helper functions in our transition rules. The function *pred* returns the previous statement of a given label, and the function *last* returns the last element of a given list. The function *method* returns the method that encloses the statement with the given label. The function *AC* returns the class name of the actor object mapped by the given actor reference. The function *read* takes as input a free variable representing an object and the field name, and returns the variable representing the field. The function *predCls* takes as input a class name, locates the node of this class in the MFG, finds the predecessors of the node, and returns a set of class name of the predecessors.

The *Actor Creation* rule and the *Message Sending* rule say that upon an actor-creation or message-sending operation, an actor-creation or a message-sending event is added to a pool of pending events μ . Every actor keeps a local time \hat{t} , and increases its local time when an actor operation is performed. Hence, the constraint $\hat{t}' < \hat{t}$ indicates that the operation at \hat{t}' happens before the operation at \hat{t} . The *Actor Entry* rules describe potential transitions when BSE reaches the entry of the `onReceive` method of an actor. Reaching the entry of the `onReceive` method implies that this actor must have been created and have received a message. Thus, in both *Actor Entry* rules, a corresponding message request is added to the set Q , indicating that the specific message is required in order for the execution to reach this point, and BSE needs to find an actor that sends the message. There are two possibilities concerning who may create this actor or send a message to this actor. The *Actor Entry-Existing Actor* describes one possibility that this actor is created by an existing actor in α , and the message is also sent from an existing actor; there is no need to introduce new actors in α . The *Actor Entry-New Actor* describes the other possibility: either the actor creation or the message send is done by actors not in α . As a result, a new actor is added to α . The MFG is queried to obtain the predecessors of this actor class, which is the set of actor classes that may create or send a message to this actor. Then an actor with the default initial state is created in α with its type being one of the predecessors. This is the only rule that introduces new actors to our exploration.

A message request is fulfilled either by a pending message event in μ sent from an actor inside the system or, if the actor is a receptionist, by a message sent from the external environment. The *Messaging Event Matching-Internal* rule describes the first case, in which the matched request and event are removed from Q and μ respectively, and a happens-before constraint between the message receive and send operations is added to ϕ . The *Messaging Event Matching-External* rule describes the second case, in which the request is removed from Q , and an external message is added to χ . The *Creation Event Matching* rule says that a pending actor-creation event is matched with an actor in α . Note that to match a creation event, the type of the actor must be the same as the type specified in the creation event, and the message request set Q of the actor must be empty, indicating all message requests are fulfilled. The *Receive Looping* rule says that an idle actor can start an execution from the exit of the `onReceive` method.

5.2 Path Exploration In BSE

The initial symbolic configuration is that the actor map α contains only one actor with the statement being the target, and the event pool μ contains the actor-creation events from the external environment. BSE starts with the initial configuration and takes one transition at a step. The computation branches when multiple transition rules can be matched on one configuration. BSE uses the depth-first search strategy for path exploration. At each branching point, we pick one transition from all enabled transitions, and check if the path constraints in the new configuration is satisfiable. If satisfiable, we continue the exploration on the new configuration; otherwise, we backtrack. The final accepting configurations are the ones with α being empty and ϕ being satisfiable. A system test can be constructed from the model of ϕ , the transition path, and the set of external messages.

Because actors in the configuration proceed their computations concurrently, almost any configuration has multiple enabled transitions. As a result, the search space in BSE is intractable. To address this problem, we propose two search heuristics and a feedback-directed search technique to efficiently find a feasible path in the huge search space.

Search heuristics. Our first heuristic is that BSE always explores a message handler atomically. In other words, once BSE starts a transition of local computations in a message handler of an actor, all transitions on other actors are disabled and BSE will keep exploring this message handler until reaching the entry of the message handler. As a result, the number of enabled transitions on each symbolic configuration is reduced. This heuristic leverages the atomicity of the macro-step semantics [8] in the Actor model—messages to a given actor are processed one at a time without interleaving. Macro-step is also enabled by the fact that the concurrent execution of message handlers on different actors need not be interleaved (i.e., messages to different actors can be sequentialized). This is because actors do not share states. Therefore, the heuristic is safe: it reduces the search space in BSE without missing any tests that can potentially cover the target.

Our second heuristic keeps the number of actors in the generated test small in order to avoid exploring unnecessary paths. This heuristic is based on the conjecture that most concurrency bugs may be triggered by considering interactions of a small number of actors. The conjecture is the result of a previous finding that most concurrency bugs in multi-threaded programs can be triggered using two threads [23]. With this conjecture, we assign different weights to transition rules for actor operations. When multiple transition rules are enabled on a configuration, the probability of picking a rule is based on its weight (rules with more weights have a higher chance of being picked). We give a much lower weight to the *ActorEntry – NewActor* rule, which is the only rule to introduce new actors to a test. This is because introducing a new actor opens up a whole new search space – BSE has to find a feasible execution trace on this actor. In this way, we keep the number of actors in our test small, and avoid fruitless explorations. In addition, we give more weights to transition rules that consume pending events in the event pool μ so that message requests from actors can be fulfilled as soon as possible. Recall that a test is generated only when BSE reaches a final accepting configuration, where the actor map α must be empty. An actor is removed from α only when all of its message requests are fulfilled. Hence, fulfilling these message requests helps BSE find a test efficiently.

Feedback-directed search. Heuristics do not always work well. There are cases where a large number of transitions are enabled, but only a few of them can lead to a feasible path. If the heuristics do not bias towards these transitions, BSE will frequently hit infeasible paths.

```

1 private int pingsLeft = 100;
2 public void onReceive (Object message) {
3     if (message instanceof PongMessage) {
4         pongActor.tell(new PingMessage(), getSelf());
5         pingsLeft--;
6         if (pingsLeft == 0) {
7             // target
8         } ...
9     } ...
10 }

```

■ **Figure 11** An example from our subjects that illustrates the feedback-directed search technique.

The feedback-directed search technique guides BSE out from such undesirable situations by leveraging the unsatisfiable cores of the path constraint from the previous infeasible paths. An unsatisfiable core is a subset of clauses in the original constraint such that the conjunction of these clauses is unsatisfiable. To make the path constraint feasible, the clauses in the unsatisfiable core need to be changed. The idea of our feedback-directed technique is to drive the execution towards the code that changes the values of the variables in the unsatisfiable core, hoping that the changes will make the path constraint satisfiable.

Our feedback-directed technique has two steps. In the first step, we identify a set of code instructions that can potentially change the unsatisfiable core. We obtain the unsatisfiable core of the path constraint directly from the underlying SMT solver Z3 [14]. Then we extract all the variables from the unsatisfiable core, and map these variables to corresponding program variables. This can be done without additional overhead, because our symbolic execution keeps track of the mapping between the variables in path constraints and program variables. For each program variable, we identify a set of instructions that define this variable (definition sites). In our implementation, BSE is performed on an IR that is in the static-single-assignment form. Hence, there is only one definition site per variable. In the second step, we drive the execution to the definition sites identified in the first step. To do this, we compute the transitions that may lead to at least one of these definition sites. A transition may lead to a definition site if the statement transitioned to is reachable from the definition site in the inter-procedural control flow graph. We prioritize these transitions over the others.

Figure 11 shows the message handler of the ping actor in the Ping-Pong example. The `pingsLeft` field keeps track of the ping messages sent out, and is initially set to 100. To cover the target at Line 7, the ping actor has to receive 100 pong messages. Suppose that when BSE first reaches the entry of the message handler from the target, it chooses to jump to the constructor of the ping actor, meaning that only one pong message is received after creating this actor. Obviously, this path is infeasible. Its path constraint is $p = 100 \wedge p - 1 = 0 \wedge \text{subType}(\text{type}(m), \text{PongMessage})$, where p and m map to the program variables `pingsLeft` and `message`, respectively. The unsatisfiable core of this path constraint is $\{p = 100, p - 1 = 0\}$, whose only variable maps to `pingsLeft`. Thus, Line 1 and Line 5 are identified as the definition sites for `pingsLeft`. Then BSE backtracks to the entry of the message handler, and picks the transition that jumps to Line 8, because it may lead to the definition site at Line 5. This transition indicates that the ping actor has received two pong messages. Note that BSE does not pick the transition that may lead to Line 1 (i.e., the transition that jumps to the constructor), because it has been explored previously, leading to an infeasible path. This process iterates 100 times and BSE finds a feasible path in which the ping actor receives 100 pong messages. Without this technique, in each iteration, BSE may try other messages that do not affect `pingsLeft`, thus making the search inefficient.

■ **Table 1** Characteristics of the subjects in our evaluations.

Subjects	LOC	Description
Micro Bench.	50 - 200	8 well-known actor example programs
Concurrency Bench.	100 - 400	8 classic concurrency problems
Parallelism Bench.	200 - 1,000	14 realistic parallel applications
AkkaCrawler	715	A web crawler and indexer
Batch	1,309	A concurrent batch processing framework
Parallec	12,457	A parallel client firing requests and aggregating responses
Stone	20,935	An online game server framework

6 Implementation

We implement our method in a tool called TAP for actor systems developed with Java *Akka*. TAP is built on top of WALA [5], a static analysis infrastructure for Java. TAP transforms the Java bytecode of the system under test to WALA IR and performs analysis on WALA IR. The benefit of working on WALA IR is that one can directly use the basic built-in analyses provided by WALA. TAP uses multiple WALA built-in analyses such as class hierarchy analysis, call graph analysis, and points-to analysis. Since Scala *Akka* programs are also compiled to Java bytecode, TAP in principle may be used to analyze Scala *Akka* programs as well. However, Scala *Akka* has a different set of interfaces, and substantial engineering work is required to support Scala *Akka*. We plan to support Scala *Akka* in the future.

TAP consists of two major components, an MFG builder containing $\sim 4,000$ lines of Java code and a BSE engine containing $\sim 11,000$ lines of Java code. The implementation of the MFG builder closely follows the formalizations and the iterative MFG construction algorithm described in Section 4. A key part for MFG construction is resolving recipients and messages in message-sending sites. TAP maintains a map from an actor reference to a set of actor objects that are possibly referenced by it. This map is used to resolve `ActorRef` pointers. TAP queries WALA’s points-to analysis to resolve all other pointers.

The BSE engine includes a backward symbolic interpreter on WALA IR as well as the search techniques. The interpreter implements a transition rule (similar to the semantic rules in our BSE formalization) for each type of statements in WALA IR. The actor library calls are interpreted using our semantic models so that TAP does not explore the actor library methods. The BSE engine forks a new symbolic configuration whenever the computation branches. TAP uses Z3 [14] as the off-the-shelf SMT solver for solving path constraints and computing unsatisfiable cores. An important deviation from the formalizations is that TAP implements the FIFO message delivery semantics, because our target actor framework, Java *Akka* guarantees the FIFO semantics. To implement the FIFO semantics, TAP models the pending messages as a set of lists rather than a multi-set. Each list models a FIFO communication channel between a pair of actors so that the message sending order is preserved.

7 Evaluation

We evaluate TAP on a set of third-party benchmarks called *Savina* [21] as well as four randomly selected open-source projects from GitHub. Our experiments consist of two parts: 1) the evaluation on the MFG construction analysis, measuring the size of the MFGs, analysis time, and the effectiveness of the optimizations; 2) the evaluation on the effectiveness of our test generation method.

■ **Table 2** Comparison between the baseline MFG analysis and the optimized MFG analysis. The numbers for the three benchmark categories are averages.

Subjects	Baseline Analysis				Optimized Analysis			
	# Nodes	# Edges	# Labels	Time (s)	# Nodes	# Edges	# Labels	Time (s)
Micro	2.5	4.3	6.5	45	2.5	4.3	6.2	45
Concurrency	3.8	10.4	16.5	56	3.8	9.3	14.4	59
Parallelism	4.5	17.5	24.9	79	4.5	15.8	19.2	72
AkkaCrawler	3	6	15	57	3	6	12	55
Batch	5	12	31	85	5	10	21	77
Parallec	8	16	67	190	8	13	46	131
Stone	38	74	173	243	38	58	121	169

Table 1 describes the subjects used in our evaluation. The *Savina* benchmarks consist of 30 diverse programs written purely using actors. *Savina* has three categories: micro benchmarks with 8 well-known actor examples, concurrency benchmarks with 8 classic concurrency problems, and parallel benchmarks with 14 realistic parallel applications. *Savina* has been used in the actor community for various evaluation purposes, such as performance comparison of actor languages/frameworks [21, 12], actor profiling [31], and mapping from message passing concurrency to threads [39]. The original *Savina* does not have a Java *Akka* implementation. We transformed the Scala *Akka* implementation in *Savina* into Java *Akka* and used the transformed version in our experiment because TAP currently supports only Java *Akka*. We had at least two actor programmers double check that the transformed Java version is equivalent to the Scala version.

All four open source projects are written in Java using the Java *Akka* library. Most of their application logic is implemented in actors. *AkkaCrawler* is a parallel web crawler and indexer. *Batch* is a framework for concurrent batch processing. *Parallec* is a scalable asynchronous client, developed by eBay, for firing large numbers of HTTP/SSH/TCP/UDP requests and aggregating responses in parallel. *Stone* is a framework for developing online game servers. From all the actor-based Java *Akka* projects that we can find on Github, *Parallec* and *Stone* are among the largest projects. Some projects mix the Actor model with other concurrency models [36]. We exclude those projects from our evaluation because TAP does not handle other concurrency models such as threads. All our experiments ran on a quad-core machine with 16 GB of RAM, running a 64-bit Ubuntu 14 system.

7.1 Results on MFG Construction

To demonstrate the effectiveness of the optimizations described in Section 4.3, we compare the optimized MFG analysis to the one without optimizations in terms of the size of the MFGs and the time taken for MFG construction. We measure the size of an MFG using the number of nodes, the number of edges and the number of labels on all edges. Overall, 92% of the `onReceive` methods in our subjects match the code pattern for optimizations (i.e., the message handler has a top-level *if* statement that checks for the message type).

MFG Size. Table 2 shows the comparison results. The numbers for the three benchmark categories are averages because there are multiple projects in each category. On average, the optimized analysis reduces the number of edges by 11% and the number of labels by 23%. The number of nodes is not reduced because our analysis creates only one node per

actor class. Recall that our optimizations are safe, indicating that all the reduced edges and labels are false positives. The results show that our optimizations substantially improve the precision of MFG analysis.

The results also show that the optimized analysis reduces a far larger percentage of edges and labels on larger projects. Table 2 highlights (in bold) cases where our optimizations significantly reduces the size of MFGs. For instance, the optimized analysis reduces edges by 19% and labels by 31% for the Parallec project, and reduces edges by 22% and labels by 30% for the Stone project. However, on small subjects such as the micro benchmarks, our optimizations do not produce a significant difference. The reason is that the computed points-to sets in larger projects are typically larger than those computed in smaller projects. Our optimizations often reduces the points-to set to only one element or a much smaller subset in a top-level branch. Therefore, the larger the points-to sets are, the more false positives are reduced. In summary, the optimized analysis has a bigger impact on larger projects.

Analysis Time. We ran the same experiment five times to obtain the average time taken by each analysis on each subject. An interesting observation is that the optimized analysis takes much less time than the baseline analysis does in projects where the optimized analysis reduces the MFG size significantly. For instance, on both Parallec and Stone projects, the analysis time drops about 30% with the optimizations. In other words, the optimized analysis produces more precise results with less time. Our investigation indicates that with smaller points-to sets, the iterative MFG construction algorithm reaches the fixed point faster: having larger points-to sets implies more candidate actors or messages, and this often leads to more iterations for the algorithm to converge. The overhead of our optimizations is negligible, because the optimized analysis performs only a simple structural check on the control flow graph of the `onReceive` method. As shown in the results, the two analyses take similar time on small projects such as the micro benchmarks and the AkkaCrawler project.

7.2 Results on Test Generation

To evaluate the effectiveness of our test generation method, we randomly selected basic blocks in actor classes as targets from all subjects, and for each target, we applied TAP to generate tests to cover it. To avoid biases, we evenly distributed the targets based on the size of actor classes in each project. In practice, the targets may be software patches [25], assertions, and suspicious code locations. In total, we selected 500 targets for the *Savina* benchmarks and 500 targets for the four open source projects. The effectiveness of our method is measured by the percentage of targets covered. A target is covered only when TAP finds a feasible path to the target within the given timeout.

Our problem settings require the specification of receptionists for each actor system. Unfortunately, such information is not specified in our subjects. Therefore, we manually inferred receptionists for each project from its drivers and tests. We set a timeout of ten minutes per target excluding the time for MFG construction. To compare our search techniques, we ran TAP using the following five settings: 1) **Random**, pick a transition randomly from all matched rules on a symbolic configuration; 2) **H1**, enable only the first heuristic; 3) **H2**, enable only the second heuristic; 4) **H1 + H2**, enable both heuristics; 5) **H + F**, enable both heuristics and the feedback-directed technique. All five settings used the depth-first search strategy.

■ **Table 3** The target coverage results of running TAP with five settings.

Subjects	Targets	Random	H1	H2	H1 + H2	H + F
		# Cov (%)	# Cov (%)	# Cov (%)	# Cov (%)	# Cov (%)
Micro	97	52 (54%)	55 (57%)	59 (61%)	76 (78%)	82 (85%)
Concurrency	162	73 (45%)	91 (56%)	79 (49%)	114 (70%)	124 (77%)
Parallelism	241	86 (36%)	103 (43%)	143 (59%)	161 (67%)	173 (72%)
AkkaCrawler	39	21 (54%)	25 (64%)	28 (72%)	34 (87%)	35 (90%)
Batch	60	38 (63%)	43 (72%)	42 (70%)	51 (85%)	55 (92%)
Parallec	178	75 (42%)	81 (46%)	86 (48%)	91 (51%)	139 (78%)
Stone	223	64 (29%)	96 (43%)	107 (48%)	124 (56%)	167 (75%)
Total	1000	409 (41%)	494 (49%)	544 (54%)	651 (65%)	775 (78%)
Avg. time per target (s)		258	217	176	124	91

7.2.1 Target Coverage

Table 3 summarizes the results of running TAP with the five settings. Column 2 shows the number of targets selected for each subject. Columns 3-7 show the number and the percentage of the targets covered by the five settings, respectively. The last row shows the average time (in seconds) taken for covering a target in each setting excluding the time for MFG construction. Overall, the combination of heuristics and feedback-directed technique is effective in covering targets. Search heuristics increase the target coverage from 41% to 65%. The feedback-directed technique further increases the target coverage to 78%.

The Random setting does not work well. It times out in 228 out of 1000 cases. The major problem with Random is that it often introduces many unnecessary actors to path exploration. Introducing a new actor in a test is an expensive operation, because it opens up additional search space for TAP to find a feasible execution trace on the new actor. As a result, Random wastes lots of resources exploring traces for unnecessary actors, and takes longer time to cover a target. In addition, the tests generated by Random are typically larger in terms of the number of actors. The H1 setting suffers the same problem. However, it reduces the search space by sequentializing the execution of message handlers. As a result, the number of enabled transitions on each symbolic configuration in H1 is much smaller than that in Random. Due to the space reduction, H1 improves the target coverage to 49%.

The H2 setting improves Random by keeping the tests as small as possible to avoid exploring unnecessary space. Our experiment results show that in many cases, the target can be reached with no more than three actors. For example, many subjects use the master-worker pattern to implement parallelism. The workers proceed in parallel, and do not interact with each other. In such cases, it suffices to cover any target in the worker with only two actors: one master and one worker. Creating new workers only adds complexity to the problem. H2 is very efficient in covering such targets because it assigns a very low weight to transitions that introduce new actors.

The feedback-directed technique is particularly useful when our heuristics do not work well and BSE frequently hits infeasible paths. In our experiment, we find that there are a number of cases where covering the target requires creating multiple actors of the same class (e.g., comparing the IDs of actors). In these cases, the heuristics work poorly because they prefer to reuse the existing actor rather than create a new actor of the same class. As a result, the heuristics keep hitting infeasible paths in these cases. The feedback-directed technique is quite effective in guiding BSE to find a feasible path. For instance, in the case of checking for different IDs, it directly identifies that the ID field of the actor needs to be


```

1 public void onReceive (Object message) {
2     if (message instanceof TokenMessage) {
3         TokenMessage token = (TokenMessage)message;
4         if(token.hasNext()) {
5             // bug: potential null de-reference on nextActor
6             this.nextActor.tell(token.next(), getSelf());
7         } ...
8     } else if (message instanceof DataMessage) {
9         this.nextActor = (ActorRef) ((DataMessage) message).data;
10    } ...
11 }

```

■ **Figure 12** A bug caused by out-of-order message delivery in the ThreadRing benchmark.

changed, because the unsatisfiable core contains variables that map to this field. Since the only way to change the ID field of the actor is through its constructor, the feedback-directed technique prioritizes the transitions that introduce new actors to be explored first, and thus quickly finds a feasible path.

We analyze the cases in which TAP fails to cover the targets in the H + F setting. More than half of the cases are due to a lack of environment modeling (e.g., access to database and network). Such issues can be mitigated by adding models for calls to the environment. The rest of the cases are mainly due to timeouts for the exploration and complex constraints that Z3 fails to solve.

7.2.2 Bug Detection

By running TAP to cover these 1,000 targets, we are able to find six distinct bugs in our subjects. All six bugs are found in the *Savina* benchmarks in three projects. Five out of the six bugs are crash bugs. One bug is less critical: a non-crash warning from *Akka* regarding messages sent to actors that have been killed. We have confirmed that all bugs are triggered in both the original benchmarks and the transformed versions with our generated tests. We diagnose the six bugs and find that all five crash bugs are caused by out-of-order message delivery. Such bugs are hard to reveal locally because out-of-order message delivery is unlikely to happen locally. The other bug is caused by sending two stop messages to kill an actor, and the recipient actor kills itself after receiving the first stop message.

Figure 12 shows one crash bug found in the ThreadRing benchmark. There is a potential null de-reference on the `nextActor` field at Line 6. The ThreadRing system starts with a coordinator sending a `DataMessage` to each token passer to inform them the next passer and form a ring among them. The coordinator then sends a token to one passer in the ring, and then the token is passed from one passer to another in the ring. The passer sets its `nextActor` field at Line 9 upon receiving a `DataMessage` and sends the token at Line 6 upon receiving a `TokenMessage`. The assumption is that every passer must set the `nextActor` before sending the token (i.e., receive the `DataMessage` before the `TokenMessage`). Since the *Akka* framework guarantees FIFO message delivery, this assumption holds for the first passer. However, the assumption may not hold for the other passers. It is possible that the second passer receives the `TokenMessage` from the first passer before receiving the `DataMessage` from the coordinator. Although the `DataMessage` is sent before the `TokenMessage`, the two messages are sent by different senders, and may be delivered out of order. In this case, a null pointer exception is thrown in the second passer. TAP found this bug because the exceptional branch of Line 6 (WALA IR contains exceptional branches for potential null dereferences) happened to be chosen as a target. A simple fix to this bug is adding a null check on `nextActor` before passing the token.

8 Related Work

Testing Actors. The most related work on testing Actor systems is dCUTE [33]. dCUTE differs from TAP in three aspects. First, dCUTE’s goal is to achieve overall coverage while TAP aims at covering target code locations. They can be used to complement each other. Second, dCUTE performs *forward* concolic execution while TAP does *backwards* symbolic execution without a side-by-side concrete execution. Lastly, dCUTE handles only a subset of actor operations. For example, it assumes that all actors have been created before execution, and thus does not handle dynamic actor creation. However, we provide a rigorous definition of the semantics of all actor operations in BSE.

BASSET [22] leverages a model checker to systematically explore message schedules in an actor system. BASSET assumes that input messages are given, and aims at exploring as many message schedules as possible on the given input. It uses state merging and dynamic partial order reduction (DPOR) to reduce the search space of message schedules. BITA [38] also explores possible message schedules for given input messages. It defines new *schedule coverage* criteria, and uses these criteria to guide the exploration to expose bugs. TRANSDPOR [37] proposes another DPOR technique that exploits the transitivity of the dependency relations between actors for schedule space reduction. TAP not only explores message schedules, but also generates message contents. These exploration techniques and space-reduction techniques can be integrated into TAP for more efficient test generation.

Targeted Test Generation. A number of targeted test generation techniques have been developed on sequential programs using both forward symbolic execution [24, 18, 16, 25, 10] and backward symbolic execution [15, 29, 13]. However, they cannot be directly applied to actor systems. Since an actor library often contains complex multi-threading and networking code, direct exploration of these actor library methods is impractical and the execution often fails to go across actors. Our work fills this gap by defining formal semantic models of actor operations in our analysis, and thus preventing our analysis from exploring the actor library.

Feedback-Directed Test Generation. Previous research has proposed using information from previous executions as feedback to guide test generation. RANDOOP [30] uses execution feedback from previous tests to avoid generating redundant and illegal inputs. Garg et al. [17] use the unsatisfiable cores from previous infeasible paths to generalize the reason for the infeasibility, and thus rule out more infeasible paths. We also use the unsatisfiable cores from infeasible paths, but we use them to guide BSE to efficiently find a feasible path.

Backward Symbolic Analysis. SNUGGLEBUG [11] uses backward symbolic *analysis* for computing inter-procedural weakest preconditions. The symbolic reasoning in their work is similar to ours except that their analysis works on all possible program paths to the target while our BSE aims at finding one feasible path.

Static Analysis of Actors. There has been previous work [28] on static analysis of actor programs to infer the ownership transfer of messages. This analysis works on individual actors (i.e., intra-actor), and does not model interactions between actors. Our MFG construction is a more complex whole-system analysis that requires modeling actor interactions.

9 Conclusion

We have presented a method for targeted test generation for actor systems based on BSE. Our method first constructs an MFG to capture the potential interactions between actors. Guided by the MFG, it starts BSE directly from the target to find a feasible path to the entry point of the actor system. We have provided high-level models for all actor operations and formally defined their semantics in our analysis to avoid analyzing the complex code in the actor library. To efficiently navigate the huge search space in BSE, we have proposed two heuristics and a feedback-directed search technique. We have implemented our method in TAP, and evaluated it on *Savina* and four open source projects. The evaluation results have shown that TAP is effective in targeted test generation for actor systems.

In the future, we plan to further improve our search techniques in BSE. One direction is to reduce the state space of message schedules using partial order reduction. The happens-before relation used in previous work is fairly coarse-grained. We plan to define a finer-grained partial order relation based on program analysis to further reduce the search space. Another direction is to leverage dynamic traces from existing tests to guide our explorations.

References

- 1 *Erlang Introduction*. <http://erlang.org/faq/introduction.html>.
- 2 *Java Akka*. <https://doc.akka.io/docs/akka/current/actors.html?language=java>.
- 3 *Orleans*. <https://dotnet.github.io/orleans/index.html>.
- 4 *Scala Akka*. <https://doc.akka.io/docs/akka/current/index-actors.html?language=scala>.
- 5 *Wala*. <http://wala.sourceforge.net>.
- 6 Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, USA, 1986.
- 7 Gul Agha. Concurrent Object-oriented Programming. *Commun. ACM*, 33(9):125–141, 1990.
- 8 Gul A Agha, Ian A Mason, Scott F Smith, and Carolyn L Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7(1):1–72, 1997.
- 9 Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.
- 10 Domagoj Babić, Lorenzo Martignoni, Stephen McCamant, and Dawn Song. Statically-directed dynamic automated test generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pages 12–22. ACM, 2011.
- 11 Satish Chandra, Stephen J Fink, and Manu Sridharan. Snugglebug: a powerful approach to weakest preconditions. *ACM Sigplan Notices*, 44(6):363–374, 2009.
- 12 Dominik Charousset, Raphael Hiesgen, and Thomas C Schmidt. Caf-the C++ actor framework for scalable and resource-efficient applications. In *Proceedings of the 4th International Workshop on Programming based on Actors Agents and Decentralized Control*, pages 15–28. ACM, 2014.
- 13 Florence Charretre and Arnaud Gotlieb. Constraint-based test input generation for java bytecode. In *Software Reliability Engineering (ISSRE), 2010 IEEE 21st International Symposium on*, pages 131–140. IEEE, 2010.
- 14 Leonardo 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 Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008.
- 15 Peter Dinges and Gul Agha. Targeted test input generation using symbolic-concrete backward execution. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 31–36. ACM, 2014.

- 16 Josselin Feist, Laurent Mounier, and Marie-Laure Potet. Guided dynamic symbolic execution using subgraph control-flow information. In *International Conference on Software Engineering and Formal Methods*, pages 76–81. Springer, 2016.
- 17 Pranav Garg, Franjo Ivancic, Gogul Balakrishnan, Naoto Maeda, and Aarti Gupta. Feedback-directed unit test generation for C/C++ using concolic execution. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 132–141. IEEE Press, 2013.
- 18 Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. Dowsing for overflows: A guided fuzzer to find buffer boundary violations. In *USENIX Security Symposium*, pages 49–64, 2013.
- 19 Carl Hewitt. Viewing control structures as patterns of passing messages. *Artificial intelligence*, 8(3):323–364, 1977.
- 20 Atsushi Igarashi, Benjamin C Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(3):396–450, 2001.
- 21 Shams Imam and Vivek Sarkar. Savina-an actor benchmark suite. In *4th International Workshop on Programming based on Actors, Agents, and Decentralized Control, AGERE*, 2014.
- 22 Steven Lauterburg, Mirco Dotta, Darko Marinov, and Gul Agha. A Framework for State-Space Exploration of Java-Based Actor Programs. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering, ASE '09*, pages 468–479, Washington, DC, USA, 2009.
- 23 Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *ACM Sigplan Notices*, volume 43, pages 329–339. ACM, 2008.
- 24 Kin-Keung Ma, Khoo Yit Phang, Jeffrey Foster, and Michael Hicks. Directed symbolic execution. *Static Analysis*, pages 95–111, 2011.
- 25 Paul Dan Marinescu and Cristian Cadar. Katch: high-coverage testing of software patches. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 235–245. ACM, 2013.
- 26 Matthew Might, Yannis Smaragdakis, and David Van Horn. Resolving and exploiting the k-CFA paradox: illuminating functional vs. object-oriented program analysis. In *ACM Sigplan Notices*, volume 45, pages 305–315. ACM, 2010.
- 27 Ana Milanova, Atanas Rountev, and Barbara G Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 14(1):1–41, 2005.
- 28 Stas Negara, Rajesh K Karmani, and Gul Agha. Inferring ownership transfer for efficient message passing. In *ACM SIGPLAN Notices*, volume 46, pages 81–90. ACM, 2011.
- 29 Oswaldo Olivo, Isil Dillig, and Calvin Lin. Detecting and exploiting second order denial-of-service vulnerabilities in web applications. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 616–628. ACM, 2015.
- 30 Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-Directed Random Test Generation. In *Proceedings of the 29th International Conference on Software Engineering, ICSE '07*, pages 75–84, Washington, DC, USA, 2007.
- 31 Andrea Rosà, Lydia Y Chen, and Walter Binder. Profiling actor utilization and communication in Akka. In *Proceedings of the 15th International Workshop on Erlang*, pages 24–32. ACM, 2016.
- 32 Jan Schäfer and Arnd Poetzsch-Heffter. JCoBox: Generalizing Active Objects to Concurrent Components. In Theo D’Hondt, editor, *ECOOP 2010 – Object-Oriented Programming*, pages 275–299, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

- 33 Koushik Sen and Gul Agha. Automated systematic testing of open distributed programs. In *International Conference on Fundamental Approaches to Software Engineering*, pages 339–356. Springer, 2006.
- 34 Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a Concolic Unit Testing Engine for C. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ESEC/FSE-13*, pages 263–272, New York, NY, USA, 2005.
- 35 Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. Pick your contexts well: understanding object-sensitivity. In *ACM SIGPLAN Notices*, volume 46, pages 17–30. ACM, 2011.
- 36 Samira Tasharofi, Peter Dinges, and Ralph E Johnson. Why do scala developers mix the actor model with other concurrency models? In *European Conference on Object-Oriented Programming*, pages 302–326. Springer, 2013.
- 37 Samira Tasharofi, Rajesh K Karmani, Steven Lauterburg, Axel Legay, Darko Marinov, and Gul Agha. TransDPOR: A novel dynamic partial-order reduction technique for testing actor programs. In *Formal Techniques for Distributed Systems*, pages 219–234. Springer, 2012.
- 38 Samira Tasharofi, Michael Pradel, Yu Lin, and Ralph Johnson. Bita: Coverage-guided, automatic testing of actor programs. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 114–124. IEEE, 2013.
- 39 Ganesha Upadhyaya and Hridesh Rajan. Effectively mapping linguistic abstractions for message-passing concurrency to threads on the Java virtual machine. *ACM SIGPLAN Notices*, 50(10):840–859, 2015.
- 40 Carlos Varela and Gul Agha. Programming dynamically reconfigurable open systems with salsa. *SIGPLAN Not.*, 36(12):20–34, 2001.

A Semantics of Local Computations in MFG Analysis

Figure 13 shows the transition rules for local computations in the MFG analysis. Since local computations concern only the actor state ς in ω , we omit other states in ω in our transition rules for better readability (the other states are the same on both sides of the rules). The operator \sqcup is used to merge two maps by merging the values of the same key in both maps. The *dispatch* function takes as input an object and a method name, and returns the dispatched method³. The transition rules describe precisely the 1-object-sensitive points-to analysis [27]. The *Object Allocation* rule says that the heap context of an object is the label of its allocation site. The *Method Invocation* rule describes the context sensitivity. The rule says that the context used for analyzing a method is the heap context of the receiver object, which is the label of its allocation site.

B Semantics Of Local Computations In BSE

Figure 14 and Figure 15 show respectively the semantics of intra-procedural BSE and the semantics of inter-procedural BSE for local computations in an actor. Since local computations concern only the local state β and the path condition ϕ , we omit other states in the symbolic configuration in our transition rules for better readability. Note that *subType* is a predicate in FOL to check the sub-type relation, and *type* and *field* are functions in FOL.

³ Our language does not support method overloading, and thus a method can be dispatched based on the given object and its method name

Variable Reference

$$(\llbracket v = v'; \ell \rrbracket, st, \sigma, _) \Rightarrow_A (succ(\ell), st, \sigma', _), \text{ where } \sigma' = \sigma \sqcup [st(v) \mapsto \sigma(st(v'))]$$
Field Reference

$$(\llbracket v = v'.f; \ell \rrbracket, st, \sigma, _) \Rightarrow_A (succ(\ell), st, \sigma', _), \text{ where}$$

$$(_, [f \mapsto a_f]) \in \sigma(st(v')) \quad \sigma' = \sigma \sqcup [st(v) \mapsto \sigma(a_f)]$$
Object Allocation

$$(\llbracket v = \text{new } C(\vec{v}'); \ell \rrbracket, st, \sigma, _) \Rightarrow_A (succ(\ell), st, \sigma', _), \text{ where}$$

$$hc = \ell \quad \vec{f} = \mathcal{F}(C) \quad a_i = (f_i, hc) \quad o = (hc, [f_i \mapsto a_i])$$

$$\sigma' = \sigma \sqcup [st(v) \mapsto \{o\}, a_i \mapsto \sigma(st(v'_i))]$$
Method Invocation

$$(\llbracket v = v_0.m(\vec{v}'); \ell \rrbracket, st, \sigma, cs, c) \Rightarrow_A (s, st', \sigma', cs', c'), \text{ where}$$

$$M = \llbracket C \ m(\vec{C}' \ v'') \ \{ \vec{C}'' \ v'''; \vec{s}' \} \rrbracket = \text{dispatch}(o_0, m) \quad o_0 \in \sigma(st(v_0)) \quad (hc_0, _) = o_0$$

$$c' = hc_0 \quad a_i = (v''_i, m, c') \quad a'_i = (v'''_i, m, c') \quad st' = \text{cons}([v''_i \mapsto a_i, v'''_i \mapsto a'_i], st)$$

$$s = \text{car}(\vec{s}') \quad \sigma' = \sigma \sqcup [a_i \mapsto \sigma(st(v'_i))] \quad cs' = \text{cons}((succ(\ell), c, st(v)), cs)$$
Return

$$(\llbracket \text{return } v; \ell \rrbracket, st, \sigma, cs, c) \Rightarrow_A (s, \text{cdr}(st), \sigma', \text{cdr}(cs), c'), \text{ where}$$

$$(s, c', a_{ret}) = \text{car}(cs) \quad \sigma' = \sigma \sqcup [a_{ret} \mapsto \sigma(st(v))]$$
Casting

$$(\llbracket v = (C) v'; \ell \rrbracket, st, \sigma, _) \Rightarrow_A (succ(\ell), st, \sigma', _), \text{ where } \sigma' = \sigma \sqcup [st(v) \mapsto \sigma(st(v'))]$$

■ **Figure 13** Abstract semantics for local computations in MFG analysis.

We also use a number of helper functions in our transition rules. The function *pred* returns the previous statement of a given label, and the function *last* returns the last element of a given list. The function *method* returns the method that encloses the statement with the given label. The function *callee* takes as input the label of a call site *s*, and returns the set of all possible callees. Specifically, it retrieves the signature *sig* of the called method from *s*, locates the enclosing method *M* of *s* in the call graph, and returns the set of all callees of *M* that match *sig*. The function *callsites* returns the set of all possible call sites of a given method.

In what follows, we explain the inter-procedural rules, which are more interesting. We assume that our language uses the *call-by-value* evaluation strategy. To perform inter-procedural BSE, a context-insensitive call graph is used to guide the execution. The entry point of the call graph is the message handler of the actor. As we execute a method *m* backwards, there are two possible cases regarding the target: 1) the target is outside *m*, indicating that BSE has previously reached the call site of *m* and has jumped from that call site to *m*, and the current call stack must be not empty; 2) the target is inside *m*, indicating that BSE starts from *m* and the current call stack must be empty. The first three rules in Figure 15 apply to the first case. The *Method Invocation* rule says that upon a method invocation, BSE queries the call graph for all possible callees of the invocation, jumps to the last statement of a possible callee, and adds the constraint that every parameter must

Variable Reference

$$(\llbracket v = v'; \ell \rrbracket, _) \Rightarrow_S ((pred(\ell), _), \phi[\hat{v}'/\hat{v}])$$

Binary Expression

$$(\llbracket v = v' \text{ op } v''; \ell \rrbracket, _) \Rightarrow_S ((pred(\ell), _), \phi'), \text{ where } \phi' = \phi[(\hat{v}' \text{ op } \hat{v}'')/\hat{v}]$$

Field Reference

$$(\llbracket v = v'.f; \ell \rrbracket, _) \Rightarrow_S ((pred(\ell), _), \phi[read(\hat{v}', f)/\hat{v}])$$

Field Update

$$(\llbracket v.f = v'; \ell \rrbracket, _) \Rightarrow_S ((pred(\ell), _), \phi[update(f, v, v')/f])$$

Casting

$$(\llbracket v = (C) v'; \ell \rrbracket, _) \Rightarrow_S ((pred(\ell), _), \phi[\hat{v}'/\hat{v}] \wedge subType(type(\hat{v}), C))$$

Object Allocation

$$(\llbracket v = \text{new } C(\vec{v}'); \ell \rrbracket, _) \Rightarrow_S ((pred(\ell), _), \phi'), \text{ where}$$

$$\hat{v}'' \text{ is fresh} \quad \vec{f} = \mathcal{F}(C) \quad \phi' = \phi[\hat{v}''/\hat{v}, update(f_i, v'', v'_i)/f_i] \wedge type(\hat{v}'') == C$$

If-True

$$(\llbracket \text{if } (e) \vec{s} \text{ else } \vec{s}'; \ell \rrbracket, _) \Rightarrow_S ((last(\vec{s}), _), \phi \wedge \hat{e})$$

If-False

$$(\llbracket \text{if } (e) \vec{s} \text{ else } \vec{s}'; \ell \rrbracket, _) \Rightarrow_S ((last(\vec{s}'), _), \phi \wedge \neg \hat{e})$$

■ **Figure 14** Transition rules for intra-procedural backward symbolic execution.

Method Invocation

$$(\llbracket v = v'.m(\vec{v}''); \ell \rrbracket, cs, \hat{v}_0), \phi \Rightarrow_S ((s, cs', \hat{v}'), \phi'), \text{ where}$$

$$M = \llbracket C \ m(\overline{C'} \ v''') \ \{\vec{s}'\} \rrbracket \quad M \in callees(\ell) \quad s = last(\vec{s}')$$

$$cs' = cons(pred(\ell), \hat{v}, \hat{v}'), cs \quad \phi' = \phi \wedge \hat{v}_i''' == \hat{v}_i''$$

Return-CallStack Not Empty

$$(\llbracket \text{return } v; \ell \rrbracket, cs, _) \Rightarrow_S ((pred(\ell), cs, _), \phi[\hat{v}/\hat{v}']), \text{ where } (_, \hat{v}', _) = car(cs)$$

Method Entry-CallStack Not Empty

$$(\llbracket \text{entry}; \ell \rrbracket, cs, \hat{v}_0), \phi \Rightarrow_S (s, cdr(cs), \hat{v}_0'), \phi), \text{ where } (s, _, \hat{v}_0') = car(cs)$$

Return-CallStack Empty

$$(\llbracket \text{return } v; \ell \rrbracket, [], _) \Rightarrow_S ((pred(\ell), [], _), \phi)$$

Method Entry-CallStack Empty

$$(\llbracket \text{entry}; \ell \rrbracket, [], \hat{v}_0), \phi \Rightarrow_S ((pred(\ell'), [], \hat{v}'), \phi'), \text{ where}$$

$$s = \llbracket v = v'.m(\vec{v}''); \ell' \rrbracket \quad \ell' \in callsites(M)$$

$$M = \llbracket C \ m'(\overline{C'} \ v''') \ \{\vec{s}'\} \rrbracket = method(\ell') \quad \phi' = \phi \wedge \hat{v}_i''' == \hat{v}_i''$$

■ **Figure 15** Transition rules for inter-procedural backward symbolic execution.

be equal to its corresponding argument of the callee (call-by-value). The *Return-CallStack Not Empty* rule says that the variable to which the return value is assigned at the call site is replaced with the return value in the path constraint. The *Method Entry-CallStack Not Empty* rule says that the execution returns to the call site, and the top frame is popped from the call stack. The last two rules in Figure 15 apply to the second case. The *Return-CallStack Empty* rule does not update the path constraint, because the caller is unknown at this point, so is the variable that would hold the return value. The *Method Entry-CallStack Empty* says that BSE queries the call graph for all possible callers of the current method, jumps back to a possible call site, and adds the constraint that every argument of the callee are equal to its corresponding parameter in the call site. Note that no constraint over the variable v that holds the return value is added to the path constraint, because once the execution returns to the call site, it moves backwards and will never use the variable v . The constraints over v do not affect covering the target, and thus need not be added.

Typed First-Class Traits

Xuan Bi

The University of Hong Kong, Hong Kong, China
xbi@cs.hku.hk

Bruno C. d. S. Oliveira

The University of Hong Kong, Hong Kong, China
bruno@cs.hku.hk

Abstract

Many dynamically-typed languages (including JavaScript, Ruby, Python or Racket) support *first-class classes*, or related concepts such as first-class traits and/or mixins. In those languages classes are first-class values and, like any other values, they can be passed as an argument, or returned from a function. Furthermore first-class classes support *dynamic inheritance*: i.e. they can inherit from other classes at *runtime*, enabling programmers to abstract over the inheritance hierarchy. In contrast, type system limitations prevent most statically-typed languages from having first-class classes and dynamic inheritance.

This paper shows the design of SEDEL: a polymorphic statically-typed language with *first-class traits*, supporting *dynamic inheritance* as well as conventional OO features such as *dynamic dispatching* and *abstract methods*. To address the challenges of type-checking first-class traits, SEDEL employs a type system based on the recent work on *disjoint intersection types* and *disjoint polymorphism*. The novelty of SEDEL over core disjoint intersection calculi are *source level* features for practical OO programming, including first-class traits with dynamic inheritance, dynamic dispatching and abstract methods. Inspired by Cook and Palsberg's work on the denotational semantics for inheritance, we show how to design a source language that can be elaborated into Alpuim et al.'s F_i (a core polymorphic calculus with records supporting disjoint polymorphism). We illustrate the applicability of SEDEL with several example uses for first-class traits, and a case study that modularizes programming language interpreters using a highly modular form of visitors.

2012 ACM Subject Classification Software and its engineering → Object oriented languages

Keywords and phrases traits, extensible designs

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2018.9

Supplement Material ECOOP Artifact Evaluation approved artifact available at <http://dx.doi.org/10.4230/DARTS.4.3.9>

Funding Hong Kong Research Grant Council projects number 17210617 and 17258816

Acknowledgements We thank the anonymous reviewers for their helpful comments.

1 Introduction

Many dynamically-typed languages (including JavaScript [1], Ruby [4], Python [2] or Racket [3]) support *first-class classes* [26], or related concepts such as first-class mixins and/or traits. In those languages classes are first-class values and, like any other values, they can be passed as an argument, or returned from a function. Furthermore first-class classes support *dynamic inheritance*: i.e., they can inherit from other classes at *runtime*, enabling



© Xuan Bi and Bruno C. d. S. Oliveira;
licensed under Creative Commons License CC-BY
32nd European Conference on Object-Oriented Programming (ECOOP 2018).
Editor: Todd Millstein; Article No. 9; pp. 9:1–9:28



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



programmers to abstract over the inheritance hierarchy. Those features make first-class classes very powerful and expressive, and enable highly modular and reusable pieces of code, such as:

```
const mixin = Base => { return class extends Base { ... } };
```

In this piece of JavaScript code, `mixin` is parameterized by a class `Base`. Note that the concrete implementation of `Base` can be even dynamically determined at runtime, for example after reading a configuration file to decide which class to use as the base class. When applied to an argument, `mixin` will create a new class on-the-fly and return that as a result. Later that class can be instantiated and used to create new objects, as any other classes.

In contrast, most statically-typed languages do not have first-class classes and dynamic inheritance. While all statically-typed OO languages allow first-class *objects* (i.e. objects can be passed as arguments and returned as results), the same is not true for classes. Classes in languages such as Scala, Java or C++ are typically a second-class construct, and the inheritance hierarchy is *statically determined*. The closest thing to first-class classes in languages like Java or Scala are classes such as `java.lang.Class` that enable representing classes and interfaces as part of their reflective framework. `java.lang.Class` can be used to mimic some of the uses of first-class classes, but in an essentially dynamically-typed way. Furthermore simulating first-class classes using such mechanisms is highly cumbersome because classes need to be manipulated programmatically. For example instantiating a new class cannot be done using the standard `new` construct, but rather requires going through API methods of `java.lang.Class`, such as `newInstance`, for creating a new instance of a class.

Despite the popularity and expressive power of first-class classes in dynamically-typed languages, there is surprisingly little work on typing of first-class classes (or related concepts such as first-class mixins or traits). First-class classes and dynamic inheritance pose well-known difficulties in terms of typing. For example, in his thesis, Bracha [15] comments several times on the difficulties of typing dynamic inheritance and first-class mixins, and proposes the restriction to static inheritance that is also common in statically-typed languages. He also observes that such restriction poses severe limitations in terms of expressiveness, but that appeared (at the time) to be a necessary compromise when typing was also desired. Only recently some progress has been made in statically typing first-class classes and dynamic inheritance. In particular there are two works in this area: Racket's gradually typed first-class classes [51]; and Lee et al.'s model of typed first-class classes [30]. Both works provide typed models of first-class classes, and they enable encodings of mixins [16] similar to those employed in dynamically-typed languages.

However, as far as we known no previous work supports statically-typed *first-class traits*. Traits [47] are an alternative to mixins, and other models of (multiple) inheritance. The key difference between traits and mixins lies on the treatment of conflicts when composing multiple traits/mixins. Mixins adopt an *implicit* resolution strategy for conflicts, where the compiler automatically picks one implementation in case of conflicts. For example, Scala uses the order of mixin composition to determine which implementation to pick in case of conflicts. Traits, on the other hand, employ an *explicit* resolution strategy, where the compositions with conflicts are rejected, and the conflicts are explicitly resolved by programmers.

Schärli et al. [47] make a good case for the advantages of the trait model. In particular, traits avoid bugs that could arise from accidental conflicts that were not detected by programmers. With the mixin model, such conflicts would be silently resolved, possibly resulting in unexpected runtime behaviour due to a wrong method implementation choice. In a setting with dynamic inheritance and first-class classes this problem is exacerbated by not knowing all components being composed statically, greatly increasing the possibility of accidental conflicts. From a modularity point-of-view, the trait model also ensures that

composition is *commutative*, thus the order of composition is irrelevant and does not affect the semantics. Bracha [15] claims that “*The only modular solution is to treat the name collisions as errors...*”, strengthening the case for the use of a trait model of composition. Otherwise, if the semantics is affected by the order of composition, global knowledge about the full inheritance graph is required to determine which implementations are chosen. Schärli et al. discuss several other issues with mixins, which can be improved by traits. We refer to their paper for further details.

This paper presents the design of SEDEL: a polymorphic statically-typed (pure) language with *first-class traits*, supporting *dynamic inheritance* as well as conventional OO features such as *dynamic dispatching* and *abstract methods*. Traits pose additional challenges when compared to models with first-class classes or mixins, because method conflicts should be detected *statically*, even in the presence of features such as dynamic inheritance and composition and *parametric polymorphism*. To address the challenges of typing first-class traits and detecting conflicts statically, SEDEL adopts a polymorphic structural type system based on *disjoint polymorphism* [7]. The choice of structural typing is due to its simplicity, but we think similar ideas should also work in a nominal type system.

The main contribution of this paper is to show how to model source language constructs for first-class traits and dynamic inheritance, supporting standard OO features such as *dynamic dispatching* and *abstract methods*. Previous work on disjoint intersection types is aimed at core record calculi, and omits important features for practical OO languages, including (dynamic) inheritance, dynamic dispatching and abstract methods. Based on Cook and Palsberg’s work on the denotational semantics for inheritance [19], we show how to design a source language that can be elaborated into Alpuim et al.’s F_i [7], a polymorphic calculus with records supporting disjoint polymorphism. SEDEL’s elaboration into F_i is proved to be both type-safe and coherent. Coherence ensures that the semantics of SEDEL is unambiguous. In particular this property is useful to ensure that programs using traits are free of conflicts/ambiguities (even when the types of the object parts being composed are not fully statically known).

We illustrate the applicability of SEDEL with several example uses for first-class traits. Furthermore we conduct a case study that modularizes programming language interpreters using a highly modular form of Object Algebras [39] and VISITORS. In particular we show how SEDEL can easily compose multiple object algebras into a single object algebra. Such composition operation has previously been shown to be highly challenging in languages like Java or Scala [41, 44]. The previous state-of-the-art implementations for such operation require employing type-unsafe reflective techniques to simulate the features of first-class classes. Moreover conflicts are not statically detected. In contrast the approach in this paper is fully type-safe, convenient to use and conflicts are statically detected.

In summary the contributions of this paper are:

- **Typed first-class traits:** We present SEDEL: a statically-typed language design that supports first-class traits, dynamic inheritance, as well as standard high-level OO constructs such as dynamic dispatching and abstract methods.
- **Elaboration of first-class traits into disjoint intersection types/polymorphism:** We show how the semantics of SEDEL can be defined by elaboration into Alpuim et al.’s F_i [7]. The elaboration is inspired by the work of Cook and Palsberg [19] to model inheritance.
- **Implementation and modularization case study:** SEDEL is implemented and available.¹ To evaluate SEDEL we conduct a case study. The case study shows that support

¹ The implementation, case study code and appendix are available at <https://goo.gl/uFrWkr>.

for composition of Object Algebras and VISITORS is greatly improved in SEDEL. Using such improved design patterns we re-code the interpreters in Cook's undergraduate Programming Languages book [18] in a modular way in SEDEL.

2 Overview

This section aims at introducing first-class classes and traits, their possible uses and applications, as well as the typing challenges that arise from their use. We start by describing a hypothetical JavaScript library for text editing widgets, inspired and adapted from Racket's GUI toolkit [51]. The example is illustrative of typical uses of dynamic inheritance/composition, and also the typing challenges in the presence of first-class classes/traits. Without diving into technical details, we then give the corresponding typed version in SEDEL, and informally presents its salient features.

2.1 First-Class Classes in JavaScript

A class construct was officially added to JavaScript in the ECMAScript 2015 Language Specification [23]. One purpose of adding classes to JavaScript was to support a construct that is more familiar to programmers who come from mainstream class-based languages, such as Java or C++. However classes in JavaScript are *first-class* and support functionality not easily mimicked in statically-typed class-based languages.

Conventional Classes. Before diving into the more advanced features of JavaScript classes, we first review the more conventional class declarations supported in JavaScript as well as many other languages. Even for conventional classes there are some interesting points to note about JavaScript that will be important when we move into a typed setting. An example of a JavaScript class declaration is:

```
class Editor {
  onKey(key) { return "Pressing " + key; }
  doCut()    { return this.onKey("C-x") + " for cutting text"; }
  showHelp() { return "Version: " + this.version() + " Basic usage..."; }
};
```

This form of class definition is standard and very similar to declarations in class-based languages (for example Java). The `Editor` class defines three methods: `onKey` for handling key events, `doCut` for cutting text and `showHelp` for displaying help message. For the purpose of demonstration, we elide the actual implementation, and replace it with plain messages.

We wish to bring the readers' attention to two points in the above class. Firstly, note that the `doCut` method is defined in terms of the `onKey` method via the keyword `this`. In other words the call to `onKey` is enabled by the *self* reference and is *dynamically dispatched* (i.e., the particular implementation of `onKey` will only be determined when the class or subclass is instantiated). Secondly, notice that there is no definition of the `version` method in the class body, but such method is used inside the `showHelp` method. In a untyped language, such as JavaScript, using undefined methods is error prone – accidentally instantiating `Editor` and then calling `showHelp` will cause a runtime error! Statically-typed languages usually provide some means to protect us from this situation. For example, in Java, we would need an *abstract version* method, which effectively makes `Editor` an abstract class and prevents it from being instantiated. As we will see, SEDEL's treatment of abstract methods is quite different from mainstream languages. In fact, SEDEL has a unified (typing) mechanism for dealing with both dynamic dispatch and abstract methods. We will describe SEDEL's mechanism for dealing with both features and justify our design in Section 3.

First-Class Classes and Class Expressions. Another way to define a class in JavaScript is via a *class expression*. This is where the class model in JavaScript is very different from the traditional class model found in many mainstream OO languages, such as Java, where classes are second-class (static) entities. JavaScript embraces a dynamic class model that treats classes as *first-class* expressions: a function can take classes as arguments, or return them as a result. First-class classes enable programmers to abstract over patterns in the class hierarchy and to experiment with new forms of OOP such as mixins and traits. In particular, mixins become programmer-defined constructs. We illustrate this by presenting a simple mixin that adds spell checking to an editor:

```
const spellMixin = Base => {
  return class extends Base {
    check() { return super.onKey("C-c") + " for spell checking"; }
    onKey(key) { return "Process " + key + " on spell editor"; }
  }
};
```

In JavaScript, a mixin is simply a function with a superclass as input and a subclass extending that superclass as an output. Concretely, `spellMixin` adds a method `check` for spell checking. It also provides a method `onKey`. The function `spellMixin` shows the typical use of what we call *dynamic inheritance*. Note that `Base`, which is supposed to be a superclass being inherited, is *parameterized*. Therefore `spellMixin` can be applied to any base class at *runtime*. This is impossible to do, in a type-safe way, in conventional statically-typed class-based languages like Java or C++.²

It is noteworthy that not all applications of `spellMixin` to base classes are successful. Notice the use of the `super` keyword in the `check` method. If the base class does not implement the `onKey` method, then mixin application fails with a runtime error. In a typed setting, a type system must express this requirement (i.e., the presence of the `onKey` method) on the (statically unknown) base class that is being inherited.

We invite the readers to pause for a while and think about what the type of `spellMixin` would look like. Clearly our type system should be flexible enough to express this kind of dynamic pattern of composition in order to accommodate mixins (or traits), but also not too lenient to allow any composition.

Mixin Composition and Conflicts. The real power of mixins is that `spellMixin`'s functionality is not tied to a particular class hierarchy and is composable with other features. For example, we can define another mixin that adds simple modal editing – as in Vim – to an arbitrary editor:

```
const modalMixin = Base => {
  return class extends Base {
    constructor() {
      super();
      this.mode = "command";
    }
    toggleMode() { return "toggle succeeded"; }
    onKey(key) { return "Process " + key + " on modal editor"; }
  }
};
```

² With C++ templates, it is possible to implement a so-called mixin pattern [49], which enables extending a parameterized class. However C++ templates defer type-checking until instantiation, and such pattern still does not allow selection of the base class at runtime (only at up to class instantiation time).

9:6 Typed First-Class Traits

`modalMixin` adds a `mode` field that controls which keybindings are active, initially set to the command mode, and a method `toggleMode` that is used to switch between modes. It also provides a method `onKey`.

Now we can compose `spellMixin` with `modalMixin` to produce a combination of functionality, mimicking some form of multiple inheritance:

```
class IDEEditor extends modalMixin(spellMixin(Editor)) {  
  version() { return 0.2; }  
}
```

The class `IDEEditor` extends the base class `Editor` with modal editing and spell checking capabilities. It also defines the missing `version` method.

At first glance, `IDEEditor` looks quite fine, but it has a subtle issue. Recall that two mixins `modalMixin` and `spellMixin` both provide a method `onKey`, and the `Editor` class also defines an `onKey` method of its own. Now we have a name clash. A question arises as to which one gets picked inside the `IDEEditor` class. A typical mixin model resolves this issue by looking at the order of mixin applications. Mixins appearing later in the order overrides *all* the identically named methods of earlier mixins. So in our case, `onKey` in `modalMixin` gets picked. If we change the order of application to `spellMixin(modalMixin(Editor))`, then `onKey` in `spellMixin` is inherited.

Problem of Mixin Composition. From the above discussion, we can see that mixins are composed linearly: all the mixins used by a class must be applied one at a time. However, when we wish to resolve conflicts by selecting features from different mixins, we may not be able to find a suitable order. For example, when we compose the two mixins to make the class `IDEEditor`, we can choose which of them comes first, but in either order, `IDEEditor` cannot access to the `onKey` method in the `Editor` class.

Trait Model. Because of the total ordering and the limited means for resolving conflicts imposed by the mixin model, researchers have proposed a simple compositional model called traits [47, 21]. Traits are lightweight entities and serve as the primitive units of code reuse. Among others, the key difference from mixins is that the order of trait composition is irrelevant, and conflicting methods must be resolved *explicitly*. This gives programmers fine-grained control, when conflicts arise, of selecting desired features from different components. Thus we believe traits are a better model for multiple inheritance in statically-typed OO languages, and in `SEDEL` we realize this vision by giving traits a first-class status in the language, achieving more expressive power compared with traditional (second-class) traits.

Summary of Typing Challenges. From our previous discussion, we can identify the following typing challenges for a type system to accommodate the programming patterns (first-class classes/mixins) we have just seen in a typed setting:

- How to account for, in a typed way, abstract methods and dynamic dispatch.
- What are the types of first-class classes or mixins.
- How to type dynamic inheritance.
- How to express constraints on method presence and absence (the use of `super` clearly demands that).
- In the presence of first-class traits, how to detect conflicts statically, even when the traits involved are not statically known.

`SEDEL` elegantly solves the above challenges in a unified way, as we will see next.

2.2 A Glance at Typed First-Class Traits in SEDEL

We now rewrite the above library in SEDEL, but this time with types. The resulting code has the same functionality as the dynamic version, but is statically typed. All code snippets in this and later sections are runnable in our prototype implementation. Before proceeding, we ask the readers to bear in mind that in this section we are not using traits in the most canonical way, i.e., we use traits as if they are classes (but with built-in conflict detection). This is because we are trying to stay as close as possible to the structure of the JavaScript version for ease of comparison. In Section 3 we will remedy this to make better use of traits.

Simple Traits. Below is a simple trait `editor`, which corresponds to the JavaScript class `Editor`. The `editor` trait defines the same set of methods: `on_key`, `do_cut` and `show_help`:

```
trait editor [self : Editor & Version] => {
  on_key(key : String) = "Pressing " ++ key;
  do_cut = self.on_key "C-x" ++ " for cutting text";
  show_help = "Version: " ++ self.version ++ " Basic usage..."
};
```

The first thing to notice is that SEDEL uses a syntax (similar to Scala's self type annotations [36]) where we can give a type annotation to the `self` reference. In the type of `self` we use `&` construct to create intersection types. `Editor` and `Version` are two record types:

```
type Editor = {on_key : String → String, do_cut : String, show_help : String};
type Version = {version : String};
```

For the sake of conciseness, SEDEL uses `type` aliases to abbreviate types.

Self-Types Encode Abstract Methods. Recall that in the JavaScript class `Editor`, the `version` method is undefined, but is used inside `showHelp`. How can we express this in the typed setting, if not with an abstract method? In SEDEL, self-types play the role of trait requirements. As the first approximation, we can justify the use of `self.version` by noticing that (part of) the type of `self` (i.e., `Version`) contains the declaration of `version`. An interesting aspect of SEDEL's trait model is that there is no need for abstract methods. Instead, abstract methods can be simulated as requirements of a trait. Later, when the trait is composed with other traits, *all* requirements on the self-types must be satisfied and one of the traits in the composition must provide an implementation of the method `version`.

As in the JavaScript version, the `on_key` method is invoked on `self` in the body of `do_cut`. This is allowed as (part of) the type of `self` (i.e., `Editor`) contains the signature of `on_key`. Comparing `editor` to the JavaScript class `Editor`, almost everything stays the same, except that we now have the typed version. As a side note, since SEDEL is currently a pure functional OO language, there is no difference between fields and methods, so we can omit empty arguments and parameter parentheses.

First-Class Traits and Trait Expressions. SEDEL treats traits as first-class expressions, putting them in the same syntactic category as objects, functions, and other primitive forms. To illustrate this, we give the SEDEL version of `spellMixin`:

```
type Spelling = {check : String};
type OnKey    = {on_key : String → String};

spell_mixin [A * Spelling & OnKey] (base : Trait[Editor & Version, Editor & A]) =
  trait [self : Editor & Version] inherits base => {
    override on_key(key : String) = "Process " ++ key ++ " on spell editor";
    check = super.on_key "C-c" ++ " for spelling check"
  };
```


This looks daunting at first, but `spell_mixin` has almost the same structure as its JavaScript cousin `spellMixin`, albeit with some type annotations. In SEDEL, we use capital letters (`A`, `B`, ...) to denote type variables, and trait expressions `trait [self : ...] inherits ... => {...}` to create first-class traits. Trait expressions have trait types of the form `Trait [T1, T2]` where `T1` and `T2` denote trait requirements and functionality respectively. We will explain trait types in Section 3. Despite the structural similarities, there are several significant features that are unique to SEDEL (e.g., the disjointness operator `*`). We discuss these in the following.

Disjoint Polymorphism and Conflict Detection. SEDEL uses a type system based on *disjoint intersection types* [40] and *disjoint polymorphism* [7]. Disjoint intersections empower SEDEL to detect conflicts statically when trying to compose two traits with identically named features. For example composing two traits `a` and `b` that both provide `foo` gives a type error (the overloaded `&` operator denotes trait composition):

```
trait a => { foo = 1 };
trait b => { foo = 2 };
trait c inherits a & b => {}; -- type error!
```

Disjoint polymorphism, as a more advanced mechanism, allows detecting conflicts even in the presence of polymorphism – for example when a trait is parameterized and its full set of methods is not statically known. As can be seen, `spell_mixin` is actually a polymorphic function. Unlike ordinary parametric polymorphism, in SEDEL, a type variable can also have a disjointness constraint. For instance, `A * Spelling & OnKey` means that `A` can be instantiated to any type as long as it *does not* contain `check` and `on_key`. To mimic mixins, the argument `base`, which is supposed to be some trait, serves as the “base” trait that is being inherited. Notice that the type variable `A` appears in the type of `base`, which essentially states that `base` is a trait that contains at least those methods specified by `Editor`, and possibly more (which we do not know statically). Also note that leaving out the `override` keyword will result in a type error. The type system is forcing us to be very specific as to what is the intention of the `on_key` method because it sees the same method is also declared in `base`, and blindly inheriting `base` will definitely cause a method conflict. As a final note, the use of `super` inside `check` is allowed because the “super” trait `base` implements `on_key`, as can be seen from its type.

Dynamic Inheritance. Disjoint polymorphism enables us to correctly type dynamic inheritance: `spell_mixin` is able to take any trait that conforms with its assigned type, equips it with the `check` method and overrides its old `on_key` method. As a side note, the use of disjoint polymorphism is essential to correctly model the mixin semantics. From the type we know `base` has some features specified by `Editor`, plus something more denoted by `A`. By inheriting `base`, we are guaranteed that the result trait will have everything that is already contained in `base`, plus more features. This is in some sense similar to row polymorphism [55] in that the result trait is prohibited from forgetting methods from the argument trait. As we will discuss in Section 6, disjoint polymorphism is more expressive than row polymorphism.

Typing Mixin Composition. Next we give the typed version of `modalMixin` as follows:

```
type ModalEdit = {mode : String, toggle_mode : String};

modal_mixin [A * ModalEdit & OnKey] (base : Trait[Editor & Version, Editor & A]) =
  trait [self : Editor & Version] inherits base => {
    override on_key(key : String) = "Process " ++ key ++ " on modal editor";
    mode = "command";
    toggle_mode = "toggle succeeded"
  };
```


Now the definition of `modal_mixin` should be self-explanatory. Finally we can apply both “mixins” one by one to `editor` to create a concrete editor:

```
type IDEEditor = Editor & Version & Spelling & ModalEdit;

trait ide_editor [self : IDEEditor]
  inherits modal_mixin Spelling (spell_mixin T editor) => { version = "0.2" };
```

As with the JavaScript version, we need to fill in the missing `version` method. It is easy to verify that the `on_key` method in `modal_mixin` is inherited. Compared with the untyped version, here this behaviour is reasonable because in each mixin we specifically tags the `on_key` method to be an overriding method. Let us take a close look at the mixin applications. Since SEDEL is currently explicitly typed, we need to provide concrete types when using `modal_mixin` and `spell_mixin`. In the inner application (`spell_mixin T editor`), we use the top type `T` to instantiate `A` because the `editor` trait provides exactly those method specified by `Editor` and nothing more (hence `T`). In the outer application, we use `Spelling` to instantiate `A`. This is where implicit conflict resolution of mixins happens. We know the result of the inner application actually forms a trait that provides both `check` and `on_key`, but the disjointness constraint of `A` requires the absence of `on_key`, thus we cannot instantiate `A` to `Spelling & OnKey` for example when applying `modal_mixin`. Therefore the outer application effectively excludes `on_key` from `spell_mixin`. In summary, the order of mixin applications is reflected by the order of function applications, and conflict resolution code is implicitly embedded. Of course changing the mixin application order to `spell_mixin ModalEdit (modal_mixin T editor)` gives the expected behaviour.

Admittedly the typed version is unnecessarily complicated as we were mimicking mixins by functions over traits. The final editor `ide_editor` suffers from the same problem as the class `IDEEditor`, since there is no obvious way to access the `on_key` method in the `editor` trait.³ Section 3 makes better use of traits to simplify the editor code.

3 Typed First-Class Traits

In Section 2 we have seen some examples of first-class traits at work in SEDEL. In this section we give a detailed account of SEDEL’s support for typed first-class traits, to complement what has been presented so far. In doing so, we simplify the examples in Section 2 to make better use of traits. Section 4 presents the formal type system of first-class traits.

3.1 Traits in SEDEL

SEDEL supports a simple, yet expressive form of traits [47]. Traits provide a simple mechanism for fine-grained code reuse, which can be regarded as a disciplined form of multiple inheritance. A trait is similar to a mixin in that it encapsulates a collection of related methods to be added to a class. The practical difference between traits and mixins is the way conflicting features that typically arise in multiple inheritance are dealt with. Instead of automatically resolved by scoping rules, conflicts are, in SEDEL, detected by the type system, and explicitly resolved by the programmer. Compared with traditional trait models, there are three interesting points about SEDEL’s traits: (1) they are *statically typed*; (2) they are *first-class* values; and (3) they support *dynamic inheritance*. The support for such combination of features is one of the key novelties of SEDEL. Another minor difference from traditional traits (e.g., in Scala) is that, due to the use of structural types, a trait name is not a type.

³ In fact, as we will see in Section 3, we can still access `on_key` in `editor` by the forwarding operator.

3.2 Two Roles of Traits in SEDEL

Traits as Templates for Creating Objects. An obvious difference between traits in SEDEL and many other models of traits [47, 25, 37] is that they directly serve as templates for objects. In many other trait models, traits are complemented by classes, which take the responsibility for object creation. In particular, most models of traits do not allow constructors for traits. However, a trait in SEDEL has a single constructor of the same name. Take our last trait `ide_editor` in Section 2 for example:

```
a_editor1 = new[IDEditor] ide_editor;
```

As with conventional OO languages, the keyword `new` is used to create an object. A difference to other OO languages is that the keyword `new` also specifies the intended type of the object. We instantiate the `ide_editor` trait and create an object `a_editor1` of type `IDEditor`. As we will see in Section 3.4, constructors with parameters can also be expressed.

It is tempting to try to instantiate the `editor` trait such as `new[Editor] editor`. However this results in a type error, because as we discussed, `editor` has no definition of `version`, and blindly instantiating it would cause runtime error. This behaviour is on a par with Java's abstract classes – traits with undefined methods cannot be instantiated on their own.

Traits as Units of Code Reuse. The traditional role of traits is to serve as units of code reuse. SEDEL's traits can have this role as well. Our `spell_mixin` function in Section 2 is more complicated than it should be. This is because we were mimicking classes as traits, and mixins as functions over traits. Instead, traits already provide a mechanism of code reuse. To illustrate this, we simplify `spell_mixin` as follows:

```
trait spell [self : OnKey] => {
  on_key(key : String) = "Process " ++ key ++ " on spell editor";
  check = self.on_key "C-c" ++ " for spell checking"
};
```

This is much cleaner. The trait `spell` adds a method `check`. It also defines a method `on_key`. A key difference with `spell_mixin` is that `on_key` is invoked on the `self` parameter instead of `super`. Note that this does not necessarily mean `check` will call `on_key` defined in the same trait. As we will see, the actual behaviour entirely depends on how we compose `spell` with other traits. One minor difference is that we do not need to tag `on_key` with the `override` keyword, because `spell` stands as a standalone entity. Another interesting point is that the self-type `OnKey` is not the same as that of the trait body, which also contains the `check` method. In SEDEL, self-types of traits are known as trait *requirements*.

Classes and/or Traits. In the literature on traits [21, 47], the aforementioned two roles are considered as competing. One reason of the two roles conflicting in class-based languages is because a class must adopt a fixed position in the class hierarchy and therefore it can be difficult to reuse and resolve conflicts, whereas in SEDEL, a trait is a standalone entity and is not tied to any particular hierarchy. Therefore we can view our traits either as generators of instances, or units of reuse. Another important reason why our model can do just with traits is because we have a pure language. Mutable state can often only appear in classes in imperative models of traits, which is a good reason for having both classes and traits.

3.3 Trait Types and Trait Requirements

Object Types and Trait Types. SEDEL adopts a relatively standard foundational model of object-oriented constructs [30] where objects are encoded as records with a structural type.

This is why the type of the object `a_editor1` is the record type `IDEEditor`. In SEDEL, an object type is different from a trait type. A trait type is specified with the keyword **Trait**. For example, the type of the `spell` trait is **Trait**[`OnKey`, `OnKey & Spelling`].

Trait Requirements and Functionality. In general, a trait type **Trait** [`T1`, `T2`] specifies both the *requirements* `T1` and the *functionality* `T2` of a trait. The requirements of a trait denote the types/methods that the trait needs to support for defining the functionality it provides. Both are reflected in the trait type. For example, `spell` has type **Trait**[`OnKey`, `OnKey & Spelling`], which means that `spell` requires some implementation of the `on_key` method, and it provides implementations for the `on_key` and `check` methods. When a trait has no requirements, the absence of a requirement is denoted by using the top type (\top). A simplified sugar **Trait** [`T`] is used to denote a trait without requirements, but providing functionality `T`.

Trait Requirements as Abstract Methods. Let us go back to our very first trait `editor`. Note how in `editor` the type of the `self` parameter is `Editor & Version`, where `Version` contains a declaration of the `version` method that is needed for the definition of `show_help`. Note also that the trait itself does not actually contain a `version` definition. In many other OO models a similar program could be achieved by having an *abstract* definition of `version`. In SEDEL there are no abstract definitions (methods or fields), but a similar result can be achieved via trait requirements. Requirements of a trait are met at the object creation point. For example, as we mentioned before, the `editor` trait alone cannot be instantiated since it lacks `version`. However, when it is composed with a trait that provides `version`, the composition can be instantiated, as shown below:

```
trait foo => { version = "0.2" };
bar = new[Editor & Version] foo & editor;
```

SEDEL uses a syntax where the `self` parameter can be explicitly named (not necessarily named `self`) with a type annotation. When the `self` parameter is omitted (for example in the `foo` trait above), its type defaults to \top . This is different from typical OO languages, where the default type of the `self` parameter is the same as the class being defined.

Intersection Types Model Subtyping. `IDEEditor` is defined as an intersection type (`Editor & Version & Spelling & ModalEdit`). Intersection types [20, 43] have been woven into many modern languages these days. A notable example is Scala, which makes fundamental use of intersection types to express a class/trait that extends multiple other traits. An intersection type such as `T1 & T2` contains exactly those values which can be used as values of type `T1` and of type `T2`, and as such, `T1 & T2` immediately introduces a subtyping relation between itself and its two constituent types `T1` and `T2`. Unsurprisingly, `IDEEditor` is a subtype of `Editor`.

3.4 Traits with Parameters and First-Class Traits

So far our uses of traits involve no parameters. Instead of inventing another trait syntax with parameters, a trait with parameters is just a function that produces a trait expression, since functions already have parameters of their own. This is one benefit of having first-class traits in terms of language economy. To illustrate, let us simplify `modal_mixin` in a similar way as in `spell_mixin`:

```
modal (init_mode : String) = trait => {
  on_key(key : String) = "Process " ++ key ++ " on modal editor";
  mode = init_mode;
  toggle_mode = "toggle succeeded"
};
```

The first thing to notice is that `modal` is a function with one argument, and returns a trait expression, which essentially makes `modal` a trait with one parameter. Now it is easy to see that a trait declaration `trait name [self : ...] => {...}` is just syntactic sugar for function definition `name = trait [self : ...] => {...}`. The body of the `modal` trait is straightforward. We initialize the `mode` field to `init_mode`. The `modal` trait also comes with a constructor with one parameter, so we can do `new[ModalEdit] (modal "insert")` for example.

3.5 Detecting and Resolving Conflicts in Trait Composition

A common problem in multiple inheritance is how to detect and/or resolve conflicts. For example, when inheriting from two traits that have the same field, then it is unclear which implementation to choose. There are various approaches to dealing with conflicts. The trait-based approach requires conflicts to be resolved at the level of the composition, otherwise the program is rejected by the type system. SEDEL provides a means to help resolve conflicts.

We start by assembling all the traits defined in this section to create the final editor with the same functionality as `ide_editor` in Section 2. Our first try is as follows:

```
ide_editor (init_mode : String) = trait [self : IDEEditor]
  -- conflict
  inherits editor & spell & modal init_mode => { version = "0.2" };
```

Unfortunately the above trait gets rejected by SEDEL because `editor`, `spell` and `modal` all define an `on_key` method. Recall that in Section 2, when we use a mixin-style composition, the conflict resolution code has been hardwired in the definition. However, in a trait-style composition, this is not the case: conflicts must be resolved *explicitly*. The above definition is ill-typed precisely because there is a conflicting method `on_key`, thus violating the disjointness conditions of disjoint intersection types.

Resolving Conflicts. To resolve the conflict, we need to explicitly state which `on_key` gets to stay. SEDEL provides such a means, the so-called *exclusion* operator (denoted by `\`), which allows one to exclude a field/method from a given trait. The following matches the behaviour in Section 2 where `on_key` in the `modal` trait is selected:

```
ide_editor (init_mode : String) = trait [self : IDEEditor]
  inherits editor \ {on_key : String → String} &
    spell \ {on_key : String → String} & modal init_mode =>
    { version = "0.2" };
```

Now the above code type checks. We can also select `on_key` in the `spell` trait as easily:

```
ide_editor2 (init_mode : String) = trait [self : IDEEditor]
  inherits editor \ {on_key : String → String} &
    spell & (modal init_mode) \ {on_key : String → String} =>
    { version = "0.2" };
```

In Section 2 we mentioned that in the mixin style, it is impossible to select `on_key` in the `editor` trait, but this is not a problem here:

```
ide_editor3 (init_mode : String) = trait [self : IDEEditor]
  inherits editor & spell \ {on_key : String → String} &
    (modal init_mode) \ {on_key : String → String} =>
    { version = "0.2" };
```

The Forwarding Operator. Another operator that SEDEL provides is the so-called *forwarding* operator, which can be useful when we want to access some method that has been

explicitly excluded in the `inherits` clause. This is a common scenario in diamond inheritance, where `super` is not enough. Below we show a variant of `ide_editor`:

```
ide_editor4 (init_mode : String) = trait [self : IDEEditor]
  inherits editor \ {on_key : String → String} &
    spell \ {on_key : String → String} &
    modal init_mode => {
      version = "0.2";
      override on_key(key : String) =
        super.on_key key ++ " and " ++ (spell ^ self).on_key key
    };
```

Notice that `on_key` in `spell` has been excluded. However, we can still access it by using the forwarding operator as in `spell ^ self`, which gives full access to all the methods in `spell`. Also note that using `super` only gives us access to `on_key` in the modal trait. To see `ide_editor4` in action, we create a small test:

```
a_editor2 = new[IDEEditor] (ide_editor4 "command");
main = a_editor2.do_cut
-- "Process C-x on modal editor and Process C-x on spell editor for cutting text"
```

3.6 Disjoint Polymorphism and Dynamic Composition

SEDEL supports disjoint polymorphism. The combination of disjoint polymorphism and first-class traits enables the highly modular code where traits with *statically unknown* types can be instantiated and composed in a type-safe way! The following is illustrative of this:

```
merge A [B * A] (x : Trait[A]) (y : Trait[B]) = new[A & B] x & y;
```

The `merge` function takes two traits `x` and `y` of some arbitrary types `A` and `B`, composes them, and instantiates an object with the resulting composed trait. Clearly such composition cannot always work if `A` and `B` can have conflicts. However, `merge` has a constraint `B * A` that ensures that whatever types are used to instantiate `A` and `B` they must be disjoint. Thus, under the assumption that `A` and `B` are disjoint the code type-checks. We want to emphasize that row polymorphism is unable to express this kind of disjointness of two polymorphic types, thus systems using row polymorphism is unable to define the `merge` function, which plays an essential role in Section 5.

4 Formalizing Typed First-Class Traits

This section presents the syntax and semantics of SEDEL. In particular, we show how to elaborate high-level source language constructs (self-references, abstract methods, first-class traits, dynamic inheritance, etc) in SEDEL to F_i [7], a pure record calculus with disjoint polymorphism. The treatment of the self-reference and dynamic dispatching is inspired by Cook and Palsberg's work on the denotational semantics for inheritance [19]. We then prove the elaboration is type safe, i.e., well-typed SEDEL expressions are translated to well-typed F_i terms. Finally we show that SEDEL is coherent. Full proofs can be found in the appendix.

4.1 Syntax

The core syntax of SEDEL is shown in Fig. 1, with trait related constructs highlighted. For brevity of the meta-theoretic study, we do not consider definitions, which can be added in standard ways.

9:14 Typed First-Class Traits

Types	A, B, C	$::=$	$\top \mid \text{Int} \mid A \rightarrow B \mid A \& B \mid \{l : A\} \mid \alpha \mid \forall(\alpha * A). B \mid \mathbf{Trait} [A, B]$
Expressions	E	$::=$	$\top \mid i \mid x \mid \lambda x. E \mid E_1 E_2 \mid \Lambda(\alpha * A). E \mid E A \mid E_1 , , E_2 \mid E : A$ $\mid \{l = E\} \mid E.l \mid \mathbf{letrec} x : A = E_1 \mathbf{in} E_2 \mid \mathbf{new} [A](\overline{E}_i^i) \mid E_1 \hat{\ } E_2$ $\mid \mathbf{trait} [\mathbf{self} : B] \mathbf{inherits} \overline{E}_i^i \{ \overline{l}_j = \overline{E}_j^j \} : A$
Contexts	Γ	$::=$	$\bullet \mid \Gamma, x : A \mid \Gamma, \alpha * A$
	Record types	$\{l_1 : A_1, \dots, l_n : A_n\}$	$::= \{l_1 : A_1\} \& \dots \& \{l_n : A_n\}$
	Records	$\{l_1 = E_1, \dots, l_n = E_n\}$	$::= \{l_1 = E_1\} , , \dots , , \{l_n = E_n\}$

■ **Figure 1** SEDEL core syntax and syntactic abbreviations.

Types. Metavariables A, B, C range over types. Types include a top type \top , type of integers Int , function types $A \rightarrow B$, intersection types $A \& B$, singleton record types $\{l : A\}$, type variables α and disjoint (universal) quantification $\forall(\alpha * A). B$. The main novelty is the type of first-class traits $\mathbf{Trait} [A, B]$, which expresses the requirement A and the functionality B . We will use $[A/\alpha]B$ to denote capture-avoiding substitution of A for α inside B .

Expressions. Metavariable E ranges over expressions. We start with constructs required to encode objects based on records: term variables x , lambda abstractions $\lambda x. E$, function applications $E_1 E_2$, singleton records $\{l = E\}$, record projections $E.l$, recursive let bindings $\mathbf{letrec} x : A = E_1 \mathbf{in} E_2$, disjoint type abstraction $\Lambda(\alpha * A). E$ and type application $E A$. The calculus also supports a merge construct $E_1 , , E_2$ for creating values of intersection types and annotated expressions $E : A$. We also include a canonical top value \top and integer literals i .

First-class traits and trait expressions. The central construct of SEDEL is the trait expression $\mathbf{trait} [\mathbf{self} : B] \mathbf{inherits} \overline{E}_i^i \{ \overline{l}_j = \overline{E}_j^j \} : A$, which specifies a (possibly empty) list of trait expressions \overline{E}_i in the **inherits** clause, an explicit **self** reference (with type annotation B), and a set of methods $\{ \overline{l}_j = \overline{E}_j^j \}$. Intuitively this trait expression has type $\mathbf{Trait} [B, A]$. Unlike the conventional trait model, a trait expression denotes a first-class value: it may occur anywhere where an expression is expected. Trait instantiation expressions $\mathbf{new} [A](\overline{E}_i^i)$ instantiate a composition of trait expressions \overline{E}_i to create an object of type A . Finally $E_1 \hat{\ } E_2$ is the forwarding expression, where E_1 should be some trait.

Abbreviations. For ease of programming, multiple-field record types are merely syntactic sugar for intersections of single-field record types. Similarly, multi-field record expressions are syntactic sugar for merges of single-field records.

4.2 Semantics

Subtyping and Well-formedness. Figure 2 shows the most relevant subtyping and well-formedness rules for SEDEL. Omitted rules are standard and can be found in previous work [7]. The subtyping rule for trait types (rule SUB-TRAIT) resembles the one for function types (rule SUB-ARR) in that it is contravariant on the first type A and covariant on the second type B . The well-formedness rule for trait types is straightforward.

Disjointness. Figure 3 shows the disjointness judgment $\Gamma \vdash A * B$, which is used for example in rule WF-AND. The disjointness checking is the underlying mechanism of conflict detection.

$$\boxed{A <: B} \quad (\textit{Subtyping})$$

$$\frac{B_1 <: A_1 \quad A_2 <: B_2}{A_1 \rightarrow A_2 <: B_1 \rightarrow B_2} \text{SUB-ARR} \qquad \frac{B_1 <: A_1 \quad A_2 <: B_2}{\mathbf{Trait}[A_1, A_2] <: \mathbf{Trait}[B_1, B_2]} \text{SUB-TRAIT}$$

$$\boxed{\Gamma \vdash A} \quad (\textit{Well formedness})$$

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B \quad \Gamma \vdash A * B}{\Gamma \vdash A \& B} \text{WF-AND} \qquad \frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash \mathbf{Trait}[A, B]} \text{WF-TRAIT}$$

■ **Figure 2** Subtyping and well-formedness of SEDEL (excerpt).

We naturally extend the disjointness rules in F_i to cover trait types. We refer to their paper [7] for further explanation. Here we discuss the rules related with traits. Rule D-TRAIT says that as long as the functionalities that two traits provide are disjoint, the two trait types are disjoint. Rules D-TRAITARR1 and D-TRAITARR2 deal with situations where one of the two types is a function type. At first glance, these two look strange because a trait type is *different* from a function type, and they ought to be disjoint as an axiom. The reason is that SEDEL has an elaboration semantics, and as we will see, trait types are translated to function types. In order to ensure the elaboration is type-safe, we have to have special treatment for trait and function types. In principle, if SEDEL has its own semantics, then trait types are always disjoint to function types. The axiom rules of the form $A *_{ax} B$ take care of two types with different language constructs.

Typing Traits. The typing rules of trait related constructs are shown in Fig. 4. The full set of rules can be found in the appendix. The reader is advised to ignore the highlighted parts for now. SEDEL employs two modes: the inference mode (\Rightarrow) and the checking mode (\Leftarrow). The inference judgment $\Gamma \vdash E \Rightarrow A$ says that we can synthesize a type A for expression E in the context Γ . The checking judgment $\Gamma \vdash E \Leftarrow A$ checks E against A in the context Γ . One representative of inference rules is

$$\frac{\Gamma \vdash E_1 \Rightarrow A \rightsquigarrow e_1 \quad \Gamma \vdash E_2 \Rightarrow B \rightsquigarrow e_2 \quad \Gamma \vdash A * B}{\Gamma \vdash E_1, , E_2 \Rightarrow A \& B \rightsquigarrow e_1, , e_2} \text{INF-MERGE}$$

which says that a merge of two expressions is valid only if their types are disjoint. This is the underlying mechanism for conflict detection. One representative of checking rules is

$$\frac{\Gamma \vdash E \Rightarrow A \rightsquigarrow e \quad A <: B \quad \Gamma \vdash B}{\Gamma \vdash E \Leftarrow B \rightsquigarrow e} \text{CHK-SUB}$$

where subtyping is used to coerce expressions of one type to another.

To type-check a trait (rule INF-TRAIT) we first type-check if its inherited traits \overline{E}_i are valid traits. Note that each trait E_i can possibly refer to self. Methods must all be well-typed in the usual sense. Apart from these, we have several side-conditions to make sure traits are well-behaved. The well-formedness judgment $\Gamma \vdash C_1 \& .. \& C_n \& C$ ensures that we do not have conflicting methods (in inherited traits and the body). The subtyping judgments $\overline{B} <: \overline{B}_i$ ensure that the self parameter satisfies the requirements imposed by each inherited trait. Finally the subtyping judgment $C_1 \& .. \& C_n \& C <: A$ sanity-checks that the assigned type A is compatible.

$\Gamma \vdash A * B$	<i>(Disjointness)</i>		
$\frac{\text{D-TOP}}{\Gamma \vdash \top * A}$	$\frac{\text{D-TOPSYM}}{\Gamma \vdash A * \top}$	$\frac{\text{D-VAR} \quad \alpha * A \in \Gamma \quad A <: B}{\Gamma \vdash \alpha * B}$	$\frac{\text{D-VARSYM} \quad \alpha * A \in \Gamma \quad A <: B}{\Gamma \vdash B * \alpha}$
$\frac{\text{D-FORALL} \quad \Gamma, \alpha * A_1 \& A_2 \vdash B * C}{\Gamma \vdash \forall (\alpha * A_1). B * \forall (\alpha * A_2). C}$	$\frac{\text{D-REC} \quad \Gamma \vdash A * B}{\Gamma \vdash \{l : A\} * \{l : B\}}$	$\frac{\text{D-RECN} \quad l_1 \neq l_2}{\Gamma \vdash \{l_1 : A\} * \{l_2 : B\}}$	
$\frac{\text{D-ARROW} \quad \Gamma \vdash A_2 * B_2}{\Gamma \vdash A_1 \rightarrow A_2 * B_1 \rightarrow B_2}$	$\frac{\text{D-ANDL} \quad \Gamma \vdash A_1 * B \quad \Gamma \vdash A_2 * B}{\Gamma \vdash A_1 \& A_2 * B}$	$\frac{\text{D-ANDR} \quad \Gamma \vdash A * B_1 \quad \Gamma \vdash A * B_2}{\Gamma \vdash A * B_1 \& B_2}$	
$\frac{\text{D-TRAIT} \quad \Gamma \vdash A_2 * B_2}{\Gamma \vdash \mathbf{Trait} [A_1, A_2] * \mathbf{Trait} [B_1, B_2]}$	$\frac{\text{D-TRAITARR1} \quad \Gamma \vdash A_2 * B_2}{\Gamma \vdash \mathbf{Trait} [A_1, A_2] * B_1 \rightarrow B_2}$		
$\frac{\text{D-TRAITARR2} \quad \Gamma \vdash A_2 * B_2}{\Gamma \vdash A_1 \rightarrow A_2 * \mathbf{Trait} [B_1, B_2]}$		$\frac{\text{D-AX} \quad A *_{ax} B}{\Gamma \vdash A * B}$	
$A *_{ax} B$	<i>(Disjointness axiom)</i>		
$\frac{\text{DAX-INTTRAIT}}{\mathbf{Int} *_{ax} \mathbf{Trait} [A_1, A_2]}$	$\frac{\text{DAX-TRAITFORALL}}{\mathbf{Trait} [A_1, A_2] *_{ax} \forall (\alpha * B_1). B_2}$	$\frac{\text{DAX-TRAITREC}}{\mathbf{Trait} [A_1, A_2] *_{ax} \{l : B\}}$	

■ **Figure 3** Disjointness rules of SEDEL (excerpt).

Trait instantiation (rule INF-NEW) requires that each instantiated trait is valid. There are also several side-conditions, which serve the same purposes as in rule INF-TRAIT. Rule INF-FORWARD says that the first operand E_1 of the forwarding operator must be a trait. Moreover, the type of the second operand E_2 must satisfy the requirement of E_1 .

Treatments of Exclusion, Super and Override. One may have noticed that in Fig. 1 we did not include the exclusion operator in the core SEDEL syntax, neither do **super** and **override** appear. The reason is that in principle all uses of the exclusion operator can be replaced by type annotations. For example to exclude a **bar** field from $\{\mathbf{foo} = \mathbf{a}, \mathbf{bar} = \mathbf{b}, \mathbf{baz} = \mathbf{c}\}$, all we need is to annotate the record with type $\{\mathbf{foo} : \mathbf{A}, \mathbf{baz} : \mathbf{C}\}$ (suppose **a** has type **A**, etc). By rule CHK-SUB, the resulting record is guaranteed to contain no **bar** field. In the same vein, the use of **override** can be explained using the exclusion operator. The **super** keyword is internally a variable pointing to the **inherits** clause (its typing rule is similar to rule INF-TRAIT and can be found in the appendix). We omit all of these features in the meta-theoretic study in order to focus our attention on the essence of first-class traits. However in practice, this is rather inconvenient as we need to write down all types we wish to retain rather than the one to exclude. So in our implementation we offer all of them.

Elaboration. The operational semantics of SEDEL is given by means of a type-directed translation into F_i extended with (lazy) recursive let bindings. This extension is standard and type-safe. The syntax of F_i is shown in Fig. 5. Let us go back to Fig. 4, now focusing on the **highlighted** parts, which denote the elaborated F_i terms. Most of them are straightforward

$$\boxed{\Gamma \vdash E \Rightarrow A \rightsquigarrow e} \quad (\text{Infer})$$

$$\begin{array}{c}
\text{INF-TRAIT} \\
\frac{\Gamma, \text{self} : B \vdash E_i \Rightarrow \mathbf{Trait} [B_i, C_i] \rightsquigarrow e_i^{i \in 1..n} \quad \Gamma, \text{self} : B \vdash \{ l_j = \overline{E_j^{j \in 1..m}} \} \Rightarrow C \rightsquigarrow e}{\overline{B} <: \overline{B_i}^{i \in 1..n} \quad \Gamma \vdash C_1 \& \dots \& C_n \& C \quad C_1 \& \dots \& C_n \& C <: A} \\
\Gamma \vdash \mathbf{trait} [\text{self} : B] \mathbf{inherits} \overline{E_i}^{i \in 1..n} \{ l_j = \overline{E_j^{j \in 1..m}} \} : A \Rightarrow \mathbf{Trait} [B, A] \rightsquigarrow \lambda(\text{self} : |B|). (\overline{(e_i \text{self})}^{i \in 1..n}), e
\end{array}$$

$$\begin{array}{c}
\text{INF-FORWARD} \\
\frac{\Gamma \vdash E_1 \Rightarrow \mathbf{Trait} [A, B] \rightsquigarrow e_1 \quad \Gamma \vdash E_2 \Leftarrow A \rightsquigarrow e_2}{\Gamma \vdash E_1 \wedge E_2 \Rightarrow B \rightsquigarrow e_1 e_2}
\end{array}$$

$$\begin{array}{c}
\text{INF-NEW} \\
\frac{\Gamma \vdash E_i \Rightarrow \mathbf{Trait} [A_i, B_i] \rightsquigarrow e_i^{i \in 1..n} \quad A <: A_i^{i \in 1..n} \quad \Gamma \vdash B_1 \& \dots \& B_n \quad B_1 \& \dots \& B_n <: A}{\Gamma \vdash \mathbf{new} [A] (\overline{E_i}^{i \in 1..n}) \Rightarrow A \rightsquigarrow \mathbf{letrec} \text{ self} : |A| = (\overline{e_i \text{self}})^{i \in 1..n} \mathbf{in} \text{self}}
\end{array}$$

■ **Figure 4** Typing of SEDEL (excerpt).

$$\begin{array}{l}
\text{Types} \quad \tau, \sigma ::= \top \mid \mathbf{Int} \mid \tau \rightarrow \sigma \mid \tau \& \sigma \mid \{ l : \tau \} \mid \alpha \mid \forall (\alpha * \tau). \sigma \\
\text{Expressions} \quad e ::= \top \mid i \mid x \mid \lambda x. e \mid e_1 e_2 \mid \Lambda (\alpha * \tau). e \mid e \tau \mid e_1, e_2 \mid e : \tau \\
\quad \quad \quad \mid \{ l = e \} \mid e. l \mid \mathbf{letrec} \ x : \tau = e_1 \mathbf{in} \ e_2
\end{array}$$

■ **Figure 5** Syntax of F_i with let bindings.

translations and are thus omitted. We explain the most involved rules regarding traits. In rule INF-TRAIT, a trait is translated into a lambda abstraction with `self` as the formal parameter. In essence a trait corresponds to what Cook and Palsberg [19] call a *generator*. The translations of the inherited traits (i.e., $\overline{e_i}$) are each applied to `self` and then merged with the translation of the trait body e . Now it is clear why we require B (the type of `self`) to be a subtype of each B_i (the requirement of each inherited trait). Note that we abuse the bar notation here with the intention that $\overline{(e_i \text{self})}^{i \in 1..n}$ means $e_1 \text{self}, \dots, e_n \text{self}$. Here is an example of translating the `ide_editor` trait from Section 2 into plain F_i terms equipped with definitions (suppose `modal_mixin` and `spell_mixin` have been translated accordingly):

The gray parts in rule INF-NEW show the translation of trait instantiation. First we apply every translation (i.e., e_i) of the instantiated traits to the `self` parameter, and then merge the applications together. The bar notation is interpreted similarly to the translation in rule INF-TRAIT. Finally we compute the *lazy* fixed-point of the resulting merge term, i.e., self-reference must be updated to refer to the whole composition. Taking the fixed-point of the traits/generators again follows the denotational inheritance model by Cook and Palsberg. This is the key to the correct implementation of dynamic dispatching. Finally, rule INF-FORWARD translates forwarding expressions to function applications. We show the translation of the `a_editor1` object in Section 3 to illustrate the translation of instantiation:

`--END_EDITOR_INST`

One remarkable point is that, while Cook and Palsberg work is done in an untyped setting, here we apply their ideas in a setting with disjoint intersection types and disjoint polymorphism. Our work shows that disjoint intersection types blend in quite nicely with Cook and Palsberg's denotational model of inheritance.

Flattening Property. In the literature of traits [21, 47, 34], a distinguished feature of traits is the so-called *flattening property*. This property says that a (non-overridden) method in a

trait has the same semantics as if it were implemented directly in the class that uses the trait. It would be interesting to see if our trait model has this property. One problem in formulating such a property is that flattening is a property that talks about the equivalence between a flattened class (i.e., a class where all trait methods have been inlined) and a class that reuses code from traits. Since SEDEL does not have classes, we cannot state exactly the same property. However, we believe that one way to talk about a similar property for SEDEL is to have something along the lines of the following example:

► **Example 1 (Flattening).** Suppose we have m well-typed (i.e., conflict-free) traits `trait t1 {l11 = E11,...}, ..., trait tm {lm1 = Em1,...}`, each with some number of methods, then `new (trait inherits t1 & ... & tm {})` = `new (trait {l11 = E11,...,lm1 = Em1,...})`

If we elaborate these two expressions, the property boils down to whether two merge terms $(E_1, E_2), (E_2, E_3)$ and $(E_1, (E_2, E_3))$ have the same semantics. As is shown by Bi et al. [13], merges are associative and commutative, so it is not hard to see that the above two expressions are semantically equivalent. We leave it as future work to formally state and prove flattening.

4.3 Type Soundness and Coherence

Since the semantics of SEDEL is defined by elaboration into F_i [7] it is easy to show that key properties of F_i are also guaranteed by SEDEL. In particular, we show that the type-directed elaboration is type-safe in the sense that well-typed SEDEL expressions are elaborated into well-typed F_i terms. We also show that the source language is coherent and each valid source program has a unique (unambiguous) elaboration.

We need a meta-function $|\cdot|$ that translates SEDEL types to F_i types, whose definition is straightforward. Only the translation of trait types deserves attention:

$$|\mathbf{Trait} [A, B]| = |A| \rightarrow |B|$$

That is, trait types are translated to function types. $|\cdot|$ extends naturally to typing contexts. Now we show several lemmas that are useful in the type-safety proof.

► **Lemma 2.** *If $\Gamma \vdash A$ then $|\Gamma| \vdash |A|$.*

Proof. By structural induction on the well-formedness judgment. ◀

► **Lemma 3.** *If $A <: B$ then $|A| <: |B|$.*

Proof. By structural induction on the subtyping judgment. ◀

► **Lemma 4.** *If $\Gamma \vdash A * B$ then $|\Gamma| \vdash |A| * |B|$.*

Proof. By structural induction on the disjointness judgment. ◀

Finally we are in a position to establish the type safety property:

► **Theorem 5 (Type-safe translation).** *We have that:*

■ *If $\Gamma \vdash E \Rightarrow A \rightsquigarrow e$ then $|\Gamma| \vdash e \Rightarrow |A|$.*

■ *If $\Gamma \vdash E \Leftarrow A \rightsquigarrow e$ then $|\Gamma| \vdash e \Leftarrow |A|$.*

Proof. By structural induction on the typing judgment. ◀

► **Theorem 6 (Coherence).** *Each well-typed SEDEL expression has a unique elaboration.*

Proof. By examining every elaboration rule, it is easy to see that the elaborated F_i term in the conclusion is uniquely determined by the elaborated F_i terms in the premises. Then by the coherence property of F_i , we conclude that each well-typed SEDEL expression has a unique unambiguous elaboration, thus SEDEL is coherent. ◀

5 Case Study: Modularizing Language Components

To further illustrate the applicability of SEDEL, we present a case study using Object Algebras [39] and Extensible VISITORS [38, 52]. Encodings of extensible designs for Object Algebras and Extensible VISITORS have been presented in mainstream languages [38, 52, 39, 41, 44]. However, prior approaches are not entirely satisfactory due to the limitations in existing mainstream OO languages. In Section 5.1, we show how SEDEL makes those designs significantly simpler and convenient to use. In particular, SEDEL’s encoding of extensible visitors gives true ASTs and supports conflict-free Object Algebra combinators, thanks to first-class traits and disjoint polymorphism. Based on this technique, Section 5.2 gives a bird-view of several orthogonal features of a small JavaScript-like language from a textbook on Programming Languages [18], and illustrates how various features can be modularly developed and composed to assemble a complete language with various operations baked in. Section 5.3 compares our SEDEL’s implementation with that of the textbook using Haskell in terms of lines of code.

5.1 Object Algebras and Extensible Visitors in SEDEL

First we give a simple introduction to Object Algebras, a design pattern that can solve the Expression Problem [54] (EP) in languages like Java. The objective of EP is to *modularly* extend a datatype in two dimensions: by adding more cases to the datatype and by adding new operations for the datatype. Our starting point is the following code:

```

type ExpAlg[E] = { lit : Int → E, add : E → E → E };
type IEval = { eval : Int };
trait evalAlg => {
  lit (x : Int) = { eval = x };
  add (x : IEval) (y : IEval) = { eval = x.eval + y.eval }
};

```

`ExpAlg[E]` is the generic interface of a simple arithmetic language with two cases, `lit` for literals and `add` for addition. `ExpAlg[E]` is also called an Object Algebra interface. A concrete Object Algebra will implement such an interface by instantiating `E` with a suitable type. Here we also define one operation `IEval`, modelled by a single-field record type. A concrete Object Algebra that implements the evaluation rules is given by a trait `evalAlg`.

First-Class Object Algebra Values. The actual AST of this simple arithmetic language is given as an internal visitor [42]:

```

type Exp = { accept : forall E . ExpAlg[E] → E };

```

Note that Object Algebras as implemented in languages like Java or Scala do not define the type `Exp` because this would make adding new variants very hard. Although extensible versions of this visitor pattern do exist, they usually require complex types using advanced features of generics [39, 52]. However, as we will see, this is not a problem in SEDEL. We can build a value of `Exp` as follows:

```

e1 : Exp = { accept E f = f.add (f.lit 2) (f.lit 3) };

```

Adding a New Operation. We add another operation `IPrint` to the language:

```
type IPrint = { print : String };
trait printAlg => {
  lit (x : Int) = { print = x.toString };
  add (x : IPrint) (y : IPrint) = {
    print = "(" ++ x.print ++ " + " ++ y.print ++ ")"
  }
};
```

This is done by giving another trait `printAlg` that implements the additional `print` method.

Adding a New Case. A second dimension for extension is to add another case for negation:

```
type ExpExtAlg[E] = ExpAlg[E] & { neg : E → E };
trait negEvalAlg inherits evalAlg => {
  neg (x : IEval) = { eval = 0 - x.eval }
};
trait negPrintAlg inherits printAlg => {
  neg (x : IPrint) = { print= "-" ++ x.print }
};
```

This is achieved by extending `evalAlg` and `printAlg`, implementing missing operations for negation, respectively. We define the actual AST similarly:

```
type ExtExp = { accept: forall E. ExpExtAlg[E] → E };
```

and build a value of `-(2 + 3)` while reusing `e1`:

```
e2 : ExtExp = { accept E f = f.neg (e1.accept E f) };
```

Relations between `Exp` and `ExpExt`. At this stage, it is interesting to point out an interesting subtyping relation between `Exp` and `ExtExp`: `ExpExt`, though being an *extension* of `Exp` is actually a *supertype* of `Exp`. As Oliveira [38] observed, these relations are important for legacy and performance reasons since it means that, a value of type `Exp` can be *automatically* and *safely* coerced into a value of type `ExpExt`, allowing some interoperability between new functionality and legacy code. However, to ensure type-soundness, Scala (or other common OO languages) forbids any kind of type-refinement on method parameter types. The consequence of this is that in those languages, it is impossible to express that `ExtExp` is both an extension and a supertype of `Exp`.

Dynamic Object Algebra Composition Support. When programming with Object Algebras, oftentimes it is necessary to pack multiple operations in the same object. For example, in the simple language we have been developing it can be useful to create an object that supports both printing and evaluation. Oliveira and Cook [39] addressed this problem by proposing *Object Algebra combinators* that combine multiple algebras into one. However, as they noted, such combinators written in Java are difficult to use in practice, and they require significant amounts of boilerplate. Improved variants of Object Algebra combinators have been encoded in Scala using intersection types and an encoding of the merge construct [41, 44]. However, the Scala encoding of the merge construct is quite complex as it relies on low-level type-unsafe programming features such as dynamic proxies, reflection or other meta-programming techniques. In SEDEL, the combination of first-class traits, dynamic inheritance and disjoint polymorphism allows type-safe, coherent and boilerplate-free composition of Object Algebras.

```
combine A [B * A] (f : Trait[ExpExtAlg[A]]) (g : Trait[ExpExtAlg[B]]) =
  trait inherits f & g => {};
```

Types	τ	::=	int bool	
Expressions	e	::=	i $e_1 + e_2$ $e_1 - e_2$ $e_1 \times e_2$ $e_1 \div e_2$	<i>natF</i>
			\mathbb{B} if e_1 then e_2 else e_3	<i>boolF</i>
			$e_1 == e_2$ $e_1 < e_2$	<i>compF</i>
			$e_1 \&\& e_2$ $e_1 e_2$	<i>logicF</i>
			x var $x = e_1; e_2$	<i>varF</i>
			$e_1 e_2$	<i>funcF</i>
Programs	<i>pgm</i>	::=	$decl_1 \dots decl_n e$	<i>funcF</i>
Functions	<i>decl</i>	::=	function $f(x : \tau)\{e\}$	<i>funcF</i>
Values	v	::=	i \mathbb{B}	

■ **Figure 6** Mini-JS expressions, values, and types.

That is it. None of the boilerplate in other approaches [39], or type-unsafe meta-programming techniques of other approaches [41, 44] are needed! Two points are worth noting: (1) `combine` relies on *dynamic inheritance*. Notice how `combine` inherits two traits `f` and `g`, for which their implementations are unknown statically; (2) the disjointness constraint (`B * A`) is *crucial* to ensure two Object Algebras (`f` and `g`) are conflict-free when being composed.

To conclude, let us see `combine` in action. We combine `negEvalAlg` and `negPrintAlg`:

```
combinedAlg = combine IEval IPrint negEvalAlg negPrintAlg;
```

The combined algebra `combineAlg` is useful to avoid multiple interpretations of the same AST when running multiple operations. For example, we can create an object `o` that supports both evaluation and printing in one go:

```
o = e2.accept (IEval & IPrint) (new[ExpExtAlg[IEval & IPrint]] combinedAlg);
main = o.print ++ " = " ++ o.eval.toString -- "-(2.0 + 3.0) = -5.0"
```

5.2 Case Study Overview

Now we are ready to see how the same technique scales to modularize different language features. A *feature* is an increment in program functionality [56, 31]. Figure 6 presents the syntax of the expressions, values and types provided by the features; each line is annotated with the corresponding feature name. Starting from a simple arithmetic language, we gradually introduce new features and combine them with some of the existing features to form various languages. Below we briefly explain what constitutes each feature:

- *natF* and *boolF* contain, among others, literals, additions and conditional expressions.
- *compF* and *logicF* introduce comparisons between numbers and logical connectives.
- *varF* introduces local variables and variable declarations.
- *funcF* introduces top-level functions and function calls.

Besides, each feature is packed with 3 operations: evaluator, pretty printer and type checker.

Having the feature set, we can synthesize different languages by selecting one or more operations, and one or more data variants, as shown in Fig. 7. For example `arith` is a simple language of arithmetic expressions, assembled from *natF*, *boolF* and *compF*. On top of that, we also define an evaluator, a pretty printer and a type checker. Note that for some languages (e.g., `simplenat`), since they have only one kind of value, we only define an evaluator and a pretty printer. We thus obtain 12 languages and 30 operations in total. The complete language `mini-JS` contains all the features and supports all the operations. The reader can refer to our supplementary material for the source code of the case study.

Language	Operations			Data variants					
	eval	print	check	<i>natF</i>	<i>boolF</i>	<i>compF</i>	<i>logicF</i>	<i>varF</i>	<i>funcF</i>
<code>simplenat</code>	✓	✓		✓					
<code>simplebool</code>	✓	✓			✓				
<code>natbool</code>	✓	✓	✓	✓	✓				
<code>varbool</code>	✓	✓			✓			✓	
<code>varnat</code>	✓	✓		✓				✓	
<code>simplelogic</code>	✓	✓			✓		✓		
<code>varlogic</code>	✓	✓			✓		✓	✓	
<code>arith</code>	✓	✓	✓	✓	✓	✓			
<code>arithlogic</code>	✓	✓	✓	✓	✓	✓	✓		
<code>vararith</code>	✓	✓	✓	✓	✓	✓		✓	
<code>vararithlogic</code>	✓	✓	✓	✓	✓	✓	✓	✓	
<code>mini-JS</code>	✓	✓	✓	✓	✓	✓	✓	✓	✓

■ **Figure 7** Overview of the languages assembled.

5.3 Evaluation

To evaluate SEDEL’s implementation of the case study, Figure 8 compares the number of source lines of code (SLOC, lines of code without counting empty lines and comments) for SEDEL’s *modular* implementation with the vanilla *non-modular* AST-based implementations in Haskell. The Haskell implementations are just straightforward AST interpreters, which duplicate code across the multiple language components.

Since SEDEL is a new language, we had to write various code that is provided in Haskell by the standard library, so they are not counted for fairness of comparison. In the left part, for each feature, we count the lines of the algebra interface (number beside the feature name), and the algebras for the operations. In the right part, for each language, we count the lines of ASTs, and those to combine previously defined operations. For example, here is the code that is needed to make the `arith` language.

```

type ArithAlg[E] = NatBoolAlg[E] & CompAlg[E];           -- Object Algebra interface
type Arith = { accept : forall E. ArithAlg[E] → E };   -- AST
evalArith (e : Arith) : IEval =                         -- Evaluator
  e.accept IEval (new[ArithAlg[IEval]] evalNatAlg & evalBoolAlg & evalCompAlg);
ppArith (e : Arith) : IPrint =                          -- Pretty printer
  e.accept IPrint (new[ArithAlg[IPrint]] ppNatAlg & ppBoolAlg & ppCompAlg);
tcArith (e : Arith) =                                   -- Type checker
  e.accept ITC (new[ArithAlg[ITC]] tcNatAlg & tcBoolAlg & tcCompAlg);

```

We only need 8 lines in total: 2 lines for the AST, and 6 lines to combine the operations.

Therefore, the total SLOC of SEDEL’s implementation is the sum of all the lines in the feature and language parts (237 SLOC of all features plus 94 SLOC of ASTs and operations). Although SEDEL is considerably more verbose than a functional language like Haskell, SEDEL’s modular implementation for 12 languages and 30 operations in total reduces approximately 60% in terms of SLOC. The reason is that, the more frequently a feature is reused by other languages directly or indirectly, the more reduction we see in the total SLOC. For example, *natF* is used across many languages. Even though `simplenat` itself *alone* has more SLOC ($40 = 7 + 23 + 7 + 3$) than that of Haskell (which has 33), we still get a huge gain when implementing other languages.

Feature	eval	print	check	Lang name	SEDEL	Haskell	% Reduced
<i>natF</i> (7)	23	7	39	<code>simplenat</code>	3	33	91%
<i>boolF</i> (4)	9	4	17	<code>simplebool</code>	3	16	81%
<i>compF</i> (4)	12	4	20	<code>natbool</code>	5	74	93%
<i>logicF</i> (4)	12	4	20	<code>varbool</code>	4	24	83%
<i>varF</i> (4)	7	4	7	<code>varnat</code>	4	41	90%
<i>funcF</i> (3)	10	3	9	<code>simplelogic</code>	4	28	86%
				<code>varlogic</code>	6	36	83%
				<code>arith</code>	8	94	91%
				<code>arithlogic</code>	8	114	93%
				<code>vararith</code>	8	107	93%
				<code>vararithlogic</code>	8	127	94%
				<code>mini-JS</code>	33	149	78%
Total			237		331	843	61%

■ **Figure 8** SLOC statistics: SEDEL implementation vs vanilla AST implementation.

Finally, we acknowledge the limitation of our case study in that SLOC is just one metric and we have not measured any other metrics. Nevertheless we believe that the case study is already non-trivial in that we need to solve EP. Note that Scala traits alone are not sufficient on their own to solve EP. While there are solutions to EP in both Haskell and Scala, they introduce significant complexity, as explained in Section 5.1.

6 Related Work

Typed First-Class Classes/Mixins/Traits. First-class classes have been used in Racket [26], along with mixin support, and have shown great practical value. For example, DrRacket IDE [24] makes extensive use of layered combinations of mixins to implement text editing features. The topic of first-class classes with static typing has been explored by Takikawa et al. [51] in Typed Racket. They designed a gradual type system that supports first-class classes. Of particular interest is their use of row polymorphism [55] to type mixins. As with our use of disjoint polymorphism, row polymorphism can express constraints on the presence or absence of members. Unlike disjoint polymorphism, row polymorphism prohibits forgetting class members. For example, in SEDEL we can write:

```
foo [A * {bar : String}] (t : Trait[{bar : String} & A]) : Trait[A] = t;
```

where `foo` drops `bar` from its argument trait `t`, which is impossible to express in Typed Racket. Also as we pointed out in Section 3.6, row polymorphism alone cannot express the `merge` function that is able to compose objects of statically unknown types. In this sense, we argue disjoint polymorphism is more powerful than row polymorphism in terms of expressivity. It would be interesting to investigate the relationship between disjoint polymorphism and row polymorphism. We leave it as future work.

More recently, Lee et al. [30] proposed a model for typed first-class classes based on tagged objects. Like our development, the semantics of their source language is defined by a translation into a target language. One notable difference to SEDEL is that they require the use of a variable rather than an expression in the `extends` clause, whereas we do not have this restriction. In their source language, subclasses define subtypes, which limits its applicability to extensible designs. Also their target calculus is significantly more complex than ours due

to the use of dependent function types and dependent sum types. As they admitted, they omit inheritance in their formalization.

Racket also supports a *dynamically-typed model* of first-class traits [26]. However, unlike Racket’s first-class classes and mixins, there’s no type system supporting the use of first-class traits. A key difficulty is *statically* detecting conflicts. As far as we know, SEDEL is the first design for typed first-class traits.

Mixin-Based Inheritance. Bracha and Cook’s seminal paper [16] extends Modula-3 with mixins. Since then, many mixin-based models have been proposed [27, 14, 8]. Mixin-based inheritance requires that mixins are composed linearly, and as such, conflicts are resolved implicitly. In comparison, the trait model in SEDEL requires conflicts to be resolved explicitly. We want to emphasize that conflict detection is essential in expressing composition operators for Object Algebras, without running into ambiguities. Bracha’s Jigsaw [15] formalized mixin composition, along with a rich trait algebra including merge, restrict, select, project, overriding and rename operators. Lagorio et al. [29] proposed FJIG that reformulates Jigsaw constructs in a Java-like setting. Allen et al. [6] described how to add first-class generic types – including mixins – to OO languages with nominal typing. As such, classes and mixins, though they enjoy static typing, are still second-class constructs, and thus their system cannot express dynamic inheritance. Bessai et al. [9] showed how to type classes and mixins with intersection types and Bracha-Cook’s merge operator [16].

Trait-Based Inheritance. Traits were proposed by Schärli et al. [47, 21] as a mechanism for fine-grained code reuse to overcome many limitations of class-based inheritance. The original proposal of traits were implemented in the dynamically-typed class-based language SQUEAK/SMALLTALK. Since then various formalizations of traits in a Java-like (statically-typed) setting have been proposed [25, 46, 50, 34]. In most of the above proposals, trait composition and class-based inheritance live together. SEDEL, in the spirit of *pure trait-based programming languages* [12, 11], embraces traits as the sole mechanism for code reuse. The deviation from traditional class-based inheritance is not only because of its simplicity, but also because we need a very *dynamic* form of inheritance.

Languages with More Advanced Forms of Inheritance. SELF [53] is a dynamically-typed, prototype-based language with a simple and uniform object model. SELF’s inheritance model is typical of what we call *mutable inheritance*, because an object’s parent slot may be assigned new values at runtime. Mutable inheritance is rather unstructured, and oftentimes access to any clashing methods will generate a “messageAmbiguous” error at runtime. Although SEDEL’s dynamic inheritance is not as powerful as mutable inheritance, its static type system can guarantee that no such errors occur at runtime. Eiffel [33] supports a sophisticated class-based multiple inheritance with deep renaming, exclusion and repeated inheritance. Of particular interest is that in Eiffel, name collisions are considered programming errors, and ambiguities must be resolved explicitly by the programmer (by means of renaming). In this regard, SEDEL is quite like Eiffel. However, the type system in SEDEL is more lenient in that two identically named methods with different signatures can coexist. Grace [35, 28] is an object-based language designed for education, where objects are created by *object constructors*. Since Grace has mutable fields, it has to consider many concerns when it comes to inheritance, resulting in a rather complex inheritance mechanism with various restrictions. Since SEDEL is pure, a relatively simple encoding of traits with late binding of `self` suffices for our applications. Grace’s support for multiple inheritance is based on

so-called *instantiable traits*. We believe that there is plenty to be learned from Grace’s design of traits if we want to extend our trait model with features such as mutable state. METAFJIG [48] (an extension of FJIG) supports *dynamic trait replacement* [50, 10, 21], a feature for changing the behavior of an object at runtime by replacing one trait for another.

Module Systems. In parallel to OOP, the ML module system originally proposed by MacQueen [32] also offers powerful support for flexible program construction, data abstraction and code reuse. Mixin modules in the Jigsaw framework [17] provides a suite of operators for adapting and combining modules. The MixML [45] module system incorporates mixin module composition, while retaining the full expressive power of ML modules. Module systems usually put more emphasis on supporting type abstraction. Support for type abstraction adds considerable complexity, which is not needed in SEDEL. SEDEL is focused on OOP and supports, among others, method overriding, self references and dynamic dispatching, which (generally speaking) are all missing features in module systems.

Intersection Types, Polymorphism and Merge Construct. There is a large body of work on intersection types. Here we only talk about work that has direct influences on ours. Dunfield [22] shows significant expressiveness of type systems with intersection types and a merge construct. However his calculus lacks coherence. The limitation was addressed by Oliveira et al. [40], where they introduced the notion of disjointness to ensure coherence. The combination of intersection types, a merge construct and parametric polymorphism, while achieving coherence was first studied in the F_i calculus [7]. F_i serves as the target language of SEDEL. Dynamic inheritance, self-references and abstract methods are all missing from F_i but, as shown in this paper, they can be encoded using an elaboration that employs ideas from Cook and Palsberg’s denotational model of inheritance [19].

7 Conclusion

This paper presents SEDEL: the first design for a polymorphic statically-typed language with first-class traits, supporting dynamic inheritance as well as conventional OO features such as dynamic dispatching and abstract methods. The paper also shows how high-level source language constructs can be elaborated into a core record calculus with disjoint polymorphism. Finally the paper illustrates the applicability of SEDEL by showing greatly improved design patterns such as Object Algebras and Extensible VISITORS, leveraging first-class traits. As for future work, we are interested to study how first-class traits interacts with features such as mutable state and recursive types. For mutable state, one immediate issue of supporting mutation is how it affects the coherence property of F_i , and we foresee major technical challenges to adjust the previous coherence proof. A more powerful proof method such as logical relations [13, 5] may be needed.

References

- 1 Javascript. URL: <https://www.javascript.com/>.
- 2 Python. URL: <https://www.python.org/>.
- 3 Racket. URL: <https://racket-lang.org/>.
- 4 Ruby. URL: <https://www.ruby-lang.org/en/>.
- 5 Amal Jamil Ahmed. *Semantics of types for mutable state*. PhD thesis, Princeton University, 2004.

- 6 Eric E. Allen, Jonathan Bannet, and Robert Cartwright. A first-class approach to genericity. In *Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, 2003.
- 7 João Alpuim, Bruno C. d. S. Oliveira, and Zhiyuan Shi. Disjoint polymorphism. In *European Symposium on Programming (ESOP)*, 2017.
- 8 Davide Ancona, Giovanni Lagorio, and Elena Zucca. Jam—designing a java extension with mixins. In *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2003.
- 9 Jan Bessai, Boris Döder, Andrej Dudenhefner, Tzu-Chun Chen, and Ugo de'Liguoro. Typing classes and mixins with intersection types. In *Workshop on Intersection Types and Related Systems (ITRS)*, 2014.
- 10 Lorenzo Bettini, Sara Capecchi, and Ferruccio Damiani. On flexible dynamic trait replacement for java-like languages. *Science of Computer Programming*, 78(7):907–932, 2013.
- 11 Lorenzo Bettini and Ferruccio Damiani. Xtraitj: Traits for the java platform. *Journal of Systems and Software*, 131:419–441, 2017.
- 12 Lorenzo Bettini, Ferruccio Damiani, Ina Schaefer, and Fabio Strocchio. Traitrecordj: A programming language with traits and records. *Science of Computer Programming*, 78(5):521–541, 2013.
- 13 Xuan Bi, Bruno C. d. S. Oliveira, and Tom Schrijvers. The essence of nested composition. In *European Conference on Object-Oriented Programming*, 2018.
- 14 Viviana Bono, Amit Patel, and Vitaly Shmatikov. A core calculus of classes and mixins. In *European Conference on Object-Oriented Programming (ECOOP)*, 1999.
- 15 Gilad Bracha. *The programming language jigsaw: mixins, modularity and multiple inheritance*. PhD thesis, Dept. of Computer Science, University of Utah, 1992.
- 16 Gilad Bracha and William R. Cook. Mixin-based inheritance. In *Object-oriented Programming, Systems, Languages and Applications (OOPSLA)*, 1990.
- 17 Gilad Bracha and Gary Lindstrom. Modularity meets inheritance. In *International Conference on Computer Languages*, pages 282–290. IEEE Computer Society, 1992.
- 18 William R. Cook. *Anatomy of Programming Languages*. The University of Texas, 2013. URL: <http://www.cs.utexas.edu/~wcook/anatomy/>.
- 19 William R. Cook and Jens Palsberg. A denotational semantics of inheritance and its correctness. In *Object-Oriented Programming: Systems, Languages and Applications (OOPSLA)*, 1989.
- 20 Mario Coppo, Mariangiola Dezani-Ciancaglini, and Betti Venneri. Functional characters of solvable terms. *Mathematical Logic Quarterly*, 27(2-6):45–58, 1981.
- 21 Stéphane Ducasse, Oscar Nierstrasz, Nathanael Schärli, Roel Wuyts, and Andrew P. Black. Traits: A mechanism for fine-grained reuse. *ACM Transactions on Programming Languages and Systems*, 28(2):331–388, mar 2006.
- 22 Joshua Dunfield. Elaborating intersection and union types. *Journal of Functional Programming*, 24(2-3):133–165, 2014.
- 23 Ecma International. *ECMAScript 2015 Language Specification*. Ecma International, Geneva, 6th edition, June 2015. URL: <http://www.ecma-international.org/ecma-262/6.0/ECMA-262.pdf>.
- 24 Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. Drscheme: a programming environment for scheme. *J. Funct. Program.*, 12(2):159–182, 2002.
- 25 Kathleen Fisher and John Reppy. A typed calculus of traits. In *Workshop on Foundations of Object-oriented Programming*, 2004.
- 26 Matthew Flatt, Robert Bruce Findler, and Matthias Felleisen. Scheme with classes, mixins, and traits. In *Programming Languages and Systems (APLAS)*, 2006.
- 27 Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *Principles of Programming Languages (POPL)*, 1998.

- 28 Timothy Jones, Michael Homer, James Noble, and Kim B. Bruce. Object inheritance without classes. In *European Conference on Object-Oriented Programming (ECOOP)*, 2016.
- 29 Giovanni Lagorio, Marco Servetto, and Elena Zucca. Featherweight jigsaw — replacing inheritance by composition in java-like languages. *Information and Computation*, 214:86–111, 2012.
- 30 Joseph Lee, Jonathan Aldrich, Troy Shaw, and Alex Potanin. A theory of tagged objects. In *European Conference on Object-Oriented Programming (ECOOP)*, 2015.
- 31 Roberto E Lopez-Herrejon, Don Batory, and William Cook. Evaluating support for features in advanced modularization technologies. In *European Conference on Object-Oriented Programming (ECOOP)*, 2005.
- 32 David MacQueen. Modules for standard ML. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming - LFP '84*, 1984.
- 33 Bertrand Meyer. Eiffel: programming for reusability and extendibility. *ACM Sigplan Notices*, 22(2):85–94, 1987.
- 34 Oscar Nierstrasz, Stéphane Ducasse, and Nathanael Schärli. Flattening traits. *Journal of Object Technology*, 5(4):129–148, 2006.
- 35 James Noble, Andrew P. Black, Kim B. Bruce, Michael Homer, and Timothy Jones. Grace’s inheritance. *Journal of Object Technology*, 16(2):2:1–35, 2017.
- 36 Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An overview of the scala programming language. Technical report, EPFL Lausanne, Switzerland, 2004.
- 37 Martin Odersky and Matthias Zenger. Scalable component abstractions. In *Object-Oriented Programming: Systems, Languages and Applications (OOPSLA 2005)*, 2005.
- 38 Bruno C. d. S. Oliveira. Modular visitor components: A practical solution to the expression families problem. In *European Conference on Object Oriented Programming (ECOOP)*, 2009.
- 39 Bruno C. d. S. Oliveira and William R. Cook. Extensibility for the masses. In *European Conference on Object-Oriented Programming (ECOOP)*, 2012.
- 40 Bruno C. d. S. Oliveira, Zhiyuan Shi, and João Alpuim. Disjoint intersection types. In *International Conference on Functional Programming (ICFP)*, 2016.
- 41 Bruno C. d. S. Oliveira, Tijs Van Der Storm, Alex Loh, and William R Cook. Feature-oriented programming with object algebras. In *European Conference on Object-Oriented Programming (ECOOP)*, 2013.
- 42 Bruno C. d. S. Oliveira, Meng Wang, and Jeremy Gibbons. The visitor pattern as a reusable, generic, type-safe component. In *Object Oriented Programming: Systems, Languages and Applications (OOPSLA)*, 2008.
- 43 Garrel Pottinger. A type assignment for the strongly normalizable λ -terms. *To HB Curry: essays on combinatory logic, lambda calculus and formalism*, pages 561–577, 1980.
- 44 Tillmann Rendel, Jonathan Immanuel Brachthäuser, and Klaus Ostermann. From object algebras to attribute grammars. In *Object Oriented Programming, Systems Languages and Applications (OOPSLA)*, 2014.
- 45 Andreas Rossberg and Derek Dreyer. Mixin’ up the ML module system. *ACM Transactions on Programming Languages and Systems*, 35(1):1–84, apr 2013.
- 46 Nathanael Scharli, St Ducasse, Roel Wuyts, Andrew Black, et al. Traits: The formal model. *CSETech*, 2003.
- 47 Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P Black. Traits: Composable units of behaviour. In *European Conference on Object-Oriented Programming (ECOOP)*, 2003.

- 48 Marco Servetto and Elena Zucca. A meta-circular language for active libraries. *Science of Computer Programming*, 95:219–253, 2014.
- 49 Yannis Smaragdakis and Don S. Batory. Mixin-based programming in C++. In *Generative and Component-Based Software Engineering (GCSE)*, 2000.
- 50 Charles Smith and Sophia Drossopoulou. Chai: Traits for java-like languages. In Andrew P. Black, editor, *European Conference on Object-Oriented Programming (ECOOP)*, 2005.
- 51 Asumu Takikawa, T. Stephen Strickland, Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. Gradual typing for first-class classes. In *Object-oriented Programming: Systems, Languages and Applications (OOPSLA)*, 2012.
- 52 Mads Torgersen. The Expression Problem Revisited. In *European Conference on Object-Oriented Programming (ECOOP)*, 2004.
- 53 David Ungar and Randall B Smith. Self: the power of simplicity (object-oriented language). In *Comcon Spring'88. Thirty-Third IEEE Computer Society International Conference, Digest of Papers*, 1988.
- 54 Philip Wadler. The expression problem. *Java-genericity mailing list*, 1998.
- 55 Mitchell Wand. Type inference for objects with instance variables and inheritance. *Theoretical aspects of object-oriented programming*, pages 97–120, 1994.
- 56 Pamela Zave. Faq sheet on feature interaction. *Link: <http://www.research.att.com/~pamela/faq.html>*, 1999.

CrySL: An Extensible Approach to Validating the Correct Usage of Cryptographic APIs

Stefan Krüger

Paderborn University, Germany
stefan.krueger@uni-paderborn.de

Johannes Späth

Fraunhofer IEM
johannes.spaeth@iem.fraunhofer.de

Karim Ali

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

Eric Bodden

Paderborn University & Fraunhofer IEM, Germany
eric.bodden@uni-paderborn.de

Mira Mezini

Technische Universität Darmstadt, Germany
mezini@cs.tu-darmstadt.de

Abstract

Various studies have empirically shown that the majority of Java and Android apps misuse cryptographic libraries, causing devastating breaches of data security. It is crucial to detect such misuses early in the development process. To detect cryptography misuses, one must first define secure uses, a process mastered primarily by cryptography experts, and not by developers.

In this paper, we present CRYSL, a definition language for bridging the cognitive gap between cryptography experts and developers. CRYSL enables cryptography experts to specify the secure usage of the cryptographic libraries that they provide. We have implemented a compiler that translates such CRYSL specification into a context-sensitive and flow-sensitive demand-driven static analysis. The analysis then helps developers by automatically checking a given Java or Android app for compliance with the CRYSL-encoded rules.

We have designed an extensive CRYSL rule set for the Java Cryptography Architecture (JCA), and empirically evaluated it by analyzing 10,000 current Android apps. Our results show that misuse of cryptographic APIs is still widespread, with 95% of apps containing at least one misuse. Our easily extensible CRYSL rule set covers more violations than previous special-purpose tools with hard-coded rules, with our tooling offering a more precise analysis.

2012 ACM Subject Classification Security and privacy → Software and application security, Software and its engineering → Software defect analysis, Software and its engineering → Syntax, Software and its engineering → Semantics

Keywords and phrases cryptography, domain-specific language, static analysis

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2018.10

Supplement Material ECOOP Artifact Evaluation approved artifact available at <http://dx.doi.org/10.4230/DARTS.4.3.6>

Funding This work was supported by the DFG through its Collaborative Research Center CROSSING, the project RUNSECURE, by the Natural Sciences and Engineering Research Council



© Stefan Krüger and Johannes Späth and Karim Ali and Eric Bodden and Mira Mezini; licensed under Creative Commons License CC-BY

32nd European Conference on Object-Oriented Programming (ECOOP 2018).

Editor: Todd Millstein; Article No. 10; pp. 10:1–10:27

Leibniz International Proceedings in Informatics



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



cil of Canada, by the Heinz Nixdorf Foundation, a Fraunhofer ATTRACT grant, and by an Oracle Collaborative Research Award.

Acknowledgements We would like to thank the maintainers of AndroZoo for allowing us to use their data set in our evaluation.

1 Introduction

Digital devices are increasingly storing sensitive data, which is often protected using cryptography. However, developers must not only use secure cryptographic algorithms, but also *securely* integrate such algorithms into their code. Unfortunately, prior studies suggest that this is rarely the case. Lazar et al. [22] examined 269 published cryptography-related vulnerabilities. They found that 223 are caused by developers misusing a security library while only 46 result from faulty library implementations. Egele et al. [13] statically analyzed 11,748 Android apps using cryptography-related application interfaces (Crypto APIs) and found 88% of them violated at least one basic cryptography rule. Chatzikonstantinou et al. [12] reached a similar conclusion by analyzing apps manually and dynamically. In 2017, VeraCode listed insecure uses of cryptography as the second-most prevalent application-security issue right after information leakage [11]. Such pervasive insecure use of Crypto APIs leads to devastating vulnerabilities such as data breaches in a large number of applications. Rasthofer et al. [31] showed that *virtually all* smartphone apps that rely on cloud services use hard-coded keys. A simple decompilation gives adversaries access to those keys and to all data that these apps store in the cloud.

Nadi et al. [27] were the first to investigate why developers often struggle to use Crypto APIs. The authors conducted four studies, two of which survey Java developers familiar with the Java Crypto APIs. The majority of participants (65%) found their respective Crypto APIs hard to use. When asked why, participants mentioned the API level of abstraction, insufficient documentation without examples, and an API design that makes it difficult to understand how to properly use the API. A potential long-term solution is to redesign the APIs such that they provide an easy-to-use interface for developers that is secure by default. However, it remains crucial to detect and fix the existing insecure API uses. When asked about what would simplify their API usage, participants wished they had tools that help them automatically detect misuses and suggest possible fixes [27]. Unfortunately, approaches based solely on specification inference and anomaly detection [33] are not viable for Crypto APIs, because – as elaborated above – most uses of Crypto APIs are insecure.

Previous work has tried to detect misuses of Crypto APIs through static analysis. While this is a step in the right direction, existing approaches are insufficient for several reasons. First, these approaches implement mostly lightweight *syntactic checks*, which yield fast analysis times at the cost of exposing a high number of false negatives. Therefore, such analyses fail to warn about many insecure (especially non-trivial) uses of cryptography. For instance, applications using password-based encryption commonly do not clear passwords from heap memory and instead rely on garbage collection to free the respective memory space. Moreover, existing tools cannot easily be extended to cover those rules; instead they have cryptography-specific usage rules *hard coded*. The Java Cryptography Architecture (JCA), the primary cryptography API for Java applications [27], offers a plugin design that enables different providers to offer different crypto implementations through the same API, often imposing slightly different usage requirements on their clients. Hard-coded rules can hardly possibly reflect this diversity.

In this paper, we present CRYSL, a definition language that enables cryptography experts to specify the secure usage of their Crypto APIs in a lightweight special-purpose syntax. We also present a CRYSL compiler that parses and type-checks CRYSL rules and translates them into an efficient, yet precise flow-sensitive and context-sensitive static data-flow analysis. The analysis automatically checks a given Java or Android app for compliance with the encoded CRYSL rules. CRYSL was specifically designed for (and with the help of) cryptography experts. Our approach goes beyond methods that are useful for general validation of API usage (e.g., tpestate analysis [3, 7, 28, 8] and data-flow checks [2, 5]) by enabling the expression of domain-specific constraints related to cryptographic algorithms and their parameters.

To evaluate CRYSL, we built the most comprehensive rule set available for the JCA classes and interfaces to date, and encoded it in CRYSL. We then used the generated static analysis $\text{COGNICRYPT}_{\text{SAST}}$ to scan 10,000 Android apps. We have also modelled the existing hard-coded rules by Egele et al. [13] in CRYSL and compared the findings of the generated static analysis ($\text{COGNICRYPT}_{\text{CL}}$) to those of $\text{COGNICRYPT}_{\text{SAST}}$. Our more comprehensive rule set reports 3× more violations, most of which are true warnings. With such comprehensive rules, $\text{COGNICRYPT}_{\text{SAST}}$ finds at least one misuse in 95% of the apps. $\text{COGNICRYPT}_{\text{SAST}}$ is also highly efficient: for more than 75% of the apps, the analysis finishes in under 3 minutes per app, where most of the time is spent in Android-specific call graph construction.

In summary, this paper presents the following contributions:

- We introduce CRYSL, a definition language to specify correct usages of Crypto APIs.
- We encode a comprehensive specification of correct usages of the JCA in CRYSL.
- We present a CRYSL compiler that translates CRYSL rules into a static analysis to find violations in a given Java or Android app.
- We empirically evaluate $\text{COGNICRYPT}_{\text{SAST}}$ on 10,000 Android apps.

We have integrated $\text{COGNICRYPT}_{\text{SAST}}$ into crypto assistant COGNICRYPT [20] and have open-sourced our implementation and artifacts on GitHub. $\text{COGNICRYPT}_{\text{SAST}}$ is available at <https://github.com/CROSSINGTUD/CryptoAnalysis>. The latest version of the CRYSL rules for the JCA can be accessed at <https://github.com/CROSSINGTUD/Crypto-API-Rules>.

2 Related Work

Before we discuss the details of our approach, we contrast it with the following related lines of work: approaches for specifying API (mis)uses, approaches for inferring API specifications, and previous approaches for detecting misuses of security APIs. Our review of these approaches shows that existing specification languages are not optimally suited for defining misuses of Crypto APIs. Additionally, automated inference of correct uses of Crypto APIs is hard to achieve, and existing tools for detecting misuses of Crypto APIs are limited mainly because they have hard-coded rule sets, and support for the most part lightweight syntactic analyses.

2.1 Languages for Specifying and Checking API Properties

There is a significant body of research on textual specification languages that ensure API properties by means of static data-flow analysis. Tracematches [3] were designed to check tpestate properties defined by regular expressions over runtime objects. Bodden et al. [8, 10]

as well as Naeem and Lhoták [28] present algorithms to (partially) evaluate state matches prior to the program execution, using static analysis.

Martin et al. [24] present Program Query Language (PQL) that enables a developer to specify patterns of event sequences that constitute potentially defective behaviour. A dynamic analysis (i.e., tracematches optimized by a static pre-analysis) matches the patterns against a given program run. A pattern may include a fix that is applied to each match by dynamic instrumentation. PQL has been applied to detecting security-related vulnerabilities such as memory leaks [24], SQL injection and cross-site scripting [23]. Compared to tracematches, PQL captures a greater variety of pattern specifications, at the disadvantage of using only flow-insensitive static optimizations. PQL serves as the main inspiration for the CRYSL syntax. Other languages that pursue similar goals include PTQL [16], PDL [26], and TS4J [9].

We investigated tracematches and PQL in detail, yet found them insufficiently equipped for the task at hand. First, both systems follow a black-list approach by defining and finding incorrect program behaviour. We initially followed this approach for crypto-usage mistakes but quickly discovered that it would lead to long, repetitive, and convoluted misuse-definitions. Consequently, CRYSL defines desired behaviour, which in the case of Crypto APIs leads to more compact specifications. Second, the above languages are general-purpose languages for bug finding, which causes them to miss features essential to define secure usages of Crypto APIs in particular. The strong focus of CRYSL on cryptography allows us to cover a greater portion of cryptography-related problems in CRYSL compared to other languages, while at the same time keeping CRYSL relatively simple. Third, the CRYSL compiler generates state-of-the-art static analyses that were shown to have better performance and precision than other approaches [36], lowering the threat of false warnings.

2.2 Inference/Mining of API-usage specifications

As an alternative to specifying API-usage properties manually, one can attempt to infer them from existing program code. Robillard et al. [34] surveyed over 60 approaches to API property inference. As this survey shows, however, all but two of the surveyed approaches infer patterns from client code (i.e., from applications that use the API in question). When it comes to Crypto APIs, however, past studies have shown that the majority of existing usages of those APIs is, in fact, insecure [13, 12, 35]. Another idea that appears sensible at first sight is to infer correct usage of Crypto APIs from posts on developer portals such as StackOverflow. However, recent studies show that the “solutions” posted there often include insecure code [1].

In result, one can only conclude that automated mining of API-usage specifications is very challenging for Crypto APIs, if it is possible at all. In the future, we plan to investigate a semi-automated approach in which we use automated inference to infer at least partial specifications, but directly in CRYSL, that security experts can then further correct and complete by hand.

2.3 Detecting Misuses of Security APIs

Only few previous approaches specifically address the detection of misuses of *security* APIs. CRYPTOLINT [13] performs a lightweight syntactic analysis to detect violations of exactly six hard-coded usage rules for the JCA in Android apps. Those six rules, while important to obey for security, resemble only a tiny fraction of the rule set we provide in this work. It is also hard to specify and validate new rules using CRYPTOLINT, because they would have to be hard-coded. Unlike CRYPTOLINT, CRYSL is designed to allow crypto experts to also


```
1 SecretKeyGenerator kG = KeyGenerator.getInstance("AES");
2 kG.init(128);
3 SecretKey cipherKey = kG.generateKey();
4
5 String plaintextMSG = getMessage();
6 Cipher ciph = Cipher.getInstance("AES/GCM");
7 ciph.init(Cipher.ENCRYPT_MODE, cipherKey);
8 byte[] cipherText = ciph.doFinal(plaintextMSG.getBytes("UTF-8"));
```

■ **Figure 1** An example illustrating the use of `javax.crypto.KeyGenerator` to implement data encryption in Java.

express comprehensive and complex rules with ease. In Section 8, we extensively compare our tool `COGNICRYPTSAST` to `CRYPTOLINT`.

Another tool that finds misuses of Crypto APIs is Crypto Misuse Analyzer (CMA) [35]. Similar to `CRYPTOLINT`, CMA's rules are hard-coded, and its static analysis is rather basic. Many of CMA's hard-coded rules are also contained in the `CRYSL` rule set that we provide. Unlike `COGNICRYPTSAST`, CMA has been evaluated on a small dataset of only 45 apps.

Chatzikonstantinou et al. [12] manually identified misuses of Crypto APIs in 49 apps and then verified their findings using a dynamic checker. All three studies concluded that at least 88% of the studied apps misuse at least one Crypto API.

None of the previous approaches facilitates rule creation by means of a higher-level specification language. Instead, the rules are hard-coded into each tool, making it hard for non-experts in static analysis to extend or alter the rule set, and impossible to share rules among tools. Moreover, such hard-coded rules are quite restricted, causing the tools to have a very low recall (i.e., missing many actual API misuses). `CRYSL`, on the other hand, due to its Java-like syntax, enables cryptography experts to easily define new rules. The `CRYSL` compiler then automatically transforms those rules into appropriate, highly-precise static-analysis checks. By defining crypto-usage rules in `CRYSL` instead of hard-coding them, one also makes those rules reusable in different contexts.

3 An Example of a Secure Usage of Crypto APIs

Throughout the paper, we will use the code example in Figure 1 to motivate the language features in `CRYSL`. The code in this figure constitutes an API usage that according to the current state of cryptography research can be considered secure. Lines 1–3 generate a 128-bit secret key to use with the encryption algorithm AES. Lines 5–7 use that key to initialize a Java `Cipher` object that encrypts `plaintextMSG`. Since AES encrypts plaintext block by block, it must be configured to use one of several *modes of operation*. The mode of operation determines how to encrypt a block based on the encryption of the preceding block(s). Line 6 configures `Cipher` to use the Galois/Counter Mode (`GCM`) of operation [25].

Although the code example may look straightforward, a number of subtle alterations to the code would render the encryption non-functional or even insecure. First, both `KeyGenerator` and `Cipher` only support a limited choice of encryption algorithms. If the developer passes an unsupported algorithm to either `getInstance` methods, the respective line will throw a runtime exception. Similarly, the design of the APIs separates the classes for key generation and encryption. Therefore, the developer needs to make sure they pass the same algorithm (here "AES") to the `getInstance` methods of `KeyGenerator` and `Cipher`. If the developer does not configure the algorithms as such, the generated key will not fit the encryption algorithm, and the encryption will fail by throwing a runtime exception. None of the existing

```

METHOD :=
  methname(PARAMETERS)

PARAMETERS :=
  varname , PARAMETERS
  varname

TYPES :=
  QualifiedClassName , TYPES
  TYPE

CONSTANTLIST :=
  constant , CONSTANTLIST
  constant

AGGREGATE :=
  label | AGGREGATE
  label ;

EVENT :=
  AGGREGATE
  label : METHOD
  label : varname = METHOD

PREDICATE :=
  predname(PARAMETERS)
  predname(PARAMETERS) after EVENT

PREDICATES :=
  PREDICATE ; PREDICATES

```

A: B = C(D) – a single event with label A consisting of method C, its parameter D, and return object B

■ **Figure 2** Basic CRYSL syntax elements.

tools discussed in Section 2.3 are capable of detecting such functional misuses. Moreover, some supported algorithms are no longer considered secure (e.g., DES or AES/ECB [15]). If the developer selects such an algorithm, the program will still run to completion, but the resulting encryption could easily be broken by attackers. To make things worse, the JCA, the most popular API, offers the insecure ECB mode by default (i.e., when developers request only "AES" without specifying a mode of operation explicitly).

To use Crypto APIs properly, developers generally have to take into consideration two dimensions of correctness: (1) the functional correctness that allows the program to run and terminate successfully and (2) the provided security guarantees. Prior empirical studies have shown that developers, for instance by looking for code examples on web portals such as StackOverflow [14], frequently succeed in obtaining functionally correct code. However, they often fail to obtain a secure use of Crypto APIs, primarily because most code examples on those web portals provide “solutions” that themselves are insecure [14].

SPEC TYPE;

OBJECTS
 OBJECTS :=
 OBJECT ; OBJECTS *A ; B – a list of objects A and B*
 OBJECT ; *A – a list of the single object A*
 OBJECT :=
 TYPE varname *A B – object B of Java type A*

EVENTS
 EVENTS :=
 EVENT ; EVENTS *A ; B – a list of events A and B*
 EVENT ; *A – a list of the single event A*

FORBIDDEN
 FMETHODS :=
 FMETHOD ; FMETHODS *A ; B – a list of forbidden A and B*
 FMETHOD ; *A – a list of the single forbidden method A*
 FMETHOD :=
 methname(TYPES) => label *A(B) => C – a forbidden method named A with parameter of Type B and replacement C*

ORDER
 USAGEPATTERN :=
 USAGEPATTERN , USAGEPATTERN *A , B – A followed by B*
 USAGEPATTERN | USAGEPATTERN *A | B – A or B*
 USAGEPATTERN ? *A? – A is optional*
 USAGEPATTERN * *A* – 0 or more As*
 USAGEPATTERN + *A+ – 1 or more As*
 (USAGEPATTERN) *(A) – grouping*
 AGGREGATE

CONSTRAINTS
 CONSTRAINTS :=
 CONSTRAINT ; CONSTRAINTS
 CONSTRAINT => CONSTRAINT *A => B – A implies B*
 CONSTRAINT
 CONSTRAINT :=
 varname in { CONSTANTLIST } *A in {1, 2} – A should be 1 or 2*

REQUIRES
 REQ_PREDICATES :=
 PREDICATES

ENSURES
 ENS_PREDICATES :=
 PREDICATES

NEGATES
 NEG_PREDICATES :=
 PREDICATES

■ **Figure 3** CRYSL rule syntax in Extended Backus-Naur Form (EBNF) [6].

4 CrySL Syntax

As we discuss in Section 2.2, mining API properties for Crypto APIs is extremely challenging, if possible at all, due to the overwhelming number of misuses one finds in actual applications. Hence, instead of relying on the security of existing usages and examples, we here follow an approach in which cryptography experts define correct API usages manually in a special-purpose language, CRYSL. In this section, we give an overview of the CRYSL syntax elements. A formal treatment of the CRYSL semantics is presented in Section 5. Figure 2 presents the basic syntactic elements of CRYSL, and Figure 3 presents the full syntax for CRYSL rules. Figure 4 shows an abbreviated CRYSL rule for `javax.crypto.KeyGenerator`.

4.1 Design Decisions Behind CrySL

We designed CRYSL specifically with crypto experts in mind, and in fact with the help of crypto experts. This work was carried out in the context of a large collaborative research center than involves more than a dozen research groups involved in cryptography research. As a result of the domain research conducted within this center, we made the following design decisions when designing CRYSL.

White listing. During our domain analysis, we observed that, for the given Crypto APIs, there are many ways they can be misused, but only a few that correspond to correct and secure usages. To obtain concise usage specifications, we decided to design CRYSL to use white listing in most places (i.e., defining secure uses explicitly, while implicitly assuming all deviations from this norm to be insecure).

Typestate and data flow. When reviewing potential misuses, we observed that many of them are related to data flows and typestate properties [38]. Such misuses occur because developers call the wrong methods on the API objects at hand, call them in an incorrect order or miss to call the methods entirely. Data-flow properties are important when reasoning about how certain data is being used (e.g., passwords, keys or seed material).

String and integer constraints. In the crypto domain, string and integer parameters are ubiquitously used to select or parametrize specific cryptography algorithms. Strings are widely used, because they are easily recognizable, configurable, and exchangeable. However, specifying an incorrect string parameter may result in the selection of an insecure algorithm or algorithm combination. Many APIs also use strings for user credentials. Those credentials, passwords in particular, should not be hard-coded into the program's bytecode. A precise specification of correct crypto uses must therefore comprise constraints over string and integer parameters.

Tool-independent semantics. We equipped CRYSL with a tool-independent semantics (to be presented in Section 5). In the future, those semantics will enable us and others to build other or more effective tools for working with CRYSL. For instance, in addition to the static analysis the CRYSL compiler derives from the semantics within this paper, we are currently working on a dynamic checker to identify and mitigate CRYSL violations at runtime.

Our desire to allow crypto experts to easily express secure crypto uses also precludes us from using existing generic definition languages such as Datalog. Such languages, or minor extensions thereof, might have sufficient expressive power. However, following discussions with crypto developers, we had to acknowledge that they are often unfamiliar with those languages' concepts. CRYSL thus deliberately only includes concepts familiar to those developers, hence supporting an easy understanding. We next explain the elements that a typical CRYSL rule comprises.

4.2 Mandatory Sections in a CrySL Rule

To provide simple and reusable constructs, a CRYSL rule is defined on the level of individual classes. Therefore, the rule starts off by stating the class that it is defined for.

In Figure 4, the **OBJECTS** section defines three objects¹ to be used in later sections of the rule (e.g., the object `algorithm` of type `String`). These objects are typically used as parameters or return values in the **EVENTS** section.

The **EVENTS** section defines all methods that may contribute to the successful use of a `KeyGenerator` object, including two *method event patterns* (Lines 17–18). The first pattern matches calls to `getInstance(String algorithm)`, but the second pattern actually matches calls to two overloaded `getInstance` methods:

- `getInstance(String algorithm, Provider provider)`
- `getInstance(String algorithm, String provider)`

The first parameter of all three methods is a `String` object whose value states the algorithm that the key should be generated for. This parameter is represented by the previously defined `algorithm` object. Two of the `getInstance` methods are overloaded with two parameters. Since we do not need to specify the second parameter in either method, we substitute it with an underscore that serves as a placeholder in one combined pattern definition (Line 18). This concept of method event patterns is similar to pointcuts in aspect-oriented programming languages such as AspectJ [19]. For CRYSL, we resort to a more lightweight and restricted syntax as we found full-fledged pointcuts to be unnecessarily complex. Subsequently, the rule defines patterns for the various `init` methods that set the proper parameter values (e.g., `keysize`) and a `generateKey` method that completes the key generation and returns the generated key.

Line 30 defines a usage pattern for `KeyGenerator` using the keyword **ORDER**. The usage pattern is a regular expression of method event patterns that are defined in **EVENTS**. Although each method pattern defines a label to simplify referencing related events (e.g., `g1`, `i2`, and `GenKey`), it is tedious and error-prone to require listing all those labels again in the **ORDER** section. Therefore, CRYSL allows defining *aggregates*. An aggregate represents a disjunction of multiple patterns by means of their labels. Line 19 defines an aggregate `GetInstance` that groups the two `getInstance` patterns. Using aggregates, the usage pattern for `KeyGenerator` reads: there must be exactly one call to one of the `getInstance` methods, optionally followed by a call to one of the `init` methods, and finally a call to `generateKey`.

Following the keyword **CONSTRAINTS**, Lines 33–35 define the constraints for objects defined under **OBJECTS** and used as parameters or return values in the **EVENTS** section. In the abbreviated CRYSL rule in Figure 4, the first constraint limits the value of `algorithm` to "AES" or "Blowfish". For each algorithm, there is one constraint that restricts the possible values of `keysize`.

The **ENSURES** section is the final mandatory construct in a CRYSL rule. It allows CRYSL to support rely/guarantee reasoning. The section specifies predicates to govern interactions between different classes. For example, a `Cipher` object uses a key obtained from a `KeyGenerator`. The **ENSURES** section specifies what a class guarantees, presuming that the object is used properly. For example, the `KeyGenerator` CRYSL rule in Figure 4 ends with the definition of a *predicate* `generatedKey` with the generated key object and its corresponding algorithm as parameters. This predicate may be *required* (i.e., relied on) by the rule for `Cipher` or other classes that make use of such a key through the optional element of the **REQUIRES** block as illustrated in Figure 5.

¹ As the example shows, in CRYSL, **OBJECTS** also comprise primitive values.

```

9  SPEC javax.crypto.KeyGenerator
10
11  OBJECTS
12  java.lang.String algorithm;
13  int keySize;
14  javax.crypto.SecretKey key;
15
16  EVENTS
17  g1: getInstance(algorithm);
18  g2: getInstance(algorithm, _);
19  GetInstance := g1 | g2;
20
21  i1: init(keySize);
22  i2: init(keySize, _);
23  i3: init(_);
24  i4: init(_, _);
25  Init := i1 | i2 | i3 | i4;
26
27  GenKey: key = generateKey();
28
29  ORDER
30  GetInstance, Init?, GenKey
31
32  CONSTRAINTS
33  algorithm in {"AES", "Blowfish"};
34  algorithm in {"AES"} => keySize in {128, 192, 256};
35  algorithm in {"Blowfish"} => keySize in {128, 192, 256, 320, 384,
36  448};
37
38  ENSURES
39  generatedKey[key, algorithm];

```

■ **Figure 4** CRYSL rule for using `javax.crypto.KeyGenerator`.

To obtain the required expressiveness, we have further enriched CRYSL with some simple built-in auxiliary functions. For example, in Figure 5, the function `alg` extracts the encryption algorithm from `transformation` (Line 55). This function is necessary, because `generatedKey` expects only the encryption algorithm as its second parameter, but `transformation` optionally specifies also the mode of operation and padding scheme (e.g., Line 6 in Figure 1). For instance, `alg` would extract "AES" from "AES/GCM" or from "AES/CBC/PKCS5Padding". Table 1 lists all of these functions. Note the last two functions `callTo` and `noCallTo` may seem redundant to the **ORDER** and **FORBIDDEN** (see Section 4.3) sections because they appear to fulfil the same purpose of requiring or forbidding certain method calls. However, these two functions go beyond that because they allow for the specification of conditional forbidden and required methods.

4.3 Optional Sections in a CrySL Rule

A CRYSL rule may contain optional sections that we showcase through the CRYSL rule for `PBEKeySpec`. In Figure 6, the **FORBIDDEN** section specifies methods that must *not* be called, because calling them is always insecure. `PBEKeySpec` derives cryptographic keys from a user-given password. For security reasons, it is recommended to use a cryptographic salt for this operation. However, the constructor `PBEKeySpec(char[] password)` does not allow for a salt to be passed, and the implementation in the default provider does not generate one. Therefore, this constructor should not be called, and any call to it should be flagged. Consequently, the CRYSL rule for `PBEKeySpec` lists it in the **FORBIDDEN** section (Line 72). In

```

39 SPEC javax.crypto.Cipher
40
41 OBJECTS
42   int encmode;
43   java.security.Key key;
44   java.lang.String transformation;
45   ...
46
47 EVENTS
48   g1: getInstance(transformation);
49   ...
50   i1: init(encmode, key);
51
52 ...
53
54 REQUIRES
55   generatedKey[key, alg(transformation)];
56
57 ENSURES
58   encrypted[cipherText, plainText];

```

■ **Figure 5** CRYSL rule for using `javax.crypto.Cipher`.

■ **Table 1** Helper Functions in CRYSL.

Function	Purpose
<code>alg(<i>transformation</i>)</code>	Extract algorithm/mode/padding from <code>transformation</code> parameter of <code>Cipher.getInstance</code> call.
<code>mode(<i>transformation</i>)</code>	
<code>padding(<i>transformation</i>)</code>	
<code>length(<i>object</i>)</code>	Retrieve length of <i>object</i>
<code>nevertypeof(<i>object</i>, <i>type</i>)</code>	Forbid <i>object</i> to be of <i>type</i>
<code>callTo(<i>method</i>)</code>	Require call to <i>method</i>
<code>noCallTo(<i>method</i>)</code>	Forbid call to <i>method</i>

the case of `PBEKeySpec`, there is an alternative secure constructor (Line 68). CRYSL allows one to specify an alternative method event pattern using the arrow notation shown in Line 72. With **FORBIDDEN** events, CRYSL’s language design deviates a bit from its usual white-listing approach. We made this choice deliberately to keep specifications concise. Without explicit **FORBIDDEN** events, one would have to simulate their effect by explicitly listing all events defined on a given type except the one that ought to be forbidden. This would significantly increase the size of CRYSL specifications.

In general, predicates are generated for a particular usage whenever it does not use any **FORBIDDEN** events, its regular **EVENTS** follow the usage pattern defined in the **ORDER** section, and if the usage fulfils all constraints in the **CONSTRAINTS** section of its corresponding rule. `PBEKeySpec`, however, deviates from that standard. The class contains a constructor that receives a user-given password, but the method `clearPassword` deletes that password later, making it no longer accessible to other objects that might use the key-spec. Consequently, a `PBEKeySpec` object fulfils its role after calling the constructor but only until `clearPassword` is called.

To model this usage precisely, CRYSL allows one to specify a method-event pattern using the keyword **after** (Line 80). If the respective method is called, a predicate is generated. Furthermore, CRYSL supports invalidating an existing predicate in the **NEGATES** section

```

59 SPEC javax.crypto.spec.PBEKeySpec
60
61 OBJECTS
62   char[] pw;
63   byte[] salt;
64   int it;
65   int keylength;
66
67 EVENTS
68   create: PBEKeySpec(pw, salt, it, keylength);
69   clear: clearPassword();
70
71 FORBIDDEN
72   PBEKeySpec(char[]) => create;
73   PBEKeySpec(char[],byte[],int) => create;
74
75 ORDER
76   create, clear
77   ...
78
79 ENSURES
80   keyspec[this, keylength] after create;
81
82 NEGATES
83   keyspec[this, _];

```

■ **Figure 6** CRYSL rule for `javax.crypto.spec.PBEKeySpec`.

(Line 83). The last call to be made on a `PBEKeySpec` object is the call to `clearPassword` (Line 76). Additionally, the rule lists the predicate `keyspec[this, _]` within the **NEGATES** block. Semantically, the negation of the predicates means the following. A final event in the **ORDER** pattern, in this case a call to `clearPassword`, invalidates the previously generated `keyspec` predicate(s) for `this`. Section 5.2.2 presents the formal semantics of predicates.

5 CrySL Formal Semantics

5.1 Basic Definitions

A CRYSL rule consists of several sections. The **OBJECTS** section comprises a set of typed variable declarations \mathbb{V} . In the syntax in Figure 3, each declaration $v \in \mathbb{V}$ is represented by the syntax element `TYPE varname`. \mathbb{M} is the set of all resolved method signatures, where each signature includes the method name and argument types. The **EVENTS** section contains elements of the form (m, v) , where $m \in \mathbb{M}$ and $v \in \mathbb{V}^*$. We denote the set of all methods referenced in **EVENTS** by M . The **FORBIDDEN** section lists a set of methods from \mathbb{M} denoted by their signatures; forbidden events cannot bind any variables. The **ORDER** section specifies the usage pattern in terms of a regular expression of labels or aggregates that are in M , i.e., over the defined **EVENTS**. We express this regular expression formally by the equivalent non-deterministic finite automaton (Q, M, δ, q_0, F) over the alphabet M , where Q is a set of states, q_0 is its initial state, F is the set of accepting states, and $\delta : Q \times M \rightarrow \mathcal{P}(Q)$ is the state transition function.

The **CONSTRAINTS** section is a subset of $\mathbb{C} := (\mathbb{V} \rightarrow \mathcal{O} \cup \mathcal{V}) \rightarrow \mathbb{B}$ (i.e., each constraint is a boolean function), where the argument is itself a function that maps variable names in \mathbb{V} to objects in \mathcal{O} or values with primitive types in \mathcal{V} .

A CRYSL rule is a tuple $(T, \mathcal{F}, \mathcal{A}, \mathcal{C})$, where T is the reference type specified by the **SPEC** keyword, $\mathcal{F} \subseteq \mathbb{M}$ is the set of forbidden events, $\mathcal{A} = (Q, M, \delta, q_0, F) \in \mathbb{A}$ is the automaton induced by the regular expression of the **ORDER** section, and $\mathcal{C} \subseteq \mathbb{C}$ is the set of **CONSTRAINTS** that the rule lists. We refer to the set of all CRYSL rules as **SPEC**.

Our formal definition of a CRYSL rule does not contain the sections **REQUIRES**, **ENSURES**, and **NEGATES**. Those sections reason about the interaction of predicates, whose formal treatment we discuss in Section 5.2.2.

5.2 Runtime Semantics

Each CRYSL rule encodes usage constraints to be validated for all runtime objects of the reference type T stated in its **SPEC** section. We define the semantics of a CRYSL rule in terms of an evaluation over a runtime program trace that records all relevant runtime objects and values, as well as all events specified within the rule.

► **Definition 1** (Event). Let \mathcal{O} be the set of all runtime objects and \mathcal{V} the set of all primitive-typed runtime values. An *event* is a tuple $(m, e) \in \mathbb{E}$ of a method signature $m \in \mathbb{M}$ and an *environment* e (i.e., a mapping $\mathbb{V} \rightarrow \mathcal{O} \cup \mathcal{V}$ of the parameter variable names to concrete runtime objects and values). If the environment e holds a concrete object for the **this** value, then it is called the event's *base object*.

► **Definition 2** (Runtime Trace). A *runtime trace* $\tau \in \mathbb{E}^*$ is a finite sequence of events $\tau_0 \dots \tau_n$.

► **Definition 3** (Object Trace). For any $\tau \in \mathbb{E}^*$, a subsequence $\tau_{i_1} \dots \tau_{i_n}$ is called an *object trace* if $i_1 < \dots < i_n$ and all base objects of τ_{i_j} are identical.

Lines 1–2 in Figure 1 result in an object trace that has two events:

$$(m_0, \{algorithm \mapsto \text{"AES"}, \text{this} \mapsto o_{kg}\})$$

$$(m_1, \{algorithm \mapsto \text{"AES"}, keySize \mapsto 128, \text{this} \mapsto o_{kg}\})$$

where m_0 and m_1 are the signatures of the `getInstance` and `init` methods of the `KeyGenerator` class. For static factory methods such as `getInstance`, we assume that **this** is bound to the returned object. We use o_{kg} to denote that the object o is bound to the variable `kg` at runtime.

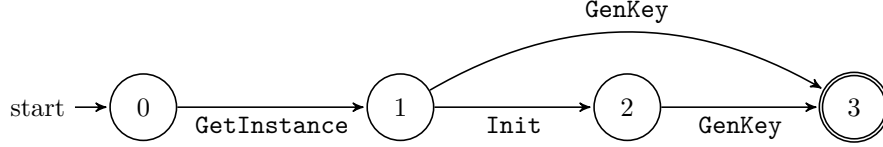
The decision whether a runtime trace τ satisfies a set of CRYSL rules involves two steps. In the first step, individual object traces are evaluated independently of one another. Yet, different runtime objects may still interact with each other. CRYSL rules capture this interaction by means of rely/guarantee reasoning, implemented through predicates that a rule ensures on a runtime object. These interactions between different objects are checked against the specification in a second step by considering the predicates they require and ensure. We first discuss individual object traces in more detail.

5.2.1 Individual Object Traces

The sections **FORBIDDEN**, **ORDER** and **CONSTRAINTS** are evaluated on individual object traces. Figure 7 defines the function sat^o that is true if and only if a given trace τ^o for a runtime object o satisfies its CRYSL rule. This definition of sat^o ignores interactions with other object traces. We will discuss later how such interactions are resolved. In the following, we assume the trace $\tau^o = \tau_0^o, \dots, \tau_n^o$, where $\tau_i^o = (m_i^o, e_i^o)$. To illustrate the computation, we will also refer to our example from Figure 1 and the involved rules of `KeyGenerator` (Figure 4) and `Cipher` (Figure 5). The function sat^o is composed of three sub-functions:

$$\begin{aligned}
 sat^o: \mathbb{E}^* \times \text{SPEC} &\rightarrow \mathbb{B} \\
 [\tau^o, (T^o, \mathcal{F}^o, \mathcal{A}^o, \mathcal{C}^o)] &\rightarrow sat_F^o(\tau^o, \mathcal{F}^o) \wedge \\
 &sat_A^o(\tau^o, \mathcal{A}^o) \wedge \\
 &sat_C^o(\tau^o, \mathcal{C}^o)
 \end{aligned}$$

■ **Figure 7** The function sat^o verifies an individual object trace for the object o .



■ **Figure 8** The state machine for the CRYSL rule in Figure 4 (without an implicit error state).

5.2.1.1 Forbidden Events (sat_F^o)

Given a trace τ^o and a set of forbidden events \mathcal{F} , sat^o ensures that none of the trace events is forbidden.

$$sat_F^o(\tau^o, \mathcal{F}^o) := \bigwedge_{i=0..n} m_i^o \notin \mathcal{F}^o$$

The CRYSL rule for `KeyGenerator` does not list any forbidden methods. Hence, sat^o trivially evaluates to true for object `kG` in Figure 1.

5.2.1.2 Order Errors (sat_A^o)

The second function checks that the trace object is used in compliance with the specified usage pattern (i.e., all methods in the rule are invoked in no other than the specified order). Formally, the sequence of method signatures of the object trace $m^o := m_0^o, \dots, m_n^o$ (i.e., the projection onto the method signatures) must be an element of the language $\mathcal{L}(\mathcal{A}^o)$ that the automaton $\mathcal{A}^o = (Q, \mathbb{M}, \delta, q_0, F)$ of the `ORDER` section induces. By definition of language containment, after the last observed signature of the trace m_n^o , the corresponding state of the automaton must be an accepting state $s \in F$. This definition ignores any variable bindings. They are evaluated in the second step.

$$sat_A^o(\tau^o, \mathcal{A}^o) := m^o \in \mathcal{L}(\mathcal{A}^o)$$

Figure 8 displays the automaton created for `KeyGenerator` using the aggregate names as labels. State 0 is the initial state, and state 3 is the only accepting state. Following the code in Figure 1 for the object `kG` of type `KeyGenerator`, the automaton transitions from state 0 to 1 at the call to `getInstance` (Line 1). With the calls to `init` (Line 2) and `generateKey` (Line 3), the automaton first moves to state 2 and finally to state 3 . Therefore, function sat_A^o evaluates to true for this example.

5.2.1.3 Constraints (sat_C^o)

The validity check of the constraints ensures that all constraints of \mathcal{C} are satisfied. This check requires the sequence of environments (e_0^o, \dots, e_n^o) of the trace τ^o . All objects that are bound

to the variables along the trace must satisfy the constraints of the rule.

$$sat_{\mathcal{C}}^o(\tau^o, \mathcal{C}^o) := \bigwedge_{c \in \mathcal{C}^o, i=0 \dots n} c(e_i^o)$$

To compute $sat_{\mathcal{C}}^o$ for the `KeyGenerator` object `kG` at the call to `getInstance` in Line 1, only the first constraint has to be checked. This is because the corresponding environment e_1^o holds a value only for `algorithm`, and the other two constraints reference other variable names. The evaluation function c returns true if `algorithm` assumes either "AES" or "Blowfish" as its value, which is the case in Figure 1. The computation of $sat_{\mathcal{C}}^o$ for Lines 2–3 works similarly.

5.2.2 Interaction of Object Traces

To define interactions between individual object traces, the **REQUIRES**, **ENSURES**, and **NEGATES** sections allow individual CRYSL rules to reference one another. For a rule for one object to hold at any given point in an execution trace, all predicates that its **REQUIRES** section lists must have been both previously *ensured* (by other specifications) and not *negated*. Predicates are *ensured* (i.e., generated) and *negated* (i.e., killed) by certain events. Formally, a predicate is an element of $\mathbb{P} := \{(name, args) \mid args \in \mathbb{V}^*\}$ (i.e., a pair of a predicate name and a sequence of variable names). Predicates are generated in specific states. Each CRYSL rule induces a function $\mathcal{G}: S \rightarrow \mathcal{P}(\mathbb{P})$ that maps each state of its automaton to the predicate(s) that the state generates.

The predicates listed in the **ENSURES** and **NEGATES** sections may be followed by the term **after** n , where n is a method event pattern label or aggregate. The states that follow the event or aggregate n in the automaton generate the respective predicate. If the term **after** is not used for a predicate, the final states of the automaton generate (or negate) that predicate (i.e., we interpret it as **after** n , where n is an event that leads to a final state).

In addition to states selected as predicate-generating, the predicate is also ensured if the object resides in any state that transitively follows the selected state, unless the states are explicitly (de-)selected for the same predicate within the **NEGATES** section. At any state that generates a predicate, the event driving the automaton into this state binds the variable names to the values that the specification previously collected along its object trace.

Formally, an event $n^o = (m^o, e^o) \in \mathbb{E}$ of a rule r and for an object o ensures a predicate $p = (predName, args) \in \mathbb{P}$ on the objects $e^o \in \mathcal{O}$ if:

1. The method m^o of the event leads to a state s of the automaton that generates the predicate p (i.e., $p \in \mathcal{G}(s)$).
2. The runtime trace of the event's base object o satisfies the function sat^o .
3. All relevant **REQUIRES** predicates of the rule are satisfied at execution of event n^o .

For the `KeyGenerator` object `kG` in Figure 1, a predicate is generated at Line 7 because (1) its automaton transitions to its only predicate-generating state (state 3 of the automaton in Figure 8), (2) sat^o evaluates to true as previously shown for each subfunction and (3) the corresponding CRYSL rule does not require any predicates.

6 Detecting Misuses of Crypto APIs

To detect all possible rule violations, our tool `COGNICRYPTSAST` approximates the evaluation function sat^o using a static data-flow analysis. In a security context, it is a requirement to detect as many misuses as possible. One drawback is the potential for false warnings that

```

84  boolean option1 = isPrime(66); //some non-trivial predicate returning
      false
85  byte[] input = "Message".getBytes("UTF-8");
86
87  String alg = "SHA-256";
88  if (option1) alg = "MD5";
89  MessageDigest md = MessageDigest.getInstance(alg);
90
91  if (input.size() > 0) md.update(input);
92  byte[] digest = md.digest();

```

■ **Figure 9** An example illustrating the usage of `java.security.MessageDigest` in Java.

originate from over-approximations any static analysis requires. In the following, we use the example in Figure 9 to illustrate why and where approximations are required. We will show later in our evaluation that, in practice, our analysis is highly precise and that the chosen approximations rarely actually lead to false warnings.

The code example in Figure 9 implements a hashing operation. By default, the code uses SHA-256. However, if the condition `option1` evaluates to true, MD5 is chosen instead (Line 88). The CRYSL rule for `MessageDigest`, displayed in Figure 10, does not allow the usage of MD5 though, because it is no longer secure [15].

The `update` operation is performed only on non-empty input (Line 91). Otherwise, the call to `update` is skipped and only the call to `digest` is executed, without any input. Although not strictly insecure, this usage does not comply with the CRYSL rule for `MessageDigest`, because it leads to no content being hashed.

To approximate sat_P^o , the analysis must search for possible forbidden events by first constructing a call graph for the whole program under analysis. It then iterates through the graph to find calls to forbidden methods. Depending on the precision of the call graph, the analysis may find calls to forbidden methods that cannot be reached at runtime.

The analysis represents each runtime object o by its allocation site. In our example, allocation sites are `new` expressions and calls to `getInstance` that return an object of a type for which a CRYSL rule exists. For each such allocation site, the analysis approximates sat_A^o by first creating a finite-state machine. $COGNICRYPT_{SAST}$ then evaluates the state machine using a typestate analysis that abstracts runtime traces by program paths. The typestate analysis is path-insensitive, thus, at branch points, it assumes that both sides of the branch may execute. In our contrived example, this feature leads to a false positive: although the condition in Line 91 always evaluates to true, and the call to `update` is never actually skipped, the analysis considers that this may happen, and thus reports a rule violation.

To approximate sat_C^o , we have extended the typestate analysis to also collect potential runtime values of variables along all program paths where an allocated object is used. The constraint solver first filters out all *irrelevant* constraints. A constraint is irrelevant if it refers to one or more variables that the typestate analysis has not encountered. In Figure 10, the rule only includes one internal constraint – on variable `algorithm`. If we add a new internal constraint to the rule about the variable `offset`, the constraint solver will filter it out as irrelevant when analyzing the code in Figure 9 because the only method this variable is associated with (`digest` labelled `d3`) is never called. The analysis distinguishes between never encountering a variable in the source code and not being able to extract the values of a variable. With the same rule and code snippet, if the analysis fails to extract the value for `algorithm`, the constraint evaluates to false. Collecting potential values of a variable over all possible program paths of an allocation site may lead to further imprecision. In our example,

```

93 SPEC java.security.MessageDigest
94
95 OBJECTS
96   java.lang.String algorithm;
97   byte[] input;
98   int offset;
99   int length;
100  byte[] hash;
101  ...
102
103 EVENTS
104   g1: getInstance(algorithm);
105   g2: getInstance(algorithm, _);
106   Gets := g1 | g2;
107   ...
108   Updates := ...;
109
110   d1: output = digest();
111   d2: output = digest(input);
112   d3: digest(hash, offset, length);
113   Digests := d1 | d2 | d3;
114
115   r: reset();
116
117 ORDER
118   Gets, (d2 | (Updates+, Digests)), (r, (d2 | (Updates+, Digests)))*
119
120 CONSTRAINTS
121   algorithm in {"SHA-256", "SHA-384", "SHA-512"};
122
123 ENSURES
124   digested[hash, ...];
125   digested[hash, input];

```

■ **Figure 10** CRYSL rule for `java.security.MessageDigest`.

the analysis cannot statically rule out that `algorithm` may be MD5. The rule forbids the usage of MD5. Therefore, the analysis reports a misuse.

Handling predicates in our analysis follows the formal description very closely. If *sat*^o evaluates to true for a given allocation site, the analysis checks whether all required predicates for the allocation site have been ensured earlier in the program. In the trivial case, when no predicate is required, the analysis immediately ensures the predicate defined in the **ENSURES** section. The analysis constantly maintains a list of all ensured predicates, including the statements in the program that a given predicate can be ensured for. If the allocation site under analysis requires predicates from other allocation sites, the analysis consults the list of ensured predicates and checks whether the required predicate, with matching names and arguments, exists at the given statement. If the analysis finds all required predicates, it ensures the predicate(s) specified in the **ENSURES** section of the rule.

7 Implementation

We have implemented the CRYSL compiler using Xtext [17], an open-source framework for developing domain-specific languages as well as the CRYSL- parameterizable static analysis COGNICRYPT_{SAST}. We have further integrated COGNICRYPT_{SAST} with COGNICRYPT [20], in which it replaces the original code-analysis component.

7.1 CrySL

Given the CRYSL grammar, Xtext provides a parser, type checker, and syntax highlighter for the language. When supplied with a type-safe CRYSL rule, Xtext outputs the corresponding AST, which is then used to generate the required static analysis.

We developed CRYSL rules for all relevant JCA classes in an iterative process. That is, we first worked through the JCA documentation to produce a set of rules and then refined these rules through selective discussions with cryptographers and searching security blogs and forums. In total, we have devised 23 rules covering classes ranging from key handling to digital signing. All rules define a usage pattern. Some classes (e.g. `IvParameterSpec`) contain one call to a constructor only, while others (e.g. `Cipher`) involve almost ten elements with several layers of nesting. Fifteen rules come with parameter constraints, eight of which contain limitations on cryptographic algorithms. The eight rules without parameter constraints are mostly related to classes whose purpose is to set up parameters for specific encryptions (e.g. `GCMParameterSpec`). All rules define at least one **ENSURES** predicate, while only eleven require predicates from other rules. Across all rules, we have only declared two methods forbidden. We do not find this low number surprising as such methods are always insecure and should not at all be part of a security API. If at all, two forbidden methods is too high a number. All rules are available at <https://github.com/CROSSINGTUD/Crypto-API-Rules>.

7.2 CogniCrypt_{sast}

COGNICRYPT_{SAST} consists of several extensions to the program analysis framework Soot [39, 21]. Soot transforms a given Java program into an intermediate representation that facilitates executing intra- and inter-procedural static analyses. The framework provides standard static analyses such as call-graph construction. Additionally, Soot can analyze a given Android app intra-procedurally. Further extensions by FlowDroid [5] enable the construction of Android-specific call graphs that are necessary to perform inter-procedural analysis.

Validating the **ORDER** section in a CRYSL rule requires solving the typestate check sat_{Δ}^o . To this end, we use IDE^{al}, a framework for efficient inter-procedural data-flow analysis [36], to instantiate a typestate analysis. The analysis defines the finite-state machine \mathcal{A}^o to check against and the allocation sites to start the analysis from. From those allocation sites, IDE^{al} performs a flow-, field-, and context-sensitive typestate analysis.

The constraints and the predicates require knowledge about objects and values associated with rule variables at given execution points in the program. The typestate analysis in COGNICRYPT_{SAST} extracts the primitive values and objects on-the-fly, where the latter are abstracted by allocation sites. When the typestate analysis encounters a call site that is referred to in an event definition, and the respective rule requires the object or value of an argument to the call, COGNICRYPT_{SAST} triggers an on-the-fly backward analysis to extract the objects or values that may participate in the call. This on-the-fly analysis yields comparatively high performance and scalability, because many of the arguments of interest are values of type `String` and `Integer`. Thus, using an on-demand computation avoids constant propagation of *all* strings and integers through the program. For the on-the-fly backward analysis, we extended the on-demand pointer analysis Boomerang [37] to propagate both allocation sites and primitive values. Once the typestate analysis is completed, and all required queries to Boomerang are computed, COGNICRYPT_{SAST} solves the internal constraints and predicates using our own custom-made solvers.

COGNICRYPT_{SAST} may be operated as a standalone command line tool. This way, it takes a program as input and produces an error report detailing misuses and their locations.

However, we have further integrated $\text{COGNICRYPT}_{\text{SAST}}$ into COGNICRYPT [20]. COGNICRYPT is a Eclipse plugin, which supports developers in using Crypto APIs by means of scenario-based code generation as well code analysis for Crypto APIs. In this context, COGNICRYPT translates misuses found by $\text{COGNICRYPT}_{\text{SAST}}$ into standard Eclipse error markers.

8 Evaluation

We evaluate our implementation $\text{COGNICRYPT}_{\text{SAST}}$ using the following research questions:

RQ1: What are the precision and recall of $\text{COGNICRYPT}_{\text{SAST}}$?

RQ1: What types of misuses does $\text{COGNICRYPT}_{\text{SAST}}$ find?

RQ1: How fast does $\text{COGNICRYPT}_{\text{SAST}}$ run?

RQ1: How does $\text{COGNICRYPT}_{\text{SAST}}$ compare to the state of the art?

To answer these questions, we applied the generated static analysis $\text{COGNICRYPT}_{\text{SAST}}$ to 10,000 Android apps from the AndroZoo dataset [4] using our full CRYSL rule set for the JCA. We ran our experiments on a Debian virtual machine with sixteen cores and 64 GB RAM. We chose apps that are available in the official Google Play Store and received an update in 2017. This ensures that we report on the most up-to-date usages of Crypto APIs. We make available all artefacts at this Github repository: <https://github.com/CROSSINGTUD/paper-crysl-reproducibility-artefacts>.

8.1 Precision and Recall (RQ1)

Setup

To compute precision and recall, the first two authors manually checked 50 randomly selected apps from our dataset for tpestate errors and violations of internal constraints. To collect this random sample, we implemented a Java program that generates random numbers using `SecureRandom` and retrieved the apps from the corresponding lines in the spreadsheet containing the results of analysing the 10,000 apps. We did not check for unsatisfied predicates or forbidden events, because these are hard to detect manually – while it may seem simple to check for calls to forbidden events, it is non-trivial to determine whether or not such calls reside in dead code. We compare the results of our manual analysis to those reported by $\text{COGNICRYPT}_{\text{SAST}}$. The goal of this evaluation is to compute precision and recall of the analysis implementation in $\text{COGNICRYPT}_{\text{SAST}}$, not the quality of our CRYSL rules. We discuss the latter in Section 8.4. Consequently, we define a false positive to be a warning that should not be reported according to the specified rule, irrespective of that rule’s semantic correctness. Similarly, a false negative would arise if $\text{COGNICRYPT}_{\text{SAST}}$ missed to report a misuse that, according to the CRYSL rule, does exist in the analyzed program.

Results

In the 50 apps we inspected, $\text{COGNICRYPT}_{\text{SAST}}$ detects 228 usages of JCA classes. Table 2 lists the misuses that $\text{COGNICRYPT}_{\text{SAST}}$ finds (156 misuses in total). In particular, $\text{COGNICRYPT}_{\text{SAST}}$ issues 27 tpestate-related warnings, with only 2 false positives. Both arise because the analysis is path-insensitive (Section 6). We further found 4 false negatives that are caused by initializing a `MessageDigest` or a `MAC` object without completing the operation. $\text{COGNICRYPT}_{\text{SAST}}$ fails to find these tpestate errors because the supporting off-the-shelf alias analysis Boomerang times out, causing $\text{COGNICRYPT}_{\text{SAST}}$ to abort the tpestate analysis

■ **Table 2** Correctness of COGNICRYPT_{SAST} warnings.

	Total Warnings	False Positives	False Negatives
Typestate	27	2	4
Constraints	129	19	0
Total	156	21	4

without reporting a warning for the object at hand. A larger timeout or future improvements to the alias analysis Boomerang would avoid this problem.

The automated analysis finds 129 constraint violations. We were able to confirm 110 of them. In the other 19 cases, highly obfuscated code causes the analysis to fail to extract possible runtime values statically. For such values, the constraint solver reports the corresponding constraint as violated. A better handling of such highly obfuscated code can be enabled by techniques complementary to ours. For instance, one could augment COGNICRYPT_{SAST} with the hybrid static/dynamic analysis Harvester [32]. We have also checked the apps for missed constraint violations (false negatives), but were unable to find any.

RQ1: In our manual assessment, the typestate analysis achieves high precision (92.6%) and recall (86.2%). The constraint resolution has a precision of 85.3% and a recall of 100%.

8.2 Types of Misuses (RQ2)

Setup

We report findings obtained by analyzing all our 10,000 Android apps from AndroZoo [4]. We then use the results of our manual analysis (Section 8.1) as a baseline to evaluate our findings on a large scale.

COGNICRYPT_{SAST} detects the usage of at least one JCA class in 8,422 apps. Further investigation unveiled that many of these usages originate from the same common libraries included in the applications. To avoid counting the same crypto usages twice, and to prevent over-counting, we exclude usages within packages `com.android`, `com.facebook.ads`, `com.google` or `com.unity3d` from the analysis.

Results

Excluding the findings in common libraries, COGNICRYPT_{SAST} detects the usage of at least one JCA class in 4,349 apps (43% of the analyzed apps). Most of these apps (95%) contain at least one misuse. Across all apps, COGNICRYPT_{SAST} started its analysis for a total of 40,295 allocation sites (i.e., abstract objects). Of these, a total of 20,426 individual object traces violate at least one part of the specified rule patterns. COGNICRYPT_{SAST} reports typestate errors (**ORDER** section in the rule) for 4,708 objects, and reports a total of 4,443 objects to have unsatisfied predicates (i.e., the object expected a predicate from another object as listed in the **REQUIRES** block of a rule). The analysis also discovered 97 reachable call sites that call forbidden events. The majority of object traces that violate at least one part of a CRYSL rule (54.7%) contradict a constraint listed in the **CONSTRAINTS** section of a rule.

Approximately 86% of these constraint-violations are related to `MessageDigest`. In our manual analysis (see RQ1), 89 of the 110 found constraint violations originated from

usages of MD5 and SHA-1. We expect a similar fraction to also hold for the 11,178 constraint contradictions reported over all 10,000 apps. Many developers still use MD5 and SHA-1, although both are no longer recommended by security experts [15]. COGNICRYPT_{SAST} identifies 1,228 (10.9%) constraint violations related to Cipher usages. In our manual analysis, all misuses of the Cipher class are due to using the insecure algorithm DES or the ECB mode of operation. This result is in line with the findings of prior studies [13, 35, 12].

More than 75% of the tpestate errors that COGNICRYPT_{SAST} issues are caused by misuses of MessageDigest. Our manual analysis attributes this high number to incorrect usages of the method reset(). In addition to misusing MessageDigest, misuses of Cipher contribute 766 tpestate errors. Finally, COGNICRYPT_{SAST} detects 157 tpestate errors related to PBEKeySpec. The ORDER section of the CRYSL rule for PBEKeySpec requires calling clearPassword() at the end of the lifetime of a PBEKeySpec object. We manually inspected 3 of the misuses and observed that the call to clearPassword() is missing in all of them.

Predicates are unsatisfied when COGNICRYPT_{SAST} expects the interaction of multiple object traces but is not able to prove their correct interaction. With 4,443 unsatisfied predicates reported, the number may seem relatively large, yet one must keep in mind that unsatisfied predicates accumulate transitively. For example, if COGNICRYPT_{SAST} cannot ensure a predicate for a usage of IVParameterSpec, it will not generate a predicate for the key object that KeyGenerator generates using the IVParameterSpec object. Transitively, COGNICRYPT_{SAST} reports an unsatisfied predicate also for any Cipher object that relies on the generated key object.

COGNICRYPT_{SAST} also found 97 calls to forbidden methods. Since only two JCA classes require the definition of forbidden methods in our CRYSL rule set (PBEKeySpec and Cipher), we do not find this low number surprising. A manual analysis of a handful of reports suggests that most of the reported forbidden methods originate from calling the insecure PBEKeySpec constructors, as we explained in Section 4.

From the 4,349 apps that use at least one JCA Crypto API, 2,896 apps (66.6%) contain at least one tpestate error, 1,367 apps (31.4%) lack required predicates, 62 apps (1.4%) call at least one forbidden method, and 3,955 apps (90.9%) violate at least one internal constraint. Ignoring the class MessageDigest, and hereby excluding MD5 and SHA-1 constraints, 874 apps still violate at least one constraint in other classes.

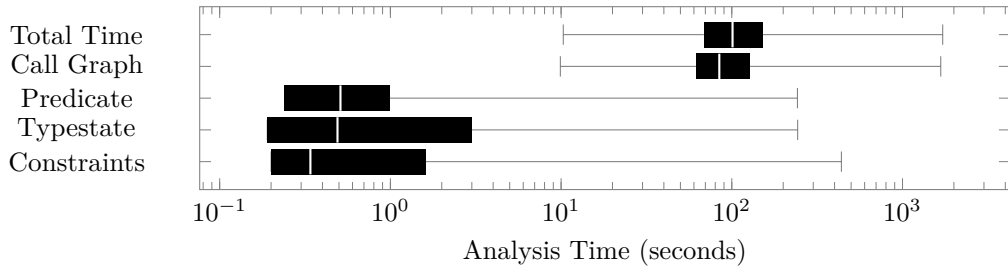
RQ2: Approximately 95% of apps misuse at least one Crypto API. Violating the constraints of MessageDigest is the most common type of misuse.

8.3 Performance (RQ3)

Setup

COGNICRYPT_{SAST} comprises four main phases. It constructs (1) a *call graph* using FlowDroid [5] and then runs the actual analysis (Section 6), which (2) calls the *tpestate analysis* and (3) *constraint analysis* as required, attempting to (4) *resolve all declared predicates*. During the analysis of our dataset, we measured the execution time that COGNICRYPT_{SAST} spent in each phase. We ran COGNICRYPT_{SAST} once per application and capped the time of each run to 30 minutes.

In Section 8.2, we report that COGNICRYPT_{SAST} found usages of the JCA in 4,349 of all 10,000 apps in our dataset. If we include in the reporting those usages that arise from misuses within the common libraries previously excluded (see Section 8.2), this number rises



■ **Figure 11** Analysis time (in log scale) of the individual phases of $\text{COGNICRYPT}_{\text{SAST}}$ when running on the apps that use the JCA.

to 8,422. We include the analysis of the libraries in this part of the evaluation because it helps evaluate the general performance of the analysis in the worst case when whole applications are analyzed.

Results

Figure 11 summarizes the distribution of analysis times for the four phases and the total analysis time across these 8,422 apps. For each phase, the box plot highlights the median, the 25% and 75% quartiles, and the minimal and maximal values of the distribution.

Across the apps in our dataset, there is a large variation in the reported execution time (10 seconds to 28.6 minutes). We attribute this variation to the following reasons. The analyzed apps have varying sizes – the number of reachable methods in the call graph varies between 116 and 16,219 (median: 3,125 methods). The majority of the total analysis time (83%) is spent on call-graph construction. For the remaining three phases of the analysis, the distribution is as follows. Across all apps, the resolution of all declared predicates takes approximately a median of 50 milliseconds, and the typestate analysis phase takes a median of 500 milliseconds. The median for the constraint phase is 350 milliseconds. Therefore, the major bottleneck for the analysis is call-graph construction, a problem orthogonal to the one we address in this work. Our analysis itself is efficient and the overall analysis time is clearly dominated by the runtime of the call-graph construction.

RQ3: On average, $\text{COGNICRYPT}_{\text{SAST}}$ analyzes an app in 101 seconds, with call-graph construction taking most of the time (83%).

8.4 Comparison to Existing Tools (RQ4)

Setup

We compare $\text{COGNICRYPT}_{\text{SAST}}$ to CRYPTOLINT [13], as we explained in Section 2.3 the most closely related tool. Unfortunately, despite contacting the authors we were unable to obtain access to CRYPTOLINT 's implementation. We thus resorted to reimplementing the original rules that are hard-coded in CRYPTOLINT as CRYSL rules. The fact that all CRYPTOLINT rules can be modelled in CRYSL shows its superior expressiveness.

In this section, $\text{RULESET}_{\text{FULL}}$ denotes COGNICRYPT 's comprehensive CRYSL rules that we have created for all the JCA classes, while $\text{RULESET}_{\text{CL}}$ denotes the set of CRYSL rules that we developed to model the original CRYPTOLINT rules. Additionally, $\text{COGNICRYPT}_{\text{SAST}}$ denotes our analysis when it runs using $\text{RULESET}_{\text{FULL}}$, and $\text{COGNICRYPT}_{\text{CL}}$ denotes the analysis when it runs using $\text{RULESET}_{\text{CL}}$.

$\text{RULESET}_{\text{FULL}}$ consists of 23 rules, one for each class of the JCA. $\text{RULESET}_{\text{CL}}$ comprises only six individual rules, and they only use the sections **ENSURES**, **REQUIRES** and **CONSTRAINTS**. In other words, the original hard-coded **CRYPTOLINT** rules do not comprise typestate properties nor forbidden methods. For three out of six rules, we managed to exactly capture the semantics of the hard-coded **CRYPTOLINT** rule in a respective **CRYSL** rule. The remaining three rules (3, 4, and 6 of the original **CRYPTOLINT** rules) cannot be perfectly expressed as a **CRYSL** rule, and our **CRYSL**-based rules over-approximate them instead.

CRYPTOLINT rule 4, for instance, requires salts in **PBEKeySpec** to be non-constant. In **CRYSL**, such a relationship is expressed through predicates. Predicates in **CRYSL**, however, follow a white-listing approach and therefore only model correct behaviour. Therefore, in **CRYSL** we model the **CRYPTOLINT** rule for **PBEKeySpec** in a stricter manner, requiring the salt to be not just non-constant but truly random, i.e., returned from a proper random generator. We followed a similar approach with the other two **CRYPTOLINT** rules that we modelled in **CRYSL**. In result, $\text{RULESET}_{\text{CL}}$ is stricter than the original implementation of **CRYPTOLINT**. In the comparison of $\text{COGNICRYPT}_{\text{SAST}}$ and $\text{COGNICRYPT}_{\text{CL}}$ in terms of their findings, the stricter rules produce more warnings than the original implementation of **CRYPTOLINT**. In our comparison against $\text{COGNICRYPT}_{\text{SAST}}$, this setup favours **CRYPTOLINT** because we assume that these additional findings to be true positives. Both rule sets are available at <https://github.com/CROSSINGTUD/Crypto-API-Rules>.

Results

$\text{COGNICRYPT}_{\text{CL}}$ detects usages of JCA classes in 1,866 Android apps. For these apps, $\text{COGNICRYPT}_{\text{CL}}$ reports 5,507 misuses, only a third of the 20,426 misuses that $\text{COGNICRYPT}_{\text{SAST}}$ identifies using $\text{RULESET}_{\text{FULL}}$, our more comprehensive rule set.

Using $\text{COGNICRYPT}_{\text{CL}}$, all reported warnings are related to 6 classes, compared to 23 classes that are specified in $\text{RULESET}_{\text{FULL}}$. As we have pointed out, **CRYPTOLINT** does not specify any typestate properties or forbidden methods. Hence, $\text{COGNICRYPT}_{\text{CL}}$ does not find the 4,805 warnings that $\text{COGNICRYPT}_{\text{SAST}}$ identifies in these categories using $\text{RULESET}_{\text{FULL}}$. Furthermore, while $\text{COGNICRYPT}_{\text{SAST}}$ reports 11,178 constraint violations with the standard rules, $\text{COGNICRYPT}_{\text{CL}}$ reports only 1,177 constraint violations. Of the 11,178 constraint violations, 9,958 are due to the rule specification for the class **MessageDigest**. **CRYPTOLINT** does not model this class. If we remove these violations, 1,609 violations are still reported by $\text{COGNICRYPT}_{\text{SAST}}$, a total of 432 more than by $\text{COGNICRYPT}_{\text{CL}}$.

We compare our findings to the study by Egele et al. [13] that identifies the use of **ECB** mode as a common misuse of cryptography. In that study, 7,656 apps use **ECB** (65.2% of apps that use **Crypto APIs**). On the other hand, in our study, $\text{COGNICRYPT}_{\text{CL}}$ identified 663 uses of **ECB** mode in 35.5% of apps that use **Crypto APIs**. Although a high number of apps still exhibit this basic misuse, there is a considerable decrease (from 65.2% to 35.5%) compared to the previous study by Egele et al. [13]. Given that all apps in our study must have received an update in 2017, we believe that the decrease of misuses reflects taking software security more seriously in today's app development.

Based on the high precision (92.6%) and recall (96.2%) values discussed in **RQ1**, we argue that $\text{COGNICRYPT}_{\text{SAST}}$ provides an analysis with a much higher recall than **CRYPTOLINT**. Although the larger and more comprehensive rule set, $\text{RULESET}_{\text{FULL}}$, detects more complex misuses, the precise analysis keeps the false-positive rate at a low percentage.

RQ4: The more comprehensive $\text{RULESET}_{\text{FULL}}$ detects 3× as many misuses as **CRYPTOLINT** in almost 4× more JCA classes.

8.5 Threats to Validity

Our ruleset $\text{RULESET}_{\text{FULL}}$ is mainly based on the documentation of the JCA [18]. Although we have significant domain expertise, our CRYSL-rule specifications for the JCA are only as correct as the JCA documentation. Our static-analysis toolchain depends on multiple external components and despite an extensive set of test cases, of course, we cannot fully rule out bugs in the implementation.

Java allows a developer to programmatically select a non-default cryptographic service provider. $\text{COGNICRYPT}_{\text{SAST}}$ currently does not detect such customizations but instead assumes that the default provider is used. This behaviour may lead to imprecise results because our rules forbid certain default values that are insecure for the default provider, but may be secure if a different one is chosen.

9 Conclusion

In this paper, we present CRYSL, a description language for correct usages of cryptographic APIs. Each CRYSL rule is specific to one class, and it may include usage pattern definitions and constraints on parameters. Predicates model the interactions between classes. For example, a rule may generate a predicate on an object if it is used successfully, and another rule may require that predicate from an object it uses. We also present a compiler for CRYSL that transforms a provided ruleset into an efficient and precise data-flow analysis $\text{COGNICRYPT}_{\text{SAST}}$ checking for compliance according to the rules. For ease of use, we have integrated $\text{COGNICRYPT}_{\text{SAST}}$ and with Eclipse crypto assistant COGNICRYPT . Applying $\text{COGNICRYPT}_{\text{SAST}}$, the analysis for our extensive ruleset $\text{RULESET}_{\text{FULL}}$, to 10,000 Android apps, we found 20,426 misuses spread over 95% of the 4,349 apps using the JCA. $\text{COGNICRYPT}_{\text{SAST}}$ is also highly efficient: for more than 75% of the apps the analysis finishes in under 3 minutes, where most of the time is spent in Android-specific call graph construction.

In future work, we plan to address the following challenges. We have developed all the rules used in $\text{COGNICRYPT}_{\text{SAST}}$ ourselves. While we have acquired some deeper familiarity with cryptographic concepts in general and the JCA in particular, we are not cryptographers. Therefore, we are open to and want cryptography experts to correct potential mistakes in our existing rules. We would further encourage domain experts to model their own cryptographic libraries in CRYSL to improve the support in $\text{COGNICRYPT}_{\text{SAST}}$ and, by extension, COGNICRYPT . CRYSL currently only supports a binary understanding of security – a usage is either secure or not. We would like to enhance CRYSL to have a more fine-grained notion of security to allow for more nuanced warnings in $\text{COGNICRYPT}_{\text{SAST}}$. This is challenging because the CRYSL language still ought to be concise. Additionally, CRYSL currently requires one rule per class per JCA provider, because there is no way to express the commonality and variability between different providers implementing the same algorithms, leading to specification overhead. To address this issue, we plan to modularize the language using import and override mechanisms. Moreover, we plan to extend CRYSL to support more complex properties such as using the same cryptographic key for multiple purposes. We will also perform consistency checks for the CRYSL rules. For now, only Xtext-based type checks are performed.

Lastly, we also intend on applying CRYSL in other contexts. One of the authors of this paper has already started to have students implement a dynamic checker to identify and mitigate violations at runtime. While the JCA is indeed the most commonly used Crypto library, other Crypto libraries such as BouncyCastle [29] are being used as well and we will to extend $\text{COGNICRYPT}_{\text{SAST}}$ to support them. Additionally, we will investigate to which

extent CRYSL is applicable to Crypto APIs in other programming languages. At the time of writing, we are exploring CRYSL's compatibility with OpenSSL [30]. We finally aim to examine whether CRYSL is expressive enough to meaningfully specify usage constraints for non-crypto APIs.

References

- 1 Y. Acar, C. Stransky, D. Wermke, C. Weir, M. L. Mazurek, and S. Fahl. Developers need support, too: A survey of security advice for software developers. In *2017 IEEE Cybersecurity Development (SecDev)*, pages 22–26, Sept 2017. doi:10.1109/SecDev.2017.17.
- 2 Dima Alhadidi, Amine Boukhtouta, Nadia Belblidia, Mourad Debbabi, and Prabir Bhattacharya. The dataflow pointcut: a formal and practical framework. In *Proceedings of the 8th International Conference on Aspect-Oriented Software Development, AOSD 2009, Charlottesville, Virginia, USA, March 2-6, 2009*, pages 15–26, 2009.
- 3 Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie J. Hendren, Sascha Kuzins, Ondrej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding trace matching with free variables to aspectj. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16-20, 2005, San Diego, CA, USA*, pages 345–364, 2005. doi:10.1145/1094811.1094839.
- 4 Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. Androzo: collecting millions of android apps for the research community. In *Proceedings of the 13th International Conference on Mining Software Repositories, MSR 2016, Austin, TX, USA, May 14-22, 2016*, pages 468–471, 2016.
- 5 Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick D. McDaniel. Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, pages 259–269, 2014.
- 6 John W. Backus, Friedrich L. Bauer, Julien Green, C. Katz, John McCarthy, Alan J. Perlis, Heinz Rutishauser, Klaus Samelson, Bernard Vauquois, Joseph Henry Wegstein, Adriaan van Wijngaarden, Michael Woodger, and Peter Naur. Revised report on the algorithm language ALGOL 60. *Communications of the ACM*, 6(1):1–17, 1963.
- 7 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, October 21-25, 2007, Montreal, Quebec, Canada*, pages 301–320, 2007. doi:10.1145/1297027.1297050.
- 8 Eric Bodden. Efficient hybrid typestate analysis by determining continuation-equivalent states. In *ICSE '10: International Conference on Software Engineering*, pages 5–14, New York, NY, USA, may 2010. ACM.
- 9 Eric Bodden. TS4J: a fluent interface for defining and computing typestate analyses. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State Of the Art in Java Program analysis, SOAP 2014, Edinburgh, UK, Co-located with PLDI 2014, June 12, 2014*, pages 1:1–1:6, 2014.
- 10 Eric Bodden, Patrick Lam, and Laurie Hendren. Partially evaluating finite-state runtime monitors ahead of time. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 34(2):7:1–7:52, 2012.
- 11 VeraCode (CA). State of software security 2017. <https://info.veracode.com/report-state-of-software-security.html>, 2017.

- 12 Alexia Chatzikonstantinou, Christoforos Ntantogian, Georgios Karopoulos, and Christos Xenakis. Evaluation of cryptography usage in android applications. In *International Conference on Bio-inspired Information and Communications Technologies*, pages 83–90, 2016.
- 13 Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. An empirical study of cryptographic misuse in android applications. In *ACM Conference on Computer and Communications Security*, pages 73–84, 2013.
- 14 Felix Fischer, Konstantin Böttinger, Huang Xiao, Christian Stransky, Yasemin Acar, Michael Backes, and Sascha Fahl. Stack overflow considered harmful? the impact of copy&paste on android application security. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, pages 121–136, 2017.
- 15 German Federal Office for Information Security (BSI). Cryptographic mechanisms: Recommendations and key lengths. Technical Report BSI TR-02102-1, BSI, 2017.
- 16 Simon Goldsmith, Robert O’Callahan, and Alexander Aiken. Relational queries over program traces. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16-20, 2005, San Diego, CA, USA*, pages 385–402, 2005.
- 17 Xtext home page. <http://www.eclipse.org/Xtext/>, 2017.
- 18 Oracle Inc. Java Cryptography Architecture (JCA) Reference Guide. <https://docs.oracle.com/javase/8/docs/technotes/guides/security/crypto/CryptoSpec.html>, 2017.
- 19 Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William Griswold. An overview of aspectj. *ECOOP 2001—Object-Oriented Programming*, pages 327–354, 2001.
- 20 Stefan Krüger, Sarah Nadi, Michael Reif, Karim Ali, Mira Mezini, Eric Bodden, Florian Göpfert, Felix Günther, Christian Weinert, Daniel Demmler, and Ram Kamath. CogniCrypt: Supporting Developers in Using Cryptography. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, pages 931–936, 2017.
- 21 Patrick Lam, Eric Bodden, Ondřej Lhoták, and Laurie Hendren. The Soot framework for Java program analysis: a retrospective. In *Cetus Users and Compiler Infrastructure Workshop (CETUS 2011)*, oct 2011.
- 22 David Lazar, Haogang Chen, Xi Wang, and Nikolai Zeldovich. Why does cryptographic software fail?: a case study and open problems. In *ACM Asia-Pacific Workshop on Systems (APSys)*, pages 7:1–7:7, 2014.
- 23 V. Benjamin Livshits and Monica S. Lam. Finding security vulnerabilities in java applications with static analysis. In *Proceedings of the 14th USENIX Security Symposium, Baltimore, MD, USA, July 31 - August 5, 2005*, 2005.
- 24 Michael C. Martin, V. Benjamin Livshits, and Monica S. Lam. Finding application errors and security flaws using PQL: a program query language. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16-20, 2005, San Diego, CA, USA*, pages 365–383, 2005.
- 25 David A. McGrew and John Viega. The security and performance of the galois/counter mode (GCM) of operation. In *Progress in Cryptology - INDOCRYPT 2004, 5th International Conference on Cryptology in India, Chennai, India, December 20-22, 2004, Proceedings*, pages 343–355, 2004.
- 26 Clint Morgan, Kris De Volder, and Eric Wohlstadter. A static aspect language for checking design rules. In *Proceedings of the 6th International Conference on Aspect-Oriented Software Development, AOSD 2007, Vancouver, British Columbia, Canada, March 12-16, 2007*, pages 63–72, 2007.

- 27 Sarah Nadi, Stefan Krüger, Mira Mezini, and Eric Bodden. Jumping through hoops: why do Java developers struggle with cryptography APIs? In *International Conference on Software Engineering (ICSE)*, pages 935–946, 2016.
- 28 Nomair A. Naeem and Ondrej Lhoták. Typestate-like analysis of multiple interacting objects. In *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 347–366, 2008.
- 29 Legion of the Bouncy Castle Inc. BouncyCastle, 2018. URL: <https://www.bouncycastle.org/java.html>.
- 30 OpenSSL. OpenSSL - Cryptography and SSL/TLS Toolkit, 2018. URL: <https://www.openssl.org/>.
- 31 Siegfried Rasthofer, Steven Arzt, Robert Hahn, Max Kohlhagen, and Eric Bodden. (in)security of backend-as-a-service. In *BlackHat Europe 2015*, 2015.
- 32 Siegfried Rasthofer, Steven Arzt, Marc Miltenberger, and Eric Bodden. Harvesting runtime values in android applications that feature anti-analysis techniques. In *Network and Distributed System Security Symposium (NDSS)*, 2016.
- 33 Martin P. Robillard, Eric Bodden, David Kawrykow, Mira Mezini, and Tristan Ratchford. Automated api property inference techniques. *IEEE TOSEM*, 39(5):613–637, 2013. doi: 10.1109/TSE.2012.63.
- 34 Martin P. Robillard, Eric Bodden, David Kawrykow, Mira Mezini, and Tristan Ratchford. Automated API property inference techniques. *IEEE Transactions on Software Engineering (TSE)*, 39:613–637, 2013.
- 35 Shuai Shao, Guowei Dong, Tao Guo, Tianchang Yang, and Chenjie Shi. Modelling analysis and auto-detection of cryptographic misuse in Android applications. In *International Conference on Dependable, Autonomic and Secure Computing*, pages 75–80, 2014.
- 36 Johannes Späth, Karim Ali, and Eric Bodden. *Ide^{al}*: Efficient and precise alias-aware data-flow analysis. In *2017 International Conference on Object-Oriented Programming, Languages and Applications (OOPSLA/SPLASH)*. ACM Press, 2017. To appear.
- 37 Johannes Späth, Lisa Nguyen Quang Do, Karim Ali, and Eric Bodden. Boomerang: Demand-driven flow- and context-sensitive pointer analysis for java. In *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy*, pages 22:1–22:26, 2016.
- 38 Robert E. Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Software Eng.*, 12(1):157–171, 1986. doi: 10.1109/TSE.1986.6312929.
- 39 Raja Vallée-Rai, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. Optimizing java bytecode using the soot framework: Is it feasible? In *Compiler Construction*, pages 18–34, 2000.

Safe Transferable Regions

Gowtham Kaki

Purdue University, USA¹
gkaki@purdue.edu

G. Ramalingam

Microsoft Research, India
grama@microsoft.com

Abstract

There is an increasing interest in alternative memory management schemes that seek to combine the convenience of garbage collection and the performance of manual memory management in a single language framework. Unfortunately, ensuring safety in presence of manual memory management remains as great a challenge as ever. In this paper, we present a C#-like object-oriented language called BROOM that uses a combination of region type system and lightweight runtime checks to enforce safety in presence of user-managed memory regions called *transferable regions*. Unsafe transferable regions have been previously used to contain the latency due to unbounded GC pauses. Our approach shows that it is possible to restore safety without compromising on the benefits of transferable regions. We prove the type safety of BROOM in a formal framework that includes its C#-inspired features, such as higher-order functions and generics. We complement our type system with a type inference algorithm, which eliminates the need for programmers to write region annotations on types. The inference algorithm has been proven sound and relatively complete. We describe a prototype implementation of the inference algorithm, and our experience of using it to enforce memory safety in dataflow programs.

2012 ACM Subject Classification Software and its engineering → Allocation / deallocation strategies, Software and its engineering → Object oriented languages, Software and its engineering → Data flow languages, Software and its engineering → Software verification and validation

Keywords and phrases Memory Safety, Formal Methods, Type System, Type Inference, Regions, Featherweight Java

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2018.11

Acknowledgements We are grateful to Kapil Vaswani, Dimitrios Vytiniotis, Michael Isard, and Steven Hand for their valuable inputs during the initial stages of this project. We would also like to thank multiple anonymous reviewers for their careful scrutiny and feedback on various versions of this work.

1 Introduction

Computations performed by a network of concurrent communicating actors often involve data transfer between producers and consumers. Consider for example the **SELECT** query operator shown in Fig. 1, which functions as an actor in a dataflow computation involving a network of other such query operators. **SELECT** receives a stream of input messages, each associated with a time window t , processed by method `onReceive`. Each input message contains a list of inputs, each processed by applying a user-defined function to create a corresponding

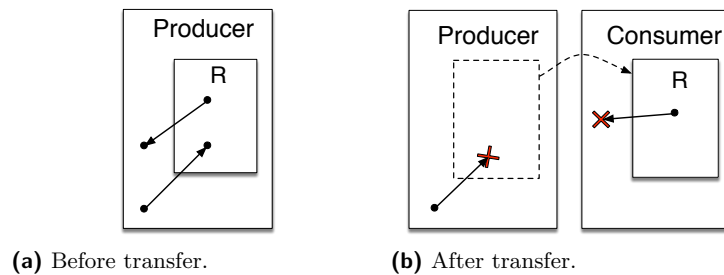
¹ Work done during an internship at Microsoft Research, India.



11:2 Safe Transferable Regions

```
1 class SelectVertex<TIn, TOut> {
2     Func<TIn, TOut> selector;
3     Dictionary<Time, List<TOut>> map;
4     ...
5     void onReceive(Time t, List<TIn> inList) {
6         if (!map.ContainsKey(t)) map[t] = new List<TOut>();
7         foreach (TIn input in inList) {
8             TOut output = selector(input);
9             map[t].add(output); } }
10    void onNotify(Time t) {
11        List<TOut> outList = map[t];
12        map.Remove(t);
13        transfer(successorId, t, outList); }
14 }
```

■ Figure 1 SELECT dataflow operator.



■ Figure 2 References in and out of a transferable region (R) become invalid after transfer.

output. Multiple messages with the same timestamp are permitted and messages with different timestamps may arrive out of order. An invocation of method `OnNotify` indicates that no more input messages with a timestamp t will be subsequently delivered. At this point, the operator completes the processing for time window t and sends a corresponding output message to its successor.

Performance. When the transferred data structures are large, as is the case with many streaming big-data analysis systems, garbage collection overhead becomes significant [7]. Further, in a distributed dataflow system, the GC pause at one node can have a cascading adverse effect on the performance of other nodes [7, 14]: a GC pause at an upstream actor can block downstream actors that are waiting for messages. However, much of the GC overhead results from the collector performing avoidable or unproductive work. For the example in Fig. 1, GC might repeatedly traverse the `map`, although its objects cannot be collected until a suitable timing message arrives. There has been increasing interest in off-heap memory management for better performance in the context of several systems and languages (e.g., Spark [25], Scala [18], Rust). Several systems (e.g., [3]) resort to using “free object pools” to partially alleviate this problem, in the absence of language support.

Safety. Several of these systems are designed to allow the producer and consumer to execute in the same address space or in different address spaces (as determined by the compiler and runtime). We wish to ensure memory safety in the presence of such transferred data. A transfer operation, with no additional checks, may cause memory safety violations, both at the producer of the data structure, and its (possibly remote) consumer. At the producer, any existing references into the transferred data structure become invalid post transfer. If

the data structure contains references to objects outside (the transferred data), then such references become invalid in the context of the consumer. Both scenarios are depicted in Fig. 2. While references from within the transferable data to outside can be disallowed in the interest of safety, references into transferable data needs to be allowed before the data is transferred. Allowing such references is crucial, as any non-trivial program creates temporary references to the internal objects of a data-structure.

Safe Transferable Regions. In this paper, we present a language-based approach to cleanly encapsulate transferable data and a safe and efficient implementation of such data based on regions. A region is a block of memory that is allocated and freed in one shot, often in constant time. A region may contain one or more contiguous range of memory locations, and individual objects may be dynamically allocated within the region over time, while they are deallocated en masse when the region is freed. Thus, a region is a good fit for a transferable data-structure. In Fig. 1, the output to be constructed for each time window t (i.e., `map[t]`) can be a separate region that is allocated when the first message with timestamp t arrives, and deallocated after `map[t]` is transferred in `onNotify`.

The use of regions alleviates the performance concern related to garbage collection described earlier. (It also enables more efficient serialization/deserialization of data-structures across address spaces, which is also a significant performance bottleneck in such systems.) However, manual region memory management introduces memory safety issues such as dangling pointers. We present a single type system that simultaneously addresses this safety concern, as well as those described above related to the use of *transferable data*.

Safe region-based memory management using types was pioneered by Tofte and Talpin [20, 21], who use only lexically scoped regions. At runtime, the set of all regions (existing at a point in time) forms a stack. Thus, the lifetimes of all regions must be well-nested: it is not possible to have two regions whose lifetimes overlap, with neither one's lifetime contained within the other. Unfortunately, the data structures in the above example do not satisfy this restriction (as the output messages for multiple time windows may be simultaneously live, without any containment relation between their lifetimes). We refer to regions with lexically scoped lifetimes as *stack regions* and to regions that do not have such a lexically scoped region as *dynamic* regions.

Our focus, in this paper, is on first-class *memory-safe* dynamic regions that can be safely transferred across address spaces. We refer to such dynamic regions as *transferable* regions. Our approach is based on a variation of ideas introduced by [23, 10] that combine linear types and regions to support dynamic regions. Unlike the prior work, we ensure memory safety in the presence of transferable regions through a *combination of a static typing discipline and lightweight runtime checks in place of linear types*.

Language. The cornerstone of this approach is an `open` lexical block for transferable regions, that “opens” a transferable region and guarantees that the region won't be transferred/freed while it is open. By nesting a Tofte-Talpin style `letregion` lexical block, that delimits the lifetime of a stack region, inside an `open` lexical block for a transferable region, we can guarantee that the transferable region will remain live as long as the stack region is live. We say that the former *outlives* the latter, and any references from the stack region to the transferable region are therefore safe. This is particularly useful, since such stack regions serve as temporary working storage while working with the (opened) transferable regions.

By controlling the outlives relationships between various regions, we only allow safe cross-region references, while prohibiting unsafe ones. In the above example, an outlives

relationship *from* the stack region *to* the transferable region means that the references in that direction are allowed, but not the references in the opposite direction. In contrast, if an **open** block of a transferable region R_0 is nested inside an **open** block of another transferable region R_1 , we do not establish any outlives relationships, thus declaring our intention to not allow any cross-region references between R_0 and R_1 . Finally, we observe that outlives relationships are established based on the lexical structure of the program, hence a static type system can enforce them effectively. By assigning region types to objects, which capture the regions such objects are allocated in, and by maintaining outlives relationships between various regions, we can statically decide the safety of all references in the program.

Type System. Formally, our type system introduces *region parameters* (for both classes and methods) and uses constrained parametric polymorphism over these parameters, where the constraints capture outlives constraints between the region parameters. The type system may be seen as a form of ownership type system [5], with a region being the owner of all objects allocated in that region.

Lightweight Runtime Checks. Ensuring memory safety using this approach requires ensuring that the use of transferable regions satisfies certain temporal properties. Firstly, a transferable region should not be transferred/freed inside an open block of that region (i.e., while it is still open). Secondly, a transferred/freed region should not be opened. These are typestate invariants on the transferable region objects, which are hard to enforce statically in the presence of unrestricted aliasing. Techniques like linear types and unique pointers can be used to restrict aliasing, but the constraints they impose are often hard to program around. We therefore enforce typestate invariants at runtime via lightweight checks. In particular, we define an acceptable state transition discipline for transferable regions (Fig. 4), and check, at runtime, whether a given transition of a transferable region (*e.g.*, from *open* state to *freed* state) is valid or not. The check is lightweight since it only involves checking a single tag that captures the current state. We believe that this is a reasonable choice since regions are coarse-grained objects manipulated infrequently, when compared to the fine-grained objects that are present inside these regions, for which safety is enforced statically.

Type Inference. One of the key contributions of this paper is a type inference algorithm that eliminates the need for users to write region type annotations. The users write programs in the underlying language that provides primitives for the users to manipulate regions (create, open, free, transfer) and to allocate objects in regions, but has no region type annotations.

Our inference algorithm proceeds in three stages: in the first stage, we elaborate the program by introducing region parameters (for classes and methods); in the second stage we generate a set of constraints that must be satisfied for the program to type check; in the third stage, we solve the set of constraints, inferring the preconditions of all classes and methods (in terms of the expected outlives-constraints between the region parameters). We show that the algorithm is sound, and the stages of constraint generation and solving are complete (i.e., the algorithm is complete relative to the first stage of elaboration).

Evaluation. Our work was inspired by [7], which presents evidence that realistic programs can be written using transferable regions and that this can yield significant performance improvements. While [7] addresses the engineering challenges in extending a managed runtime with transferable regions, it adopts an ad hoc approach insofar as language design is concerned, exposing the region functionality through an unsafe API, and in the process

losing the safety guarantees. Our contribution is an alternative approach that is grounded in sound theory and restores the language safety guarantees. The utility of our approach is demonstrated by a prototype implementation of our type inference algorithm that was able to identify unsafe memory accesses among the benchmarks extracted from [7].

Contributions. The paper makes the following contributions:

- We present BROOM, a C#-like typed object-oriented language that supports programmer-managed memory regions. BROOM extends its core language, which includes *lambdas* (higher-order functions) and *generics* (parametric polymorphism), with constructs to create, manage and free static and transferable memory regions. Transferable regions are first-class values in BROOM.
- BROOM is equipped with a region type system that statically guarantees safety of all memory accesses in a well-typed program, provided that certain typestate invariants on regions hold. The latter invariants are enforced via simple runtime checks.
- We define an operational semantics for BROOM, and a type safety result that clearly defines and proves safety guarantees described above.
- We describe a region type inference algorithm for BROOM that (a). completely eliminates the need to annotate BROOM programs with region types, and (b). enables seamless interoperability between region-aware BROOM programs and legacy standard library code that is region-oblivious. Our inference algorithm is based on a novel constraint solver that performs abduction in a partial-order constraint domain to infer weakest solutions to recursive constraints.
- We establish the soundness and relative completeness of the type inference algorithm.
- We describe an implementation of BROOM frontend in OCaml, along with case studies where the region type system was able to identify unsafe memory accesses statically.

2 An Informal Overview of Broom

BROOM enriches a simple object-oriented language (supporting parametric polymorphism and lambdas) with a set of region-specific constructs. In this section, we present an informal overview of these region-specific constructs.

2.1 Using Regions in Broom

Stack Regions. The “`letregion R { S }`” construct creates a new stack region, with a static identifier `R`, whose scope is restricted to the statement `S`. The semantics of `letregion` is similar to Tofte and Talpin [20]’s `letregion` expression: objects can be allocated by `S` in the newly created region while `R` is in scope, but the region and all objects allocated within it are freed at the end of `S`.

Object Allocation. The “`new@R T()`” construct creates a new object of type `T` in the region `R`. The specification of the allocation region `R` in this construct is optional. At runtime, BROOM maintains a stack of *active* regions, and we refer to the region at the top of the stack as the *allocation context*. The statement `new T()` allocates the newly created object in the current allocation context. This is important as it enables BROOM applications to use existing region-oblivious C# libraries, as explained soon.

11:6 Safe Transferable Regions

Transferable Regions. Transferable regions are first class values of BROOM: they are objects of the class `Region`, they are created using the `new` keyword, and can be passed as arguments, stored in data structures, and returned from methods. A transferable region is intended to encapsulate a single data-structure, consisting of a collection of objects with a distinguished root object of some type `T`, which we refer to as the region’s *root* object. The class `Region` is parametric over the type `T` of this root object.

The `Region` constructor takes as a parameter a function that constructs the root object: it creates a new region and invokes this function, with the new region as the allocation context, to create the root object of the region. The following code illustrates the creation of a transferable region, whose root is an object of type `A`.

```
Region<A> rgn = new Region<A>(() => new A())
```

In the above code, `rgn` is called the *handler* to the newly created region, and is required to read the contents of the region, or change its state. The class `Region` offers two methods: a `free` method that deallocates the region (and all the objects allocated within it), and `transfer` method that transfers the region to a (possibly remote) consumer process.

Open and Closed Regions. A transferable region must be explicitly *opened* using BROOM’s `open` construct in order to either read or update or allocate objects in the region. Specifically, the construct “`open rgn as v@R { S }`” does the following: (a). It opens the transferable region handled by `rgn` for allocation (i.e., makes it the current allocation context), (b). binds the identifier `R` to this open region, and (c). initializes the newly introduced local variable `v` to refer to the root object of the region. The `@R` part of the statement is optional and may be omitted. The `open` construct is intended to simplify the problem of ensuring memory safety, as will be explained soon. We refer to a transferable region that has not been opened as a *closed* region. A transferable region can only be transferred or freed when it is in closed state. The acceptable state transition discipline over the lifetime of a transferable region is described in Fig. 4. Enforcement of this discipline is done at runtime.

Opening a region also makes it the current allocation context. Thus, given a `C#` library function `f` (that makes no use of BROOM’s region constructs), opening a region `R` and invoking `f` has the effect that all objects created by this invocation are allocated in the region `R`.

Motivating Example. Fig 3 shows how the motivating example of Fig. 1 can be written in BROOM. The `onReceive` method receives its input message in a *transferred* region (i.e., a *closed* region whose ownership is transferred to the recipient). On Line 7, we create a new region to store the output for time `t`, initializing it to contain an empty list. On Lines 8 and 9, we open the input region `inRgn` followed by creating a new stack region `R0`. The temporary objects created by the iteration on line 10, for example, will be allocated in the stack region `R0` that lives just long enough. We open the desired output region on line 11, so that the new output objects created by the invocation of `selector` on line 12 are allocated in the output region. Finally, the input region is freed on line 14. The output region at `map[t]` stays as long as input messages with timestamp `t` keep arriving. When the timing message for `t` arrives, the `onNotify` method transfers the `outRgn` at `map[t]` to a downstream actor.

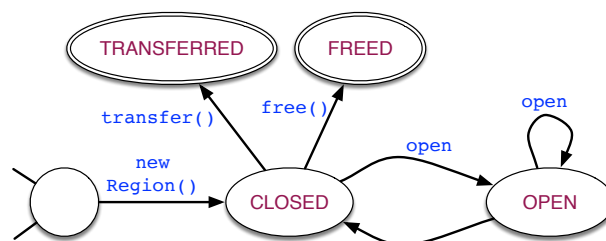
Cloning. Note that in the example from Fig. 3 the object returned by the `selector` (on Line 12) should not contain any references to the input object, since the input region, where the object resides, will be freed at the end of the method. If there is a need for the output

```

1 class SelectVertex<TIn, TOut> {
2   Func<TIn, TOut> selector;
3   Dictionary<Time, Region<List<TOut>>> map;
4   ...
5   void onReceive(Time t, Region<List<TIn>> inRgn){
6     if (!map.ContainsKey(t))
7       map[t] = new Region<List<TOut>> (() => new List<TOut>());
8     open inRgn as inList {
9       letregion R0 {
10        foreach (TIn input in inList) {
11          open map[t] as outList {
12            TOut output = selector(input);
13            outList.add(output); } } } }
14    inRgn.free();
15  }
16  void onNotify(Time t) {
17    Region<List<TOut>> outRgn = map[t];
18    map.Remove(t);
19    outRgn.transfer(successorId);
20  }
21 }

```

■ **Figure 3** SELECT dataflow operator in BROOM.



■ **Figure 4** The lifetime of a dynamic (transferable) region in BROOM.

$\pi \in$	Region identifiers	$\rho \in$	Region variables	$a, b \in$	Type variables
	$m \in$	Method names	$x, y, f \in$	Variables and fields	
$cn \in$	Class names	$::=$	Object Region A B		
$K \in$	FGJ class types	$::=$	$cn\langle\bar{T}\rangle$		
$T \in$	FGJ types	$::=$	$a \mid K \mid \text{unit} \mid \bar{T} \rightarrow T$		
$r \in$	Region annotations	$::=$	$\rho \mid \pi$		
$N \in$	Annotated class types	$::=$	$cn\langle\bar{T}\rangle\langle\bar{r}\rangle$		
$\tau \in$	types	$::=$	$T@r \mid N \mid \text{unit} \mid \langle\bar{\rho} \mid \phi\rangle\bar{\tau} \xrightarrow{r} \tau$		
$C \in$	Class definitions	$::=$	$\text{class } cn\langle\bar{a} \triangleleft \bar{K}\rangle\langle\bar{\rho} \mid \phi\rangle \triangleleft N\{\bar{\tau} \bar{f}; \bar{d}\}$		
$d \in$	Methods	$::=$	$\tau m\langle\bar{\rho} \mid \phi\rangle(\bar{\tau} \bar{x})\{\text{return } e;\}$		
$\phi, \Phi \in$	Region constraints	$::=$	$\text{true} \mid r \succeq r \mid r = r \mid \phi \wedge \phi$		
$e \in$	Expressions	$::=$	$() \mid x \mid e.f \mid e.m\langle\bar{r}\rangle(\bar{e}) \mid \text{new } N(\bar{e})$ $\mid \lambda@r\langle\bar{\rho} \mid \phi\rangle(\bar{x} : \bar{\tau}).e \mid e\langle\bar{r}\rangle(\bar{e}) \mid \text{let } x = e \text{ in } e$ $\mid \text{letregion } \pi \text{ in } e \mid \text{open } e \text{ as } y@r \text{ in } e$		

■ **Figure 5** FEATHERWEIGHT BROOM: syntax.

object to point to subobjects of the input object, such subobjects must be cloned (to copy them from the input region to the output region). Fortunately, BROOM’s region type system (§ 3) is capable of capturing such nuances in the type of `selector` and the type checker will ensure correctness. Furthermore, the type can be automatically inferred by BROOM’s region type inference (§ 4), which can perform the above reasoning on behalf of the programmer.

3 Featherweight Broom

The purpose of BROOM’s region type system is to enforce the key invariant required for memory safety, namely that an object o_1 in a region R_1 contains a reference to an object o_2 in R_2 , only if R_2 is guaranteed to outlive R_1 . While the invariant is easily stated, enforcing it in the presence of first-class dynamic regions, parametric polymorphism (generics), and higher-order functions requires new reasoning principles that we formally develop in this section. We introduce FEATHERWEIGHT BROOM (FB), an explicitly typed core language (with region types) that incorporates the features introduced in the previous section. FEATHERWEIGHT BROOM builds on the Featherweight Generic Java (FGJ) [13] formalism, and reuses notations and various definitions from [13], such as the definition of type well-formedness for the core (region-free) language. (The language used in Section 2 is essentially a version of FGJ without region types and with some syntactic sugar.)

3.1 Syntax

Fig 5 describes the syntax of FB. We refer to the class types of FGJ as *core types*. The following definition of `Pair` class in FB illustrates some of the key elements of the formal language (the symbol \triangleleft should be read *extends*, and the symbol \succeq stands for *outlives*):

```
class Pair<a  $\triangleleft$  Object, b  $\triangleleft$  Object>
    < $\rho_0, \rho_1, \rho_2 \mid \rho_1 \succeq \rho_0 \wedge \rho_2 \succeq \rho_0$ >  $\triangleleft$  Object< $\rho_0$ > {
  a@ $\rho_1$  fst;
  b@ $\rho_2$  snd;
  a@ $\rho_1$  getFst() { return this.fst; }
}
```

Note that we elide showing constructors since they are uninteresting from the type system’s standpoint; their behavior in FB is same as in FGJ.

A class in FB is parametric over zero or more type variables (as in FGJ) as well as one or more region variables ρ . We refer to the first region parameter (ρ_0 in the above example) as the *allocation region* of the class: it serves to identify the region where an instance of the class is allocated². An object in FB can contain fields referring to objects allocated in regions ($\bar{\rho}$) other than its own allocation region (ρ), provided that the former outlive the latter (i.e., $\bar{\rho} \succeq \rho$). In such case, the definition of object’s class needs to be parametric over allocation regions of its fields (i.e., their classes). Furthermore, the constraint that such regions must outlive the allocation region of the class needs to be made explicit in the definition, as the `Pair` class does in the above definition. We say that the `Pair` class exhibits *constrained region polymorphism*.

² In general, $\bar{\rho}$ denotes the sequence $\rho_1\rho_2\dots$ of region parameters of a class or a method. In some cases, we also represent region parameters as $\rho\bar{\rho}$ to clearly distinguish between the allocation region parameter (ρ) and the rest.

To construct objects of the `Pair` class, its type and region parameters need to be instantiated with core types (T) and region annotations³ (r), respectively. For example:

```

letregion  $\pi_0$  in
  let snd = new Object< $\pi_0$ >() in
    letregion  $\pi_1$  in
      let fst = new Object< $\pi_1$ >() in
        let p = new Pair<Object, Object><< $\pi_1$ ,  $\pi_1$ ,  $\pi_0$ > (fst, snd);

```

In the above code, the instantiation of ρ_0 and ρ_1 with π_1 , and ρ_2 with π_0 is allowed because (a) π_0 and π_1 are live during the instantiation, and (b). $\pi_0 \succeq \pi_1$ and $\pi_1 \succeq \pi_1$ (since outlives is reflexive). Observe that the region type of `p` conveys the fact that (a). it is allocated in region π_1 , and (b). it holds references to objects allocated in region π_0 and π_1 . In contrast, if we choose to allocate the `snd` object also in π_1 , then `p` would be contained in π_1 , and its region type would be `Pair`<`Object`, `Object`><< π_1 , π_1 , π_1 >, which we abbreviate as `Pair`<`Object`, `Object`>@ π_1 . In general, we treat $B\langle\bar{T}\rangle@_\pi$ as being equivalent to $B\langle\bar{T}\rangle\langle\bar{\pi}\rangle$. Region annotation on type a , where a is a type variable, assumes the form $a@_\pi$. If a is instantiated with `Pair`<`Object`, `Object`>, the result is the type of a `Pair` object contained in π .

Classes in FB are independently parameterized over types and regions. While this design decision has a downside in that it allows type variables to only denote the types of objects contained in a single region, it yields benefits that outweigh the costs. In particular, it lets us support region-polymorphic higher-order functions as class fields. This allows, for example, a generic class, whose type parameters are a and b , to contain a region-polymorphic function of type $\langle\rho_0, \rho_1, \rho_2 \mid \rho_1 \succeq \rho_0 \wedge \rho_2 \succeq \rho_0\rangle(a@_{\rho_1}, b@_{\rho_2}) \rightarrow \text{Pair}\langle a, b\rangle\langle\rho_0, \rho_1, \rho_2\rangle$ as its field. Such region-polymorphic higher-order fields are used frequently by the dataflow operators, which apply them in the context of various regions (*e.g.*, see Fig. 3). Keeping type and region parameterizations separate also simplifies the type system so that inference becomes practical (Sec. 4). The need for polymorphism at the field level is also why FB treats function closures specially, rather than as objects of type `Func` as in C#.

Like classes, methods can also exhibit constrained region polymorphism. A method definition in FB is necessarily polymorphic over its allocation context (§ 2.1), and optionally polymorphic with respect to the regions containing its arguments (*i.e.*, a method has at least one region parameter). Region parameters, like those on classes, are qualified with constraints (ϕ). If a method is not intended to be polymorphic with respect to its allocation context (for example, if its allocation context needs to be same as the allocation region of its *this* argument), then the required monomorphism can be captured as an equality constraint in ϕ .

FB extends FGJ's expression language with a lambda expression and an application expression ($e\langle\bar{r}\rangle(\bar{e})$) to define and apply functions (lambdas). Functions, like methods, exhibit constrained region polymorphism, as evident in their arrow region type $(\langle\bar{\rho} \mid \phi\rangle\bar{\tau} \xrightarrow{r} \tau)$. A function, like a method, is necessarily polymorphic w.r.t its allocation context. Since a function closure can escape the context in which it is created, it is important to keep track of the region in which it is created in order to avoid unsafe dereferences. The r annotation above the arrow in the arrow type denotes the allocation region of the corresponding closure.

Note that mutable object fields are conspicuously absent from the FB model (and also the FGJ model in general), but this isn't a major shortcoming considering that constructor application, which happens during a new object creation, includes assignments to the object fields (see FGJ [13]). Thus the type system is already obligated to handle unsafe assignments.

³ Region annotations (r) include region variables (ρ) and region identifiers (π). Region identifiers are to region variables, as types (T) are to type variables (a)

3.2 Types and Well-formedness

Well-formedness and typing rules of FEATHERWEIGHT BROOM establish the conditions under which a region type is considered well-formed, and an expression is considered to have a certain region type, respectively. Fig. 6 contains an illustrative subset of such rules⁴. The rules refer to a context (\mathcal{A}) , which is a tuple of:

- A set $(\Delta \in 2^r)$ of regions that are estimated to be live,
- A finite map $(\Theta \in a \mapsto K)$ of type variables to their *bounds*, i.e., classes they are declared to extend (this artifact is inherited from FGJ), and
- A constraint formula (Φ) that captures the outlives constraints on regions in Δ .

In addition, the context for the expression typing judgment also includes (a). a type environment $(\Gamma \in x \mapsto \tau)$ that contains the type bindings for variables in scope, and (b). the region (r) that serves as the allocation context for the expression being type checked. Like the judgments in FGJ [13], all the judgments defined by the rules in Fig. 6 are implicitly parameterized on a class table $(CT \in cn \mapsto D)$ that maps class names to their definitions in FB.

The well-formedness judgment on region types $(\mathcal{A} \vdash \tau \text{ ok})$ makes use of the well-formedness and subtyping judgments on core types. We use a double-piped turnstile (\Vdash) for judgments in FGJ [13], and a simple turnstile (\vdash) for those in FB. The class table $(\llbracket CT \rrbracket)$ for FGJ judgments is derived from FB’s class table (CT) by erasing all region annotations on types, and region arguments in expressions ($\llbracket \cdot \rrbracket$ denotes the region erasure operation). The well-formedness rule for class types $(B \langle \bar{T} \rangle \langle \bar{r} \rangle)$ is responsible for enforcing the safety property that prevents objects from containing unsafe references. It does so by insisting that regions \bar{r} satisfy the constraints (ϕ) imposed by the class on its region parameters. The latter is enforced by checking the validity of ϕ , with actual region arguments substituted for formal region parameters, under the conditions (Φ) guaranteed by the context. The semantics of this sequent is straightforward, and follows directly from the properties of outlives and equality relations. For any well-formed core type T , $T@r$ is a well-formed region type if r is a valid region. The type $\mathbf{Region} \langle T \rangle \langle r \rangle$ is well-formed only if $r = \pi_{\top}$, where π_{\top} is a special immortal region that outlives every other live region. This arrangement allows \mathbf{Region} handlers to be aliased and referenced freely from objects in various regions, regardless of their lifetimes. On the flip side, this also opens up the possibility of references between transferable regions, which become unsafe in context of the recipient’s address space. Fortunately, such references are explicitly prohibited by the type rule of \mathbf{Region} objects, as described below.

The type rules distinguish between the **new** expressions that create objects of the \mathbf{Region} class, and **new** expressions that create objects of other classes. The rule for the latter relies on an auxiliary definition⁵ called **fields** (undefined for \mathbf{Region} class) that returns the sequence of type bindings for fields (instance variables) of a given class type. Like in FGJ, the names and types of a constructor’s arguments in FB are same as the names and types of its class’s fields. The type rule for the field access expression $(e.f_i)$ also uses **fields** and another definition called **bound**, which returns the bound of a type variable (**bound** is an identity function for concrete types).

The type rule for the **new Region** expression expects the \mathbf{Region} class’s constructor to be called with a nullary function that returns a value in its allocation context. This ensures that the value returned by the function stores no references to objects allocated

⁴ Full formal development can be found in the appendix

⁵ All auxiliary definitions we use in this exposition originate from the FGJ calculus.

Type Well-formedness $\boxed{\mathcal{A} \vdash \tau \text{ ok}}$

$$\frac{CT(B) = \text{class } B(\bar{a} \triangleleft \bar{K}) \langle \bar{\rho} \mid \phi \rangle \triangleleft N\{\dots\} \quad \bar{r} \in \Delta \quad \Theta \Vdash B(\bar{T}) \text{ ok} \quad \Phi \vdash [\bar{r}/\bar{\rho}](\phi)}{(\Delta, \Theta, \Phi) \vdash B(\bar{T}) \langle \bar{r} \rangle \text{ ok}}$$

$$\frac{\Theta \Vdash T \text{ ok} \quad r \in \Delta \quad \Theta \Vdash T <: \text{Object}}{(\Delta, \Theta, \Phi) \vdash T@r \text{ ok}} \qquad \frac{\Theta \Vdash T \text{ ok} \quad \mathcal{A} = (\Delta, \Theta, \Phi)}{\mathcal{A} \vdash \text{Region} \langle T \rangle \langle \pi_{\top} \rangle \text{ ok}}$$

Expression Typing $\boxed{\mathcal{A}, \Gamma, r \vdash e : \tau}$

$$\frac{\mathcal{A}, \Gamma, r \vdash \bar{e} : \bar{\tau} \quad \mathcal{A} \vdash N \text{ ok} \quad \text{fields}(N) = \bar{f} : \bar{\tau}}{\mathcal{A}, \Gamma, r \vdash \text{new } N(\bar{e}) : N} \qquad \frac{\mathcal{A}, \Gamma, r \vdash e : \langle \rho \rangle \text{unit} \xrightarrow{r} T@r \rho}{\mathcal{A}, \Gamma, r \vdash \text{new Region} \langle T \rangle \langle \pi_{\top} \rangle (e) : \text{Region} \langle T \rangle \langle \pi_{\top} \rangle}$$

$$\frac{\mathcal{A}, \Gamma, r \vdash e : \tau' \quad \bar{f} : \bar{\tau} = \text{fields}(\text{bound}_{\mathcal{A}, \Theta}(\tau'))}{\mathcal{A}, \Gamma, r \vdash e.f_i : \tau_i} \qquad \frac{\mathcal{A} = (\Delta, \Theta, \Phi) \quad \mathcal{A}' = (\Delta \cup \{\pi\}, \Theta, \Phi \wedge \phi) \quad \pi \notin \Delta \quad \phi = \Delta \succeq \pi \quad \mathcal{A}', \Gamma, \pi \vdash e : \tau \quad \mathcal{A} \vdash \tau \text{ ok}}{\mathcal{A}, \Gamma, r \vdash \text{letregion } \pi \text{ in } e : \tau}$$

$$\frac{\mathcal{A} = (\Delta, \Theta, \Phi) \quad \pi \notin \Delta \quad \mathcal{A}, \Gamma, r \vdash e_a : \text{Region} \langle T \rangle \langle \pi_{\top} \rangle \quad \mathcal{A} \vdash \tau \text{ ok} \quad \mathcal{A}' = (\Delta \cup \{\pi\}, \Theta, \Phi) \quad \Gamma' = \Gamma[y \mapsto T@r \pi] \quad \mathcal{A}', \Gamma', \pi \vdash e_b : \tau}{\mathcal{A}, \Gamma, r \vdash \text{open } e_a \text{ as } y@r \text{ in } e_b : \tau} \qquad \frac{\mathcal{A} = (\Delta, \Theta, \Phi) \quad r \in \Delta \quad \Delta \succeq r \quad \rho, \bar{\rho} \notin \Delta \quad \mathcal{A}' = (\Delta \cup \{\rho, \bar{\rho}\}, \Theta, \Phi \wedge \phi) \quad \Delta \cup \{\rho, \bar{\rho}\} \vdash \phi \text{ ok} \quad \mathcal{A}' \vdash \bar{\tau}^1 \text{ ok} \quad \mathcal{A}' \vdash \tau^2 \text{ ok} \quad \mathcal{A}', \Gamma[\bar{x} \mapsto \bar{\tau}^1], \rho \vdash e : \tau^2}{\mathcal{A}, \Gamma, r \vdash \lambda@r(\rho\bar{\rho} \mid \phi)(\bar{x} : \bar{\tau}^1).e : \langle \rho\bar{\rho} \mid \phi \rangle \bar{\tau}^1 \xrightarrow{r} \tau^2}$$

$$\frac{\mathcal{A} = (\Delta, \Theta, \Phi) \quad \mathcal{A}, \Gamma, r \vdash e_0 : \tau \quad \text{mtype}(m, \text{bound}_{\Theta}(\tau)) = \langle \rho\bar{\rho} \mid \phi \rangle \bar{\tau}^1 \rightarrow \tau^2 \quad r, \bar{r} \in \Delta \quad \mathcal{A} \vdash \langle \rho\bar{\rho} \mid \phi \rangle \bar{\tau}^1 \rightarrow \tau^2 \text{ ok} \quad \mathcal{A}, \Gamma, r \vdash \bar{e} : [r\bar{r}/\rho\bar{\rho}] \bar{\tau}^1 \quad \Phi \vdash [r\bar{r}/\rho\bar{\rho}] \phi}{\mathcal{A}, \Gamma, r \vdash e_0.m(r\bar{r})(\bar{e}) : [r\bar{r}/\rho\bar{\rho}] \tau^2} \qquad \frac{\mathcal{A} = (\Delta, \Theta, \Phi) \quad r', r, \bar{r} \in \Delta \quad \mathcal{A}, \Gamma, r \vdash e : \langle \rho\bar{\rho} \mid \phi \rangle \bar{\tau}^1 \xrightarrow{r'} \tau^2 \quad \Phi \vdash [r\bar{r}/\rho\bar{\rho}] \phi \quad \mathcal{A}, \Gamma, r \vdash \bar{e} : [r\bar{r}/\rho\bar{\rho}] \bar{\tau}^1}{\mathcal{A}, \Gamma, r \vdash e(r\bar{r})(\bar{e}) : [r\bar{r}/\rho\bar{\rho}] \tau^2}$$

Method Well-formedness $\boxed{d \text{ ok in } B}$

$$\frac{\Delta = \{\bar{\rho}, \rho_m, \bar{\rho}_m\} \quad \mathcal{A} = (\Delta, [\bar{a} \mapsto \bar{K}], \phi_m) \quad \Delta \vdash \phi_m \text{ ok} \quad \mathcal{A} \vdash \bar{\tau}^1, \tau^2 \text{ ok} \quad CT(B) = \text{class } B(\bar{a} \triangleleft \bar{K}) \langle \bar{\rho} \mid \phi \rangle \triangleleft N\{\dots\} \quad \mathcal{A}, \Gamma, \rho_m \vdash e : \tau^2 \quad \Gamma = \cdot[\bar{x} \mapsto \bar{\tau}^1][\text{this} \mapsto B(\bar{a}) \langle \bar{\rho} \rangle] \quad \mathcal{A} \vdash \text{override}(m, N, \langle \rho_m \bar{\rho}_m \mid \phi_m \rangle \bar{\tau}^1 \rightarrow \tau^2)}{\tau^2 m \langle \rho_m \bar{\rho}_m \mid \phi_m \rangle (\bar{\tau}^1 \bar{x}) \{ \text{return } e; \} \text{ ok in } B}$$

■ **Figure 6** FEATHERWEIGHT BROOM: select type rules.

elsewhere, including the top region (π_{\top}), thus preventing cross-region references originating from transferable regions. The body of the function might however create new regions while execution, but this is not a problem as long as such regions, and objects allocated in them, don't find their way into the result of its evaluation.

The type rule for the `letregion` expression requires that the static identifier introduced by the expression be unique under the current context (i.e., $\pi \notin \Delta$). This condition is needed in order to prevent the new region from incorrectly assuming existing outlives relationships on an eponymous region. The expression (e) under `letregion` is typechecked with the new

11:12 Safe Transferable Regions

region as its allocation context, under the assumption that the region is live ($\Delta \cup \{\pi\}$) and that it is outlived by all existing live regions ($\Delta \succeq \pi$). The result of a `letregion` expression must have a type that is well-formed under a context not containing the new region. This ensures that the value obtained by evaluating a `letregion` expression contains no references to the temporary objects inside the region.

The rule for the `open` expression, unlike the rule for `letregion`, does not introduce any outlives relationship between the newly opened region and any pre-existing region while checking the type of the expression (e) under `open`. This prevents new objects allocated inside the transferable region from storing references to those outside. The newly opened region becomes the allocation context for e , which is type checked under an environment (Γ) extended with the type binding for the root object.

The type rule for the lambda expression typechecks the lambda-bound expression (e) under an extended type environment containing bindings for function's arguments, assuming that region parameters are live, and that declared constraints over region parameters hold. The constraints (ϕ) are required to be well-formed under Δ extended with the function's region parameters ($\rho\bar{\rho}$). Unlike a typical object, a function closure might capture references that may become unsafe if the closure escapes its allocation context. FB prevents this scenario by requiring the function closure to always be allocated in the current allocation context (r).

The type rule for method application uses an auxiliary definition `mtype` that gives the type of a class method. Method application involves instantiation of method's region parameters, and the type rule requires the first region parameter to be instantiated with the current allocation context (r). The type rule for function application does similarly. This requirement ensures the safety of closures as demonstrated by the following example.

Consider a method m with two region parameters - ρ_0 and ρ_1 (i.e., $m\langle\rho_0, \rho_1\rangle(\dots)$). Suppose the method immediately returns a function closure that contains a reference to (an object in) ρ_1 . Since function closures are always allocated in the current allocation context, the closure is allocated in ρ_0 , the allocation context for the method body (the rule for methods discussed below). Now, consider the following use of the method:

```
letregion R0 in
  let f = letregion R1 in m<R0, R1>()
in f()
```

Observe that the first region parameter of m is instantiated with $R0$, while the allocation context for the method call is $R1$. The function f returned by m stores a reference to $R1$, which is unsafe when f is finally called. The type system fortunately disallows this scenario by enforcing certain restrictions during method and function applications as described above.

The method well-formedness rule makes use of an auxiliary definition `override` that judges whether the current method is a valid overriding of any eponymous method from the super class. The type environment is extended with a binding that binds the `this` keyword to the type of the current class. Note that the type of the current method as accessed via `this` (i.e., `this.m`) is region-parametric, thus admitting region-polymorphic recursion (i.e., recursive call to m can have different region arguments than m).

3.3 Operational Semantics and Type Safety

The operational semantics of FB describes a non-trivial runtime component that introduces memory regions and performs run-time verification of their typestate. Fig. 7 shows the additional language constructs of FB that manifest only at run-time. A location (1) abstracts

l	\in	Memory locations	
r	\in	Region annotations	$::= l \mid \dots$
s	\in	Region Typestate	$::= \square \mid \blacksquare \mid \times$
e	\in	Expressions	$::= \text{letd } l \text{ in } e \mid \text{opened } l(s) \text{ in } e \mid \perp \mid \dots$

■ **Figure 7** FEATHERWEIGHT BROOM: extended syntax to accommodate run-time constructs.

all the memory locations associated with a region (i.e., each memory region is associated with a single location). A transferable region can be in one of three possible states: closed (\square), open/live (\blacksquare), and transferred/freed (\times). Constructs `letd` and `opened` are the run-time manifestations of `letregion` and `open`; `letregion` reduces to `letd` while allocating a (static) region, and `open` reduces to `opened` while opening a (transferable) region. An `opened` expression is tagged with the typestate (s) of the transferable region before it is opened (as Fig. 4 shows, a transferable region can be `open`'d from either an open state or a closed state). Special value \perp denotes an exception.

Operational semantics of FB defines a four-place small-step reduction relation of the form shown below:

$$(e, \Sigma) \longrightarrow (e', \Sigma')$$

The typestate of regions is tracked by a finite map (Σ) from locations to typestates (Fig. 4). The reduction relation relates an expression (e) and a typestate map (Σ) to a reduced expression (e') and an updated typestate map (Σ'). The semantics gets “stuck” if e attempts to access an object whose allocation region is not present in Δ , or if e tries to `open` a transferable region that is not mapped to an appropriate typestate by Σ . On the other hand, if e attempts to commit an operation on a `Region` object that is not sanctioned by the transition discipline in Fig. 4, then it raises an exception value (\perp). An illustrative subset of operational semantics that formalize the intuitions described above is shown in Fig. 8. Fig. 15 of the appendix contains rest of the rules.

To help state the type safety theorem, we define the syntactic class of (runtime) values:

$$v \in \text{values} ::= \text{new } B\langle\bar{T}\rangle\langle\bar{l}\rangle(\bar{v}) \mid \lambda@l\langle\bar{\rho}\rangle\langle\phi\rangle(\bar{\tau} \bar{x}).e \mid \text{new Region}\langle T\rangle\langle l_{\top} l\rangle(v)$$

The first two forms are obtained by using locations for region annotations in `new` and lambda expressions. The last form is the value that `new Region` expressions gets evaluated to; the location l_{\top} stands for the region π_{\top} , and l is the location of the newly allocated transferable region. The following type safety theorem shows that a well-typed program will never attempt to dereference an “invalid” reference (a reference to an object in a region that has been transferred or freed):

► **Theorem 1** (Type Safety). $\forall e, \tau, \Delta, \Sigma$, such that $\text{consistent}(\Delta, \Sigma)$ and $\Delta \vdash \Phi \text{ ok}$, if $(\Delta, \cdot, \Phi), \cdot \vdash e : \tau$, then either e is a value, or e raises an exception $((e, \Sigma) \longrightarrow \perp)$, or there exists an e' and a Σ' such that $\text{consistent}(\Delta, \Sigma')$ and $(e, \Sigma) \longrightarrow (e', \Sigma')$ and $(\Delta, \cdot, \Phi), \cdot \vdash e' : \tau$.

The relation `consistent` relates Δ and Σ only if both make consistent assumptions about liveness of regions. Since Δ is consistent with both Σ and Σ' , the theorem also captures the key property of operational semantics that no live region is ever freed or transferred.

11:14 Safe Transferable Regions

$$\boxed{(e, \Sigma) \longrightarrow (e', \Sigma')}$$

$$\begin{array}{c}
\text{[LETREGIONBEGIN]} \quad \frac{1 \notin \text{dom}(\Sigma) \quad \Sigma' = \Sigma[1 \mapsto \blacksquare]}{(\text{letregion } r \text{ in } e, \Sigma) \longrightarrow (\text{letd } 1 \text{ in } [1/r]e, \Sigma')} \\
\text{[LETREGION]} \quad \frac{(e, \Sigma) \longrightarrow (e', \Sigma')}{(\text{letd } 1 \text{ in } e, \Sigma) \longrightarrow (\text{letd } 1 \text{ in } e', \Sigma')} \\
\text{[LETREGIONEND]} \quad \frac{\Sigma' = \Sigma[1 \mapsto \times]}{(\text{letd } 1 \text{ in } v, \Sigma) \longrightarrow (v, \Sigma')} \\
\text{[NEWREGION]} \quad \frac{1 \notin \text{dom}(\Sigma) \quad \Sigma' = \Sigma[1 \mapsto \square]}{(\text{new Region } \langle T \rangle \langle \pi_{\top} \rangle (v), \Sigma) \longrightarrow (\text{new Region } \langle T \rangle \langle 1_{\top} 1 \rangle (v(1)()), \Sigma')} \\
\text{[NEWREGION]} \quad \frac{(e, \Sigma[1 \mapsto \blacksquare]) \longrightarrow (e', \Sigma')}{(\text{new Region } \langle T \rangle \langle 1_{\top} 1 \rangle (e), \Sigma) \longrightarrow (\text{new Region } \langle T \rangle \langle 1_{\top} 1 \rangle (e'), \Sigma')} \\
\text{[OPEN]} \quad \frac{v_a = \text{new Region } \langle T \rangle \langle 1_{\top} 1 \rangle (v) \quad \Sigma(1) = \square \text{ or } \Sigma(1) = \blacksquare \quad \Sigma' = \Sigma[1 \mapsto \blacksquare]}{(\text{open } v_a \text{ as } x@r \text{ in } e_b, \Sigma) \longrightarrow (\text{opened } 1(\Sigma(1)) \text{ in } [v/x][1/r]e_b, \Sigma')} \\
\text{[OPENED]} \quad \frac{(e, \Sigma) \longrightarrow (e', \Sigma')}{(\text{opened } 1(s) \text{ in } e, \Sigma) \longrightarrow (\text{opened } 1(s) \text{ in } e', \Sigma')} \\
\text{[OPENEND]} \quad \frac{\Sigma' = \Sigma[1 \mapsto s]}{(\text{opened } 1(s) \text{ in } v, \Sigma) \longrightarrow (v, \Sigma')} \\
\text{[OPENTRANSFERRED]} \quad \frac{v_a = \text{new Region } \langle T \rangle \langle 1_{\top} 1 \rangle (v) \quad \Sigma(1) \neq \square \text{ and } \Sigma(1) \neq \blacksquare}{(\text{open } v_a \text{ as } x@r \text{ in } e_b, \Sigma) \longrightarrow \perp} \\
\text{[TRANSFER]} \quad \frac{\Sigma(1) = \square \quad \Sigma' = \Sigma[1 \mapsto \times]}{((\text{new Region } \langle T \rangle \langle 1_{\top} 1 \rangle (v)).\text{transfer}(\dots), \Sigma) \longrightarrow ((), \Sigma')} \\
\text{[TRANSFEROPENED]} \quad \frac{\Sigma(1) = \blacksquare}{((\text{new Region } \langle T \rangle \langle 1_{\top} 1 \rangle (v)).\text{transfer}(\dots), \Sigma) \longrightarrow \perp}
\end{array}$$

■ **Figure 8** FEATHERWEIGHT BROOM: a select subset of operational semantics.

Integrating regions with GC heap. FB stores the region handlers in a distinguished top region, which abstracts the GC heap. Type system prevents transferable regions from storing references into the GC heap. A static region however can store references to region handles, which need to be taken into account while GC'ing the heap. Traversing the region memory to identify references into the GC heap unfortunately defeats the purpose of regions. The solution we adopt is to disable collecting region handles as long as any static region is open. Since static regions are intended to store temporary objects while populating a dynamic region, their lifetimes are short enough to let region handles to be GC'd.

4 Type Inference

BROOM’s region type system imposes an annotation burden, and manually annotating C# standard libraries with region types can be tedious. We now present our region type inference algorithm that eliminates the need to write region type annotations. Formally, the type inference algorithm is an elaboration function from programs in $\llbracket FB \rrbracket$ (i.e., FB without region types, but with `letregion` and `open` expressions, similar to the language introduced in § 2) to programs in FB. For simplicity, we assume that each `letregion` and `open` construct in the input program introduces a distinct region identifier π .

Overview. We now present a high-level outline of the type inference algorithm. The algorithm consists of the following steps:

1. *Region Parameterization.* The first step elaborates the input program by introducing *formal region parameters* (for each class and method), and *region variables* (representing yet undetermined *actual region parameters*). We also introduce for each class and method, a *predicate variable* (φ) to denote an undetermined set of outlives-constraints over the region parameters of that class/method.
2. *Constraint Generation.* In the second step, we analyze the program to generate a set of constraints (over the region identifiers and the predicate variables) that must hold (as per the static semantics in Fig. 6).
3. *Constraint Solving.* We solve the generated set of constraints using our fixpoint constraint solving algorithm, which reduces the constraint solving problem to an abduction problem. If the original program in $\llbracket FB \rrbracket$ contains unsafe references, for example, a reference from a transferable region to a stack region, then the constraints generated during the elaboration are not satisfiable. In such a case, the solver fails to solve the constraints.
4. If the solver succeeds, it returns a solution η consisting of a pair of substitution functions η_R and η_P for free region and predicate variables, respectively, introduced in step 1. We apply these substitutions to the elaborated program to produce the final program.

We later on establish the soundness of the type inference algorithm, and the completeness of the constraint-generation and constraint-solving steps (steps 2 and 3 above).

4.1 Region Parameterization for Classes

Region parameterization is an iterative process involving the following three steps, the first two of which are mutually dependent on each other.

Introduction of Formal Region Parameters. For every class C , we identify a sequence of formal region parameters π_0, \dots, π_n that C should be parametric over.

Introduction of Actual Region Parameters. We then replace every instance of class C in the program by an instance $C(\rho_0, \dots, \rho_n)$, where ρ_0, \dots, ρ_n are fresh identifiers denoting actual region parameters.

Predicate Variable Introduction. For every class C , we introduce a fresh predicate variable φ , which represents the yet undetermined outlives-constraints between the formal region parameters of class C .

We identify the region parameters of classes as follows.

Non-Recursive Classes. The class `Object` is defined to have a single region parameter π_0 (the allocation region). The region parameters for any other non-recursive class C is determined only after the region parameters of any class that C depends on have been determined: this includes the base-class B of C and the class (type) of any of its fields. We

11:16 Safe Transferable Regions

replace every dependee type T in C by its instantiated type, using fresh region parameters as needed. The sequence of region parameters for C is defined to be the sequence of region parameters for the base class B concatenated with the list of all fresh region parameters introduced while instantiating the types of the fields in the class. (The class inherits its allocation region from its base class. Note that if a class does not specify an explicit base class, it has an implicit base class `Object`.)

This transformation is illustrated below, using a non-generic `Pair` class:

```

class Pair < Object {
  Object fst;
  Object snd;
}
⇒
class Pair <ρ0, ρ1, ρ2 | φ> < Object <ρ0> {
  Object <ρ1> fst;
  Object <ρ2> snd;
}

```

Recursive Classes. The region parameters for a recursive class is computed in a similar fashion, with the following difference: any recursive field is ignored while instantiating region parameters for the fields of the class, and the region parameters of the recursive class are computed as before. We then do parameter instantiation for all recursive fields, such that their region annotations (the actual region parameters) are exactly the same as the (formal) region parameters of the class. The following example illustrates this for a non-generic `List` class. The resulting class represents a linked list with spine in the region ρ_0 and data objects in the region ρ_1 .

```

class List < Object {
  Object data;
  List next;
}
⇒
class List <ρ0, ρ1 | φ> < Object <ρ0> {
  Object <ρ1> data;
  List <ρ0, ρ1> next;
}

```

The above technique can be extended to mutually recursive classes in a straightforward manner, by simultaneously parameterizing them (and then instantiating them).

Type-Parametric Classes. The type parameter T of a class C is instantiated as $T@ρ$ using a single region parameter $ρ$. (This can be extended to use the bound specified for T , if any.)

Function Types. Since FB is higher-order, fields of function type are allowed. We explain how the parameter instantiation step instantiates function types below, after discussing parameterization for methods.

4.2 Region Parameterization for Methods and Function Types

As the next step, we introduce region parameters for every method. We do this by instantiating the types of all parameters and the return value (of the method) using fresh region identifiers (as explained previously), and then generalizing these region identifiers as formal region parameters of the method. In addition, a fresh region identifier is introduced to represent the allocation region. We also introduce a fresh predicate variable φ for every method, just as we did for each class. Thus, the method

```
Object m (List x) {...}
```

is instantiated as

```
Object<ρ3> m<ρ0, ρ1, ρ2, ρ3> (List<ρ1, ρ2> x) {...}
```

We then consider every method invocation in the program, and introduce fresh region variables representing the (yet unknown) actual region parameters for this particular invocation. We similarly perform instantiation for every constructor invocation of the form `new@π0 T(...)`, by instantiating the type T as before, turning it into `new T<π0, ρ1, ..., ρn>(...)`, where ρ_1, \dots, ρ_n are fresh region variables. Note that the programmer typically specifies the

$$\begin{array}{l}
\pi \in R \text{ (Region constants)} \quad \nu \in V \text{ (Region vars)} \quad \varphi \in P \text{ (Predicate vars)} \quad \Delta \subseteq R \\
\\
\rho \in \text{Region Identifiers} \quad ::= \quad \pi \mid \nu \\
\phi \in \text{Region Constraint} \quad ::= \quad \text{true} \mid \rho \succeq \rho \mid \phi \wedge \phi \\
F \in \text{Substitution} \quad ::= \quad \cdot \mid [\rho/\rho]F \\
\ell \in \text{Antecedent} \quad ::= \quad \phi \mid \varphi \mid \varphi \wedge \phi \\
r \in \text{Consequent} \quad ::= \quad \phi \mid F(\varphi) \\
\text{Constraint} \quad ::= \quad \ell \vdash r \mid \nu \in \Delta \mid \Delta \vdash \varphi \text{ ok}
\end{array}$$

■ **Figure 9** Syntax of constraints.

region π_0 where the object is to be allocated. If the programmer does not specify this, it is allocated in the allocation-context region by default. Region variables are introduced only for the other region parameters.

Function types (of fields and parameters) are instantiated just like methods above, (and the region where the closure is allocated is determined just as for other objects). For example, the function type $\text{List} \rightarrow \text{Object}$ is instantiated as $\langle \rho_0, \rho_1, \rho_2, \rho_3 \mid \varphi \rangle \text{List}(\rho_1, \rho_2) \xrightarrow{\pi} \text{Object}(\rho_3)$. Note that the newly introduced region identifiers are generalized as formal region parameters of the function type. This is a (heuristic) choice made in the case of higher order functions. Consider a higher order function f with a function typed parameter g . The fresh region identifiers introduced while instantiating the type of g could be alternatively generalized as formal region parameters of f , but we choose to generalize them as formal region parameters of g . We will discuss this aspect again later.

4.3 Constraint Generation

The constraint generation algorithm mimics the static type checker, but accumulates constraints that must hold for the type checking to succeed.

Syntax of Constraints. (See Fig. 9.) The constraints are expressed using a set R of region constants, a set V of region variables, and a set P of predicate variables.

Recall that a *region constant* may be either (a) a *formal region parameter* of a class or method, or (b) a *static region identifier* introduced by a `letregion` construct, or (c) an *open transferable region identifier* introduced by an `open` construct. A *region variable* is introduced to represent an unknown *actual* region parameter of a method invocation or object allocation, and the constraint-solver, if successful, will bind each region variable to a region constant.

A *region-constraint* ϕ consists of a conjunction of *outlives-constraints* of the form $\rho_1 \succeq \rho_2$. A predicate variable φ is introduced to represent, *e.g.*, the unknown precondition of a method. The constraint-solver will end up binding it to a *region-constraint* ϕ over a set of fixed formal region parameters. Our constraints also make uses of *pending substitutions* F : A pending substitution serves to bind formal region parameters in φ to the actual region parameters used in a particular context: *E.g.*, in the validity constraint $\pi_1 \succeq \pi_2 \vdash [\pi_1/\rho_1][\pi_2/\rho_2]\varphi$, the pending substitution is $[\pi_1/\rho_1][\pi_2/\rho_2]$.

The constraints are primarily of the form $\varphi_i \wedge \phi_{cx} \vdash \phi_{cs}$ or $\varphi_i \wedge \phi_{cx} \vdash F_j(\varphi_j)$. Here, φ_i is a predicate variable (representing the precondition of a method to be determined), ϕ_{cs} is a

Expression Typing Constraint Generation $\boxed{\mathcal{A}, \Gamma, r \vdash e : \tau \triangleleft C}$

$$\begin{array}{c}
\text{[NEW]} \quad \frac{\mathcal{A}, \Gamma, r \vdash \bar{e} : \bar{\tau} \triangleleft C_1 \quad \mathcal{A} \vdash N \text{ ok} \triangleleft C_2 \quad \text{fields}(N) = \bar{f} : \bar{\tau}}{\mathcal{A}, \Gamma, r \vdash \text{new } N(\bar{e}) : N \triangleleft C_1 \cup C_2} \\
\text{[NEWREGION]} \quad \frac{\mathcal{A}, \Gamma, r \vdash e : \langle \rho \rangle \text{unit} \xrightarrow{\tau} T @ \rho \triangleleft C}{\mathcal{A}, \Gamma, r \vdash \text{new Region} \langle T \rangle \langle \pi_{\top} \rangle (e) : \text{Region} \langle T \rangle \langle \pi_{\top} \rangle \triangleleft C} \\
\text{[LETREGION]} \quad \frac{\mathcal{A} = (\Delta, \Theta, \Phi) \quad \pi \notin \Delta \quad \mathcal{A}' = (\Delta \cup \{\pi\}, \Theta, \Phi \wedge (\Delta \succeq \pi))}{\mathcal{A}', \Gamma, \pi \vdash e : \tau \triangleleft C_1 \quad \mathcal{A} \vdash \tau \text{ ok} \triangleleft C_2} \\
\text{[FNAPPLY]} \quad \frac{\mathcal{A} = (\Delta, \Theta, \Phi) \quad C_1 = \{r\bar{r} \in \Delta\} \quad \mathcal{A}, \Gamma, r \vdash e : \langle \rho\bar{\rho} \mid \phi \rangle \bar{\tau}^1 \xrightarrow{\tau} \tau^2 \triangleleft C_2}{C_3 = \{\Phi \vdash [r\bar{r}/\rho\bar{\rho}]\phi\} \quad \mathcal{A}, \Gamma, r \vdash \bar{e} : [r\bar{r}/\rho\bar{\rho}]\bar{\tau}^1 \triangleleft C_4} \\
\text{[OPEN]} \quad \frac{\mathcal{A} = (\Delta, \Theta, \Phi) \quad \pi \notin \Delta \quad \mathcal{A}, \Gamma, r \vdash e_a : \text{Region} \langle T \rangle \langle \pi_{\top} \rangle \triangleleft C_1}{(\Delta \cup \{\pi\}, \Theta, \Phi), \Gamma[y \mapsto T @ \pi] \vdash e_b : \tau \triangleleft C_2 \quad \mathcal{A} \vdash \tau \text{ ok} \triangleleft C_3} \\
\mathcal{A}, \Gamma, r \vdash \text{open } e_a \text{ as } y @ \pi \text{ in } e_b : \tau \triangleleft (C_1 \cup C_2 \cup C_3)
\end{array}$$

■ **Figure 10** Constraint generation rules part 1.

region-constraint that is *required* to hold at a particular program point (within the method), and ϕ_{cx} is a region-constraint that is *known* to hold at that program point. Constraints of the form $\varphi_i \wedge \phi_{cx} \vdash F_j(\varphi_j)$ are generated by an invocation of a method with precondition φ_j .

We use *well-formedness constraints* of the form $\rho \in \Delta$ to restrict the domain of unification for a region variable (ρ) to a constant set $\Delta = \{\pi_1, \dots, \pi_n\}$ of regions in scope, and well-formedness constraints of form $\Delta \vdash \varphi \text{ ok}$ to restrict the domain of a predicate variable (φ) to the set of all possible region-constraint formulas over a fixed set of regions ($\Delta = \{\pi_1, \dots, \pi_n\}$) in scope.

Constraint Solution. We define an *assignment* η to be a pair of functions (η_R, η_P) , where η_R is a map from V to R and η_P is a map from P to a region-constraint formula. Such an assignment is said to *satisfy* a set of constraints C if every sequent in C is valid after the substitutions η_P and η_R . We say that C is *satisfiable* if it has a satisfying assignment.

Constraint Generation. The constraint generation algorithm is a direct adaption of the type checker: each type checking judgment is modified to produce a set of constraints that must hold for the type checker to succeed.

Fig. 10 illustrates this for selected language constructs. (The appendix contains the remaining constraint generation rules.) These rules use the same context as the corresponding typing judgment in Fig. 6, except that this is generalized to permit the use of region variables and predicate variables. Symbols \mathcal{A} and Γ retain their meaning, modulo this extension. Thus, the component Φ of \mathcal{A} will now be a *symbolic expression* of the form $\varphi \wedge \phi$, where φ is the predicate variable representing the undetermined precondition of the method being analyzed.

The algorithm proceeds top-down, analyzes an expression e in a context \mathcal{A}, Γ, r , and returns a type τ and a set of constraints C (expressed in the rules as $\mathcal{A}, \Gamma, r \vdash e : \tau \triangleleft C$), indicating that the expression will have a type τ provided the constraints C hold.

When we adapt a type-checking judgement rule to produce a corresponding constraint-generation rule, we treat the antecedent conditions of the type-checking rule in one of two ways. Many of these antecedent conditions are converted into constraints which are accumulated in the set C . However, some of the antecedent conditions are expected to be trivially satisfied and are expressed as antecedent conditions in the constraint-generation rule as well. E.g., our frontend ensures that the region constants introduced by `open` and `letregion` constructs are unique (and the elaboration phase ensures this by alpha-renaming them as needed). Thus, the precondition $\rho \notin \Delta$ in rules `LETREGION` and `OPEN` is expected to hold true at constraint-generation time (and does not produce any constraints).

The rules for generating constraints from a method and class definition first build a context (\mathcal{A}) containing a set (Δ) denoting regions that are currently live, a map (Θ) mapping type variables to their bounds, and a constraint formula (Φ) capturing constraints over live region variables. We use predicate variables (φ and φ_m) to capture constraints over variables in Δ that are yet to be inferred.

Let $\text{GENCONSTRAINT}(q)$ denote the set of constraints generated from an elaborated program q . The following theorem states that constraint-generation is sound and complete:

► **Theorem 2.** *Let $C = \text{GENCONSTRAINT}(q)$. An assignment η (for the region and predicate variables in q) satisfies C iff $q[\eta]$ is well-typed.*

4.4 The Constraint Solver

We refer to any validity constraint whose antecedent contains a predicate variable (*i.e.*, is of the form $\varphi \wedge \phi_{cx}$) as an *abduction constraint*. Here, ϕ_{cx} is either *true* (in which case, we call the constraint a trivial abduction constraint) or a conjunction of one or more outlives-constraints (in which case, we call the constraint a non-trivial abduction constraint).

Non-trivial abduction constraints are a key challenge in solving the constraints. We now describe some special properties of the set of constraints C generated by our algorithm, which allow us to handle abduction constraints efficiently.

For any predicate variable φ used in C , C has exactly one constraint of the form $\Delta \vdash \varphi \text{ ok}$. We will refer to this Δ as $\Delta_P^C(\varphi)$. (Our constraint-generation algorithm guarantees the preceding property. Even otherwise, a set of constraints $\Delta_i \vdash \varphi \text{ ok}$ can be replaced by the single equivalent constraint $(\bigcap_i \Delta_i) \vdash \varphi \text{ ok}$.) We say that an abduction constraint is *C-decomposable* if its antecedent is of the form $\varphi \wedge \phi_{cx}$ where φ is a predicate variable and ϕ_{cx} is a conjunction of zero or more outlives-constraints of the form $\pi_1 \succeq \pi_2$ satisfying the following conditions: (1) $\pi_2 \notin \Delta_P^C(\varphi)$. (2) if $\pi_1 \in \Delta_P^C(\varphi)$, then for every $\pi_f \in \Delta_P^C(\varphi)$, $\pi_f \succeq \pi_2$ is a conjunct in ϕ_{cx} . We will omit the reference to C in the above notation if no confusion is likely.

► **Lemma 3.** *Every abduction constraint in C , where $C = \text{GENCONSTRAINT}(q)$, is C-decomposable.*

► **Lemma 4.** *Consider any C-decomposable constraint $\varphi \wedge \phi \vdash \pi_i \succeq \pi_j$ where both π_i and π_j are region constants. Let η satisfy C .*

- (a) *If $\{\pi_i, \pi_j\} \subseteq \Delta_P(\varphi)$: η satisfies $\varphi \wedge \phi \vdash \pi_i \succeq \pi_j$ iff η satisfies $\varphi \vdash \pi_i \succeq \pi_j$.*
- (b) *If $\{\pi_i, \pi_j\} \not\subseteq \Delta_P(\varphi)$: η satisfies $\varphi \wedge \phi \vdash \pi_i \succeq \pi_j$ iff η satisfies $\phi \vdash \pi_i \succeq \pi_j$.*

11:20 Safe Transferable Regions

The above lemma shows how we can reduce a non-trivial abduction constraint to either a trivial abduction constraint or a non-abduction constraint, provided that the consequent is an outlives-constraint without region variables.

Constraint Solver. The first step in our algorithm for solving a set of constraints C computes a set $C^* \supseteq C$ of constraints by iteratively applying the following rules until a fixed point is reached:

1. (Initialization) $\ell \vdash r \in C \Rightarrow \ell \vdash r \in C^*$
2. (Transitivity)
 - a. $\ell \vdash \rho_1 \succeq \rho_2 \in C^*, \ell \wedge \phi \vdash \rho_2 \succeq \rho_3 \in C^* \Rightarrow \ell \wedge \phi \vdash \rho_1 \succeq \rho_3 \in C^*$
 - b. $\ell \wedge \phi \vdash \rho_1 \succeq \rho_2 \in C^*, \ell \vdash \rho_2 \succeq \rho_3 \in C^* \Rightarrow \ell \wedge \phi \vdash \rho_1 \succeq \rho_3 \in C^*$
3. (Substitution) $\ell \vdash F(\varphi) \in C^*, \varphi \vdash \phi \in C^* \Rightarrow \ell \vdash F(\phi) \in C^*$
4. (Abduction Decomposition)
 - a. $\varphi \wedge \phi_{cx} \vdash \pi_i \succeq \pi_j \in C^*, \{\pi_i, \pi_j\} \subseteq \Delta_P(\varphi) \Rightarrow \varphi \vdash \pi_i \succeq \pi_j \in C^*$
 - b. $\varphi \wedge \phi_{cx} \vdash \pi_i \succeq \pi_j \in C^*, \{\pi_i, \pi_j\} \not\subseteq \Delta_P(\varphi), \{\pi_i, \pi_j\} \subseteq R \Rightarrow \phi_{cx} \vdash \pi_i \succeq \pi_j \in C^*$

We can show that every new constraint added by the above rules is implied by existing constraints:

► **Theorem 5.** *Let $C = \text{GENCONSTRAINT}(q)$. An assignment η satisfies C iff η satisfies C^* .*

A key goal of this step is to identify the value every region variable must have in any solution of the set of constraints, as explained below. Consider a set of constraints D . We say that a region variable ρ occurs in a context ℓ (in D) if D contains some constraint $\ell \vdash r$ where ρ occurs in r . We say that a region variable ρ is bound to a region constant π in a context ℓ if $\{\ell \vdash \rho \succeq \pi, \ell \vdash \pi \succeq \rho\} \subseteq D$. We say that a region variable ρ is bound to a region constant π (in D) if ρ is bound to π in every context ℓ in which it occurs. We say that a region variable ρ is somewhere-bound to a region constant π (in D) if ρ is bound to π in some context ℓ in which it occurs.

The set C^* makes it easy to identify a solution $\hat{\eta}$ (if one exists), as below. For any predicate variable φ , $\hat{\eta}_P(\varphi)$ is defined to be $\bigwedge\{\pi_1 \succeq \pi_2 \mid \pi_1, \pi_2 \in R, \varphi \vdash \pi_1 \succeq \pi_2 \in C^*\}$. For any region variable ρ , we define $\hat{\eta}_R(\rho)$ to be any element of the set $\{\pi \in R \mid \rho \text{ is somewhere-bound to } \pi \text{ in } C^*\}$, if this set is non-empty. If this set is empty for any ρ , $\hat{\eta}$ is undefined.

The set C^* also makes it easy to check if C is satisfiable. Let C_g^* denote the subset of all ground constraints (*i.e.*, constraints without any region variable or predicate variable) in C^* . Let WF_R denote the subset of all well-formedness constraints for region variables in C^* . Define $\text{SOLVE}(C)$ as below.

$$\text{SOLVE}(C) = \begin{cases} \text{SOME}(\hat{\eta}) & \text{if } C_g^* \text{ is valid and } \hat{\eta} \text{ is defined and satisfies } \text{WF}_R \\ \text{NONE} & \text{otherwise} \end{cases}$$

► **Theorem 6.** *Let $C = \text{GENCONSTRAINT}(q)$. (Soundness) If $\text{SOLVE}(C) = \text{SOME}(\eta)$, then η satisfies C . (Completeness) If $\text{SOLVE}(C) = \text{NONE}$, then C is unsatisfiable.*

Checking the validity of a ground constraint is straightforward. A ground constraint is of the form $\bigwedge_{i \in I} \ell_i \vdash r$, where each ℓ_i and r is an outlives-constraint. Let D denote $\{\vdash \ell_i \mid i \in I\}$. The given constraint is valid iff r belongs to D^* .

Algorithmic Aspects. Note that checking the validity of a ground constraint can be realized using a simple graph reachability algorithm. Given a set S of outlives constraints, define the directed graph $G(S) = (V(S), E(S))$ as follows. Every distinct region identifier ρ in S is represented by a vertex, which we will also refer to as ρ . Every outlives constraint $\rho_1 \succeq \rho_2$ is represented by an edge from ρ_1 to ρ_2 . It is easy to see that $\wedge S \vdash \rho_1 \succeq \rho_2$ iff there exists a path from ρ_1 to ρ_2 in $G(S)$. Thus, a simple graph reachability algorithm can be used to check the validity of ground constraints.

This idea generalizes. Extending the simple graph reachability algorithm to incorporate the Substitution rule (in computing C^*) turns the problem into a context-free reachability problem in graphs [16] (as usual for context-sensitive interprocedural analysis).

Algorithms for context-free reachability can be adapted to incorporate the Abduction Decomposition step. Alternatively, the iterative process described above is a standard fixed point computation and can be encoded using a set of Datalog rules, allowing us to compute the closure using any Datalog engine.

4.5 Soundness and Completeness: Discussion

Theorems 2 and 6 show that the second and third steps of the type-inference algorithm are sound. It is easy to verify the soundness of the first step (the elaboration phase): For any program $p \in \llbracket FB \rrbracket$, the elaboration phase produces a q such that for any assignment η , we have $\llbracket q[\eta] \rrbracket = p$. The soundness of the type inference algorithm follows.

Theorem 2 and 6 also establish the completeness of the constraint-generation and constraint-solving steps. The only source of incompleteness in the type inference algorithm is the set of heuristic choices made during the first step, as explained below.

(1) We determine the set of region parameters for a recursive class using the heuristic that a recursive occurrence of the class has the same parameters, in the same order, as the class itself. This heuristic fails, for example, if the program uses a recursive list type whose elements alternatively come from two different regions. Such a program would require the following elaboration, which is beyond the scope of our approach:

<pre>class List < Object { Object data; List next; }</pre>	⇒	<pre>class List < ρ₀, ρ₁, ρ₂ φ > Object < ρ₀ > { Object < ρ₁ > data; List < ρ₀, ρ₂, ρ₁ > next; }</pre>
---	---	--

(2) Our technique for region parameterization also uses a heuristic in the case of higher order programs. The following examples illustrates that principal types may not exist for higher order functions.

```
unit apply ( T → unit f, T x, T y ) { f(x); f(y); }
```

This method may be typed assuming either that f is polymorphic over the region that its parameter is allocated in (permitting x and y to be allocated in any regions), or by assuming that x and y are allocated in the same region ρ_1 that f expects its parameters to be allocated in. Neither type subsumes the other. Our algorithm heuristically chooses the first option, as it appears to be the more likely and useful candidate.

If users provide partial region annotations, especially in situations (such as above) where elaboration makes a heuristic choice, the elaboration procedure can use the user-provided choices instead. This can help the type-inference overcome these limitations.

4.6 Modularity Aspects of Type Inference

The type inference algorithm, as presented, traverses the entire program to generate the set of constraints, which are solved en masse, using an iterative fixed point computation. However, the type inference can be realized in a modular and compositional fashion, subject only to the restrictions imposed by recursion.

In the elaboration phase, we can process a class C only after any class B that C depends on has been processed: class C depends on class B if B is either C 's base class or the type of any field of C depends on B . In effect, this means that any collection of mutually recursive classes must be processed together. Non-recursive dependences can be handled in a compositional fashion: if class C depends on B non-recursively, then the elaboration can be done for B first, and then C can be processed.

The same idea applies to the constraint-solving phase as well. Given a set of constraints, we say that a predicate variable φ_1 *directly-depends* on another predicate variable φ_2 if the set of constraints includes a constraint $\varphi_1 \wedge \phi_{cx} \vdash F(\varphi_2)$. We say that φ_1 *depends* on φ_2 if φ_1 transitively depends on φ_2 . The constraint solver needs to process any collection of mutually dependent predicate variables together. In effect, this requires the type inference to process any collection of mutually recursive methods together. However, methods that are not mutually recursive can be processed separately.

5 Implementation and Evaluation

As mentioned earlier, our work is a continuation of the work reported in [7], which provides users with a C#-based implementation of transferable regions (the features described in Section 2). This system has no type system and provides users with no safety guarantees. [7] presents evidence that realistic programs can be implemented using transferable regions and that this can yield significant performance gains. In particular, it reports speedups up to 34% for typical big-data analytics jobs. We now describe our implementation and experience with the type system and type inference algorithm presented in the current paper.

We have implemented BROOMC, a prototype of BROOM compiler frontend, including its region type system and type inference, in 3k+ lines of OCaml. The input to BROOMC is a program in $\llbracket FB^+ \rrbracket$, an extended version of $\llbracket FB \rrbracket$ that includes assignments, conditionals, loops, more primitive datatypes (*e.g.*, integers), and a null value. Our implementation of region type inference and constraint solving closely follows the description given in Sec. 4.

We performed two kinds of experiments to evaluate our region type system and type inference. First, we implemented some microbenchmarks (≤ 100 LOC) consisting of standard classes such as pairs, lists, list iterators, etc., in $\llbracket FB^+ \rrbracket$, and used our inference engine to infer their region types. These classes are region-oblivious. Hence, as long as they are well-typed as per the core type system, BROOMC must be able to automatically construct its region-type-annotated definition without fail. BROOMC was able to infer the expected region types for all these classes under 10ms. Fig 11 shows the region-type-annotated definition computed for the list reverse method. Observe that BROOMC was able to infer that the list and its data (of type T) can be allocated in different regions, as long as the latter outlives the former. This allows, for instance, a `preOrder` method to traverse a tree in a transferable region, and return a list of its nodes, where the list itself is allocated in the stack region.

Next, we translated 4 out of the 6 Naiad streaming query operator benchmarks (Naiad vertices) used in [7] to $\llbracket FB^+ \rrbracket$, and used BROOMC to verify their safety. The 2 remaining benchmarks were left out because their region behavior (from the perspective of the type system) is subsumed by the included benchmarks. The number of LOC performing operations

```

class LinkedList<T><R5,R4 | R4>R5> {
  ListNode<T><R5,R4> head; ...
  List<T><R17,R4> rev<R17,R4 | R4>R17>(unit u) {
    List<T><R17,R4> xs = new List<T><R17,R4>(this.head.val);
    ListNode<T><R5,R4> cur = this.head.next;
    while (!cur == Null) {
      xs.add<R17>(cur.val)
      cur = cur.next; }
    return xs;
  }
}

```

■ **Figure 11** Region-annotated definition of `rev` computed by BROOMC.

on `Region` objects relative to the total LOC is 8% or under in the Naiad benchmarks. During the process, we found multiple instances of potential memory safety violations in the $[[FB^+]]$ translation of all the 4 Naiad vertices, which we verified to be present in the original C# implementation as well. The cause of all safety violations is the creation of a reference from the outgoing message (a transferable region) to the payload of the incoming message. For example, the implementation of `SelectVertex` contains the following:

```

if (this.selector(inMsg.payload[i])) {
  outMsg.set(outputOffset, inMsg.payload[i]);
  ...
}

```

The `outMsg` is later transferred to a downstream actor, where the reference to `inMsg`'s payload becomes unsafe⁶. We eliminated such unsafe references by creating a clone of `inMsg.payload[i]` in `outMsg`, and our compiler was subsequently able to certify the safety of all references.

Our experience with Naiad benchmarks shows the utility of our type inference/checking tool, particularly because it comes at no additional cost to the developer.

6 Related Work

Following Tofte and Talpin's seminal work in [15, 20, 21], static type systems for safe region-based memory management have been extensively studied in the context of various languages and problem settings [8, 10, 24, 2, 1, 9, 23, 17, 11]. Our work differs from the existing proposals in one or more of the following respects.

1. Our design choice focuses on ensuring memory-safety while giving programmers control over region management and allocation of objects in regions. In contrast, some systems automate all aspects of memory management. This is a convenience-performance trade-off.
2. We support both lexically scoped (stack) regions and dynamic transferable regions (both programmer-managed).
3. We exploit a combination of a simple static type discipline and lightweight runtime checks to ensure memory safety. In particular, our approach circumvents the need for restrictive static mechanisms (e.g., linear types and unique pointers) or expensive runtime mechanisms (e.g., garbage collection and reference counting) in order to guarantee safety.

⁶ This unsafe reference could have gone unnoticed during experiments in [7] because their experimental setup included only one actor.

4. We present a full (interprocedural) type inference algorithm that eliminates the need to write region annotations on types.
5. Our underlying language is an object-oriented programming language, equipped with higher-order functions and parameterized (generic) types. These language features necessitate some non-trivial choices in the design of the region-parametricity aspect of the language, which also have an impact on aspects such type inference.

Tofte and Talpin’s approach [21] uses compiler-managed lexically scoped (stack) regions (as a replacement for GC). Our type inference is analogous to theirs in some respects, while differing in others. Their inference algorithm only generates equality constraints, solvable via unification. Our type inference algorithm generates partial order outlives constraints. Consequently, our constraint solving algorithm is more sophisticated, and is capable of inferring unknown outlives constraints over region arguments of polymorphic recursive functions.

Walker and Watkins [23] extend lambda calculus with first-class regions with dynamic lifetimes, and impose linear typing to control accesses to regions. Our open/close lexical block for transferable regions traces its origins to the `let!` expression in [23] and [22], which safely relaxes linear typing restrictions, allowing variables to be temporarily aliased. We don’t use linear typing (for references to regions), thus admit unrestricted aliasing, but use lightweight runtime checks for safety. Moreover, [23]’s linear type system is insufficient to enforce the invariants needed to ensure safety under region transfers, such as the absence of references that escape a transferable region.

Cyclone [8] equips C with programmer-managed stack regions, and a typing discipline that statically guarantees the safety of all pointer dereferences. Later proposals [10, 19] extends Cyclone with dynamic regions. BROOM differs from Cyclone in its non-intrusiveness design principle, which requires its safety mechanisms to not intrude on the programming practices of C#. BROOM programmers, for example, shouldn’t be forced to abandon iterators in favor of for-loops, annotate region types, or rewrite C#’s standard libraries to use in BROOM. Cyclone requires C programmers to use new language constructs and abandon some standard programming idioms in the interest of preserving safety. For instance, Cyclone programmers are required to write region types for functions; the type inference is only intraprocedural. Ensuring safety in presence of dynamic regions requires using either unique pointers or reference-counted objects. Both approaches are intrusive. For example, unique pointers constrain, or in some cases forbid, the use of the familiar iterator pattern, which requires creation of aliases to objects in a collection. Some standard library functions, for example, those that use caching, may need to be rewritten. Moreover, even with unique pointers, safety cannot be guaranteed statically; checks against NULL are needed at run-time to enforce safety. For ref-counted objects, Cyclone requires programmers to use special functions (`alias_refptr` and `drop_refptr`) to create and destroy aliases. Reference count is affected only by these functions. An alias going out of scope, for instance, does not decrement the ref-count. The requirement to use additional constructs to manage aliases makes reference counting more-or-less as intrusive as unique pointers.

Our work differs from Cyclone also in terms of its technical contributions. While Cyclone equips C with a range of region constructs [19], the semantics of (a significant subset of) such constructs, and the safety guarantees of the language are not formalized. In contrast, the (static and dynamic) semantics of Broom has been rigorously defined with respect to a well-understood formal system (FGJ). The safety guarantees have been formalized and proved. Similar contrast can be made of region type inference in both the languages. Cyclone’s type inference was only ever described as being similar to Tofte and Talpin’s, and its effectiveness

in presence of tracked pointers is not clear. In contrast, a detailed type inference algorithm is one of our core contributions.

Our region type system can also be thought of as a specialized ownership type system [5], where each region is the owner of all objects allocated in the region. An ownership type system for safe region-based memory management in real-time Java has been proposed by Boyapati *et al.* [2]. Their language permits only lexically-scoped (stack) regions. In contrast, we permit regions with dynamically determined lifetimes. Our language also admits generics and higher-order functions. We also establish type safety and transfer safety results that formalize the guarantees provided by our system. While Boyapati *et al.*'s language is explicitly typed, our language comes equipped with full type inference. However, several inference algorithms have been proposed in the context of other ownership type systems. Our type inference algorithm is also novel compared to these existing ownership inference algorithms, which are based on, e.g., pointer analysis [12] or boolean satisfiability [6]. (See Section 5.2 of Clark *et al.*'s survey of ownership type systems [5] for a more comprehensive discussion of ownership inference algorithms.) Some distinguishing characteristics of our algorithm is that it is customized to our problem, does not use any pointer analysis algorithm (which can be a source of imprecision) or SAT solvers (which can be a source of inefficiency), and comes with relative completeness guarantees.

Henglein *et al.* [9] propose a flow-sensitive approach for first-order programs to generalize Tofte and Talpin's approach to dynamic regions. Cherem and Rugina [4] describe a flow-insensitive and context-sensitive analysis that transforms Java programs to use (dynamic) regions. However, neither of them supports dynamic regions as first-class objects; they cannot be stored in data structures or passed to methods. Furthermore, while Henglein *et al.* [9] require reference counting to ensure memory safety, Cherem and Rugina's analysis [4] comes with no formal safety guarantees. Holk *et al.* [11] use regions to safely transfer data between the CPU and GPU in the context of Scheme. However, their setting only includes lexically-scoped regions for which Tofte and Talpin-style analysis suffices. In contrast, we provide first-class support for transferable regions with dynamic lifetimes.

References

- 1 Robert L. Bocchino, Jr., Vikram S. Adve, Danny Dig, Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. A type and effect system for deterministic parallel java. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '09*, pages 97–116, New York, NY, USA, 2009. ACM. doi:10.1145/1640089.1640097.
- 2 Chandrasekhar Boyapati, Alexandru Salcianu, William Beebe, Jr., and Martin Rinard. Ownership types for safe region-based memory management in real-time java. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation, PLDI '03*, pages 324–337, New York, NY, USA, 2003. ACM. doi:10.1145/781131.781168.
- 3 Badrish Chandramouli, Jonathan Goldstein, Mike Barnett, Robert DeLine, John C. Platt, James F. Terwilliger, and John Wernsing. Trill: A high-performance incremental query processor for diverse analytics. *PVLDB*, 8(4):401–412, 2014. URL: <http://www.vldb.org/pvldb/vol8/p401-chandramouli.pdf>.
- 4 Sigmund Cherem and Radu Rugina. Region analysis and transformation for java programs. In *Proceedings of the 4th International Symposium on Memory Management, ISMM '04*, pages 85–96, New York, NY, USA, 2004. ACM. doi:10.1145/1029873.1029884.

- 5 Dave Clarke, Johan Östlund, Ilya Sergey, and Tobias Wrigstad. *Ownership Types: A Survey*, pages 15–58. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. doi:10.1007/978-3-642-36946-9_3.
- 6 Werner Dietl, Michael D. Ernst, and Peter Müller. Tunable static inference for generic universe types. In *ECOOP 2011 - Object-Oriented Programming - 25th European Conference, Lancaster, UK, July 25-29, 2011 Proceedings*, pages 333–357, 2011. doi:10.1007/978-3-642-22655-7_16.
- 7 Ionel Gog, Jana Giceva, Malte Schwarzkopf, Kapil Vaswani, Dimitrios Vytiniotis, Ganesan Ramalingam, Manuel Costa, Derek Gordon Murray, Steven Hand, and Michael Isard. Broom: Sweeping out garbage collection from big data systems. In *15th Workshop on Hot Topics in Operating Systems, HotOS XV, Kartause Ittingen, Switzerland, May 18-20, 2015*, 2015. URL: <https://www.usenix.org/conference/hotos15/workshop-program/presentation/gog>.
- 8 Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in cyclone. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation, PLDI '02*, pages 282–293, New York, NY, USA, 2002. ACM. doi:10.1145/512529.512563.
- 9 Fritz Henglein, Henning Makholm, and Henning Niss. A direct approach to control-flow sensitive region-based memory management. In *Proceedings of the 3rd ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming, PPDP '01*, pages 175–186, New York, NY, USA, 2001. ACM. doi:10.1145/773184.773203.
- 10 Michael Hicks, Greg Morrisett, Dan Grossman, and Trevor Jim. Experience with safe manual memory-management in cyclone. In *Proceedings of the 4th International Symposium on Memory Management, ISMM '04*, pages 73–84, New York, NY, USA, 2004. ACM. doi:10.1145/1029873.1029883.
- 11 Eric Holk, Ryan Newton, Jeremy Siek, and Andrew Lumsdaine. Region-based memory management for gpu programming languages: Enabling rich data structures on a spartan host. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA '14*, pages 141–155, New York, NY, USA, 2014. ACM. doi:10.1145/2660193.2660244.
- 12 Wei Huang, Werner Dietl, Ana Milanova, and Michael D. Ernst. Inference and checking of object ownership. In *ECOOP 2012 - Object-Oriented Programming - 26th European Conference, Beijing, China, June 11-16, 2012. Proceedings*, pages 181–206, 2012. doi:10.1007/978-3-642-31057-7_9.
- 13 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.
- 14 Martin Maas, Tim Harris, Krste Asanovic, and John Kubiawicz. Trash day: Coordinating garbage collection in distributed systems. In *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems, HOTOS'15*, pages 1–1, Berkeley, CA, USA, 2015. USENIX Association. URL: <http://dl.acm.org/citation.cfm?id=2831090.2831091>.
- 15 Mads Tofte and Jean-Pierre Talpin. A Theory of Stack Allocation in Polymorphically Typed Languages. Technical Report DIKU-report 93/15, University of Copenhagen, 1993. URL: <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.51.6564>.
- 16 Thomas W. Reps. Program analysis via graph reachability. *Information & Software Technology*, 40(11-12):701–726, 1998. doi:10.1016/S0950-5849(98)00093-7.
- 17 The Rust Programming Language, 2015. Accessed: 2015-11-7 13:21:00. URL: <https://doc.rust-lang.org/book>.
- 18 Type-safe off-heap memory, 2016. Accessed: 2016-06-6 13:21:00. URL: <https://github.com/densh/scala-offheap>.

- 19 Nikhil Swamy, Michael Hicks, Greg Morrisett, Dan Grossman, and Trevor Jim. Safe manual memory management in cyclone. *Science of Computer Programming*, 62(2):122–144, 2006. Special Issue: Five perspectives on modern memory management - Systems, hardware and theory. doi:10.1016/j.scico.2006.02.003.
- 20 Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value λ -calculus using a stack of regions. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '94, pages 188–201, New York, NY, USA, 1994. ACM. doi:10.1145/174675.177855.
- 21 Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Inf. Comput.*, 132(2):109–176, 1997. doi:10.1006/inco.1996.2613.
- 22 Philip Wadler. Linear Types Can Change the World! In M. Broy and C. B. Jones, editors, *IFIP TC 2 Working Conference on Programming Concepts and Methods*, pages 561–581, Sea of Gallilee, Israel, 1990. North-Holland.
- 23 David Walker and Kevin Watkins. On regions and linear types (extended abstract). In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming*, ICFP '01, pages 181–192, New York, NY, USA, 2001. ACM. doi:10.1145/507635.507658.
- 24 Bennett Norton Yates. A type-and-effect system for encapsulating memory in java. Master's thesis, Department of Computer Science and Information Science, University of Oregon, 1999.
- 25 Matei Zaharia. New developments in spark, 2015. URL: <http://www.slideshare.net/databricks/new-developments-in-spark>.

A Appendix

A.1 Static Semantics

$$\begin{array}{llll}
 \text{allocRgn}(A\langle r\bar{r}\rangle\langle\bar{T}\rangle) & = & r & \text{bound}_{\Theta}(a@r) & = & \Theta(a)@r \\
 \text{allocRgn}(\langle\bar{\rho}|\phi\rangle\bar{\tau}^1 \xrightarrow{r} \tau^2) & = & r & \text{bound}_{\Theta}(N) & = & N \\
 \text{shape}(A\langle\bar{r}\rangle\langle\bar{T}\rangle) & = & A\langle\bar{T}\rangle & \text{fields}(\text{Object}\langle r\rangle) & = & \bullet
 \end{array}$$

$$\frac{CT(B) = \text{class } B\langle\bar{a}\triangleleft\bar{K}\rangle\langle\bar{\rho}|\phi\rangle\triangleleft N\{\bar{\tau}^f \bar{f}; \dots\} \quad \mathcal{S} = [\bar{r}/\bar{\rho}, \bar{T}/\bar{a}] \quad \text{fields}(\mathcal{S}(N)) = \bar{g} : \bar{\tau}^g}{\text{fields}(B\langle\bar{T}\rangle\langle\bar{r}\rangle) = \bar{g} : \bar{\tau}^g, \bar{f} : \mathcal{S}(\bar{\tau}^f)}$$

$$\frac{CT(B) = \text{class } B\langle\bar{a}\triangleleft\bar{K}\rangle\langle\bar{\rho}|\phi\rangle\triangleleft N\{\bar{\tau}^f \bar{f}; \bar{d}\} \quad m \notin \bar{d} \quad \mathcal{S} = [\bar{r}/\bar{\rho}, \bar{T}/\bar{a}]}{\text{mtype}(m, B\langle\bar{T}\rangle\langle\bar{r}\rangle) = \text{mtype}(m, \mathcal{S}(N))}$$

$$\frac{CT(B) = \text{class } B\langle\bar{a}\triangleleft\bar{K}\rangle\langle\bar{\rho}|\phi\rangle\triangleleft N\{\bar{\tau}^f \bar{f}; \bar{d}\} \quad \tau^2 m\langle\bar{\rho}_m|\phi_m\rangle(\bar{\tau}^1 \bar{x})\{\dots\} \in \bar{d} \quad \mathcal{S} = [\bar{r}/\bar{\rho}, \bar{T}/\bar{a}]}{\text{mtype}(m, B\langle\bar{T}\rangle\langle\bar{r}\rangle) = \mathcal{S}(\langle\bar{\rho}_m|\phi_m\rangle\bar{\tau}^1 \rightarrow \tau^2)}$$

$$\frac{\text{mtype}(m, N) = \langle\bar{\rho}_1|\phi_1\rangle\bar{\tau}^{11} \rightarrow \tau^{12} \quad \text{implies} \quad \mathcal{A}, \Phi \vdash \phi_2 \Leftrightarrow [\bar{\rho}_2/\bar{\rho}_1](\phi_1) \quad \text{and} \quad \bar{\tau}^{21} = [\bar{\rho}_2/\bar{\rho}_1](\bar{\tau}^{11}) \quad \text{and} \quad \mathcal{A} \vdash \tau^{22} <: [\bar{\rho}_2/\bar{\rho}_1](\tau^{12})}{\mathcal{A} \vdash \text{override}(m, N, \langle\bar{\rho}_2|\phi_1\rangle\bar{\tau}^{21} \rightarrow \tau^{22})}$$

■ **Figure 12** FEATHERWEIGHT BROOM: auxiliary definitions.

Subtyping $\boxed{\mathcal{A} \vdash \tau_1 <: \tau_2}$

$$\frac{\mathcal{A} \vdash \tau <: \tau \quad (\Delta, \Theta, \Phi) \vdash a @ \rho <: \Theta(a) @ \rho}{CT(B) = \text{class } B \langle \bar{a} \triangleleft \bar{K} \rangle \langle \bar{\rho} | \phi \rangle \triangleleft N \{ \dots \}} \quad \mathcal{A} \vdash B \langle \bar{T} \rangle \langle \bar{r} \rangle <: [\bar{r} / \bar{\rho}, \bar{T} / \bar{a}] (N)$$

$$\frac{\mathcal{A} \vdash \tau_1 <: \tau_2 \quad \mathcal{A} \vdash \tau_2 <: \tau_3}{\mathcal{A} \vdash \tau_1 <: \tau_3} \quad \frac{\mathcal{A} \cdot \Phi \vdash \phi_1 \Rightarrow \phi_2 \quad \mathcal{A} \vdash \bar{\tau}^{11} <: \bar{\tau}^{21} \quad \mathcal{A} \vdash \tau^{22} <: \tau^{12}}{\mathcal{A} \vdash \langle \bar{\rho} | \phi_2 \rangle \bar{\tau}^{21} \xrightarrow{r} \tau^{22} <: \langle \bar{\rho} | \phi_1 \rangle \bar{\tau}^{11} \xrightarrow{r} \tau^{12}}$$

Type, and Type Constraint Well-formedness $\boxed{\mathcal{A} \vdash \tau \text{ ok}, \Delta \vdash \phi \text{ ok}}$

$$\frac{r \in \Delta}{(\Delta, \Theta, \Phi) \vdash \text{Object} \langle r \rangle \text{ ok}} \quad \frac{r_0, r_1 \in \Delta}{\Delta \vdash r_0 \succeq r_1 \text{ ok}} \quad \frac{\Delta \vdash \phi_0 \text{ ok} \quad \Delta \vdash \phi_1 \text{ ok}}{\Delta \vdash \phi_0 \wedge \phi_1 \text{ ok}}$$

$$\frac{r \in \Delta \quad \bar{\rho} \notin \Delta \quad \Delta' = \Delta \cup \{ \bar{\rho} \} \quad \mathcal{A}' = (\Delta', \Theta, \Phi \wedge \phi) \quad \Delta' \vdash \phi \text{ ok} \quad \mathcal{A}' \vdash \bar{\tau}^1 \text{ ok} \quad \mathcal{A}' \vdash \tau^2 \text{ ok}}{(\Delta, \Theta, \Phi) \vdash \langle \bar{\rho} | \phi \rangle \bar{\tau}^1 \xrightarrow{r} \tau^2 \text{ ok}}$$

Class Well-formedness $\boxed{B \text{ ok}}$

$$\frac{\Delta = \{ \rho, \bar{\rho} \} \quad \Theta = [\bar{a} \mapsto \bar{K}] \quad \Phi = \phi \quad \mathcal{A} = (\Delta, \Theta, \Phi) \quad \Delta \vdash \phi \text{ ok} \quad \Theta \Vdash \bar{K} \text{ ok} \quad \bar{d} \text{ ok in } B \quad \mathcal{A} \vdash N, \bar{\tau}^f \text{ ok} \quad \text{shape}(N) \neq \text{Region} \langle T \rangle \quad \Phi \vdash \text{allocRgn}(\bar{\tau}^f) \succeq \rho \quad \text{allocRgn}(N) = \rho}{\text{class } B \langle \bar{a} \triangleleft \bar{K} \rangle \langle \bar{\rho} | \phi \rangle \triangleleft N \{ \bar{\tau}^f \bar{f}; \bar{d} \} \text{ ok}}$$

■ **Figure 13** FEATHERWEIGHT BROOM: subtyping and well-formedness rules (continuation of Fig. 6).

Expression Typing $\boxed{\mathcal{A}, \Gamma, r \vdash e : \tau}$

$$\frac{\mathcal{A}, \Gamma, r \vdash e_1 : \tau_1 \quad \mathcal{A}, \Gamma[x \mapsto \tau_1], r \vdash e_2 : \tau_2}{\mathcal{A}, \Gamma, r \vdash \text{let } x = e_1 \text{ in } e_2 : \tau_2} \quad \frac{\mathcal{A} = (\Delta, \Theta, \Phi) \quad 1 \notin \Delta \quad \mathcal{A}' = (\Delta \cup \{1\}, \Theta, \Phi \wedge \Delta \succeq 1) \quad \mathcal{A}', \Gamma, 1 \vdash e : \tau \quad \mathcal{A} \vdash \tau \text{ ok}}{\mathcal{A}, \Gamma, r \vdash \text{letd } 1 \text{ in } e : \tau}$$

$$\frac{\mathcal{A} = (\Delta, \Theta, \Phi) \quad \mathcal{A}' = (\Delta \cup \{1\}, \Theta, \Phi) \quad \mathcal{A}' \vdash T @ 1 \text{ ok} \quad \mathcal{A}', \Gamma, 1 \vdash e : T @ 1}{\mathcal{A}, \Gamma, r \vdash \text{new Region} \langle T \rangle \langle 1 \tau 1 \rangle (e) : \text{Region} \langle T \rangle \langle \pi \tau \rangle}$$

$$\frac{\mathcal{A} = (\Delta, \Theta, \Phi) \quad \mathcal{A}' = (\Delta \cup \{1\}, \Theta, \Phi) \quad 1 \in \Delta \text{ implies } s = \blacksquare \quad \mathcal{A}', \Gamma, 1 \vdash e : \tau \quad \mathcal{A} \vdash \tau \text{ ok}}{\mathcal{A}, \Gamma, r \vdash \text{opened } 1(s) \text{ in } e : \tau} \quad \frac{\mathcal{A}, \Gamma, r \vdash e : \tau \quad \mathcal{A} \vdash \tau <: \tau'}{\mathcal{A}, \Gamma, r \vdash e : \tau'}$$

■ **Figure 14** FEATHERWEIGHT BROOM: expression typing (continuation of Fig. 6).

A.2 Operational Semantics

$$\boxed{(e, \Sigma) \longrightarrow (e', \Sigma')}$$

$$\begin{array}{c}
\text{[EVALORDER]} \quad \frac{(e, \Sigma) \longrightarrow (e', \Sigma')}{(E[e], \Sigma) \longrightarrow (E[e'], \Sigma')} \\
\text{[EXCEPTION]} \quad \frac{(e, \Sigma) \longrightarrow \perp}{(E[e], \Sigma) \longrightarrow \perp} \\
\text{[FIELDACCESS]} \quad \frac{\Sigma(\mathbf{1}) = \blacksquare \quad \text{fields}(A\langle\bar{T}\rangle\langle\mathbf{1}\bar{\mathbf{1}}\rangle) = \bar{f} : \bar{\tau}}{((\text{new } A\langle\bar{T}\rangle\langle\mathbf{1}\bar{\mathbf{1}}\rangle)(\bar{v})).f_i, \Sigma) \longrightarrow (v_i, \Sigma)} \\
\text{[METHODINV]} \quad \frac{\Sigma(\mathbf{1}) = \blacksquare \quad \text{mbody}(m, A\langle\bar{T}\rangle\langle\mathbf{1}\bar{\mathbf{1}}\rangle) = \rho\bar{p}.\bar{x}.e}{((\text{new } A\langle\bar{T}\rangle\langle\mathbf{1}\bar{\mathbf{1}}\rangle)(\bar{v})).m\langle\mathbf{1}'\bar{\mathbf{1}}'\rangle(\bar{v}'), \Sigma) \longrightarrow \\ (\mathbf{1}'\bar{\mathbf{1}}'/\rho\bar{p}][\bar{v}'/\bar{x}][\text{new } A\langle\bar{T}\rangle\langle\mathbf{1}\bar{\mathbf{1}}\rangle(\bar{v})/\text{this}]e, \Sigma)} \\
\text{[FNAPPLY]} \quad \frac{v_a = \lambda\mathbf{1}'\langle\rho\bar{p}\rangle(\bar{\tau} \bar{x}).e \quad \Sigma(\mathbf{1}') = \blacksquare}{(v_a\langle\mathbf{1}\bar{\mathbf{1}}\rangle(\bar{v}), \Sigma) \longrightarrow ([\bar{v}/\bar{x}][\mathbf{1}\bar{\mathbf{1}}/\rho\bar{p}]e, \Sigma)} \\
\text{[EXCEPTION]} \quad \frac{\Sigma(\mathbf{1}) = \square \quad (e, \Sigma[\mathbf{1} \mapsto \blacksquare]) \longrightarrow \perp}{(\text{new Region}\langle T \rangle\langle\mathbf{1}\bar{\mathbf{1}}\rangle)(e), \Sigma) \longrightarrow \perp} \\
\text{[LETEXP]} \quad \frac{}{(\text{let } x = v \text{ in } e, \Sigma) \longrightarrow ([v/x]e, \Sigma)}
\end{array}$$

Evaluation Context \boxed{E}

$$\begin{array}{l}
E ::= \bullet \mid (\bullet).f \mid \bullet.m\langle\bar{\mathbf{1}}\rangle(\bar{e}) \mid v.m\langle\bar{\mathbf{1}}\rangle(\dots, \bullet, \dots) \mid \text{new } N(\dots, \bullet, \dots) \\
\quad \mid \text{new Region}\langle T \rangle\langle\pi_{\top}\rangle(\bullet) \mid \bullet\langle\bar{\mathbf{1}}\rangle(\bar{e}) \mid v\langle\bar{\mathbf{1}}\rangle(\dots, \bullet, \dots) \\
\quad \mid \text{let } x = \bullet \text{ in } e \mid \text{open } \bullet \text{ as } y@r \text{ in } e
\end{array}$$

■ **Figure 15** FEATHERWEIGHT BROOM: operational semantics (continuation of Fig. 8).

A.3 Constraint Generation Rules

Expression Typing Constraint Generation $\boxed{\mathcal{A}, \Gamma, r \vdash e : \tau \triangleleft C}$	
[UNIT]	$\mathcal{A}, \Gamma, r \vdash () : \mathbf{unit} \triangleleft \{\}$
[VAR]	$\mathcal{A}, \Gamma, r \vdash x : \Gamma(\tau) \triangleleft \{\}$
[FIELDACCESS]	$\frac{\mathcal{A}, \Gamma, r \vdash e : \tau' \triangleleft C \quad \bar{f} : \bar{\tau} = \mathbf{fields}(\mathbf{bound}_{\mathcal{A}, \Theta}(\tau'))}{\mathcal{A}, \Gamma, r \vdash e.f_i : \tau_i \triangleleft C}$
[LET]	$\frac{\mathcal{A}, \Gamma, r \vdash e_1 : \tau_1 \triangleleft C_1 \quad \mathcal{A}, \Gamma[x \mapsto \tau_1], r \vdash e_2 : \tau_2 \triangleleft C_2}{\mathcal{A}, \Gamma, r \vdash \mathbf{let } x = e_1 \mathbf{ in } e_2 : \tau_2 \triangleleft C_1 \cup C_2}$
[METHODINV]	$\frac{\begin{array}{l} \mathcal{A}, \Gamma, r \vdash e_0 : \tau \triangleleft C_1 \quad C_2 = \{r\bar{r} \in \mathcal{A}.\Delta\} \\ \mathbf{mtype}(m, \mathbf{bound}_{\mathcal{A}, \Theta}(\tau)) = \langle \rho\bar{\rho} \mid \phi \rangle \bar{\tau}^1 \rightarrow \tau^2 \\ \mathcal{A} \vdash \langle \rho\bar{\rho} \mid \phi \rangle \bar{\tau}^1 \rightarrow \tau^2 \mathbf{ok} \triangleleft C_3 \quad \mathcal{A}, \Gamma, r \vdash \bar{e} : [r\bar{r}/\rho\bar{\rho}](\bar{\tau}^1) \triangleleft C_4 \\ C_5 = \{\mathcal{A}.\Phi \vdash [r\bar{r}/\rho\bar{\rho}](\phi)\} \end{array}}{\mathcal{A}, \Gamma, r \vdash e_0.m(r\bar{r})(\bar{e}) : [r\bar{r}/\rho\bar{\rho}](\tau^2) \triangleleft C_1 \cup C_2 \cup C_3 \cup C_4 \cup C_5}$
[LAMBDA]	$\frac{\begin{array}{l} r \in \mathcal{A}.\Delta \quad \rho\bar{\rho} \notin \mathcal{A}.\Delta \quad \mathcal{A}' = (\mathcal{A}.\Delta \cup \{\rho\bar{\rho}\}, \mathcal{A}.\Theta, \mathcal{A}.\Phi \wedge \phi) \\ \mathcal{A}'.\Delta \vdash \phi \mathbf{ok} \quad \mathcal{A}' \vdash \bar{\tau}^1 \mathbf{ok} \triangleleft C_1 \quad \mathcal{A}' \vdash \tau^2 \mathbf{ok} \triangleleft C_2 \\ \mathcal{A}', \Gamma[\bar{x} \mapsto \bar{\tau}^1], \rho \vdash e : \tau^2 \triangleleft C_3 \end{array}}{\mathcal{A}, \Gamma, r \vdash \lambda @ r \langle \rho\bar{\rho} \mid \phi \rangle (\bar{x} : \bar{\tau}^1).e : \langle \rho\bar{\rho} \mid \phi \rangle \bar{\tau}^1 \xrightarrow{\tau} \tau^2 \triangleleft \bigcup_{i=1}^4 C_i}$
[SUBTYPING]	$\frac{\mathcal{A}, \Gamma, r \vdash e : \tau \triangleleft C_1 \quad \mathcal{A} \vdash \tau <: \tau' \triangleleft C_2}{\mathcal{A}, \Gamma, r \vdash e : \tau' \triangleleft C_1 \cup C_2}$
Method Well-formedness Constraint Generation $\boxed{\vdash d \mathbf{ok in } B \triangleleft C}$	
[METHOD]	$\frac{\begin{array}{l} CT(B) = \mathbf{class } B \langle \bar{a} \triangleleft \bar{K} \rangle \langle \bar{\rho} \mid \varphi \rangle \triangleleft N \{ \dots \} \\ \mathcal{A} = (\Delta, \Theta, \Phi) = (\{\bar{\rho}, \rho_m, \bar{\rho}_m\}, [\bar{a} \mapsto \bar{K}], \varphi_m) \quad C_1 = \{\Delta \vdash \varphi_m \mathbf{ok}\} \\ \Gamma = \cdot [\mathbf{this} \mapsto B \langle \bar{a} \rangle \langle \bar{\rho} \rangle][\bar{x} \mapsto \bar{\tau}^1] \quad \mathbf{mtype}(m, N) = \langle \bar{\rho}_m \mid \phi_m \rangle \bar{\tau}^1 \rightarrow \tau^2 \\ \mathcal{A}, \Gamma, \rho_m \vdash e : \tau^2 \triangleleft C_2 \quad \mathcal{A} \vdash \bar{\tau}^1 \mathbf{ok} \triangleleft C_3 \quad \mathcal{A} \vdash \tau^2 \mathbf{ok} \triangleleft C_4 \end{array}}{\vdash \tau^2 m \langle \rho_m \bar{\rho}_m \mid \varphi_m \rangle (\bar{\tau}^1 \bar{x}) \{ \mathbf{return } e; \} \mathbf{ok in } B \triangleleft (C_1 \cup C_2 \cup C_3 \cup C_4)}$
Class Well-formedness Constraint Generation $\boxed{\vdash B \mathbf{ok} \triangleleft C}$	
[CLASS]	$\frac{\begin{array}{l} \mathcal{A} = (\Delta, \Theta, \Phi) = (\{\rho, \bar{\rho}\}, [\bar{a} \mapsto \bar{K}], \varphi) \\ C_1 = \{\Delta \vdash \varphi \mathbf{ok}\} \quad \Theta \Vdash \bar{K} \mathbf{ok} \quad \mathcal{A} \vdash N \mathbf{ok} \triangleleft C_2 \quad \mathcal{A} \vdash \bar{\tau}^f \mathbf{ok} \triangleleft C_3 \\ C_4 = \{\Phi \vdash \mathbf{allocRgn}(\bar{\tau}^f) \succeq \rho \wedge \mathbf{allocRgn}(N) = \rho\} \\ \vdash \bar{d} \mathbf{ok in } B \triangleleft C_5 \end{array}}{\vdash \mathbf{class } B \langle \bar{a} \triangleleft \bar{K} \rangle \langle \rho\bar{\rho} \mid \varphi \rangle \triangleleft N \{ \bar{\tau}^f \bar{x}; \bar{d} \} \mathbf{ok} \triangleleft \bigcup_{i=1}^5 C_i}$

■ **Figure 16** FEATHERWEIGHT BROOM: Constraint generation rules part 2 (Continuation of Fig. 10).

	Subtyping Constraint Generation $\boxed{\mathcal{A} \vdash \tau_1 <: \tau_2 \triangleleft C}$
[REFLEXIVITY]	$\mathcal{A} \vdash \tau <: \tau \triangleleft \{\}$
[UNIFY]	$\mathcal{A} \vdash \tau <: [\pi/\rho](\tau) \triangleleft \{\pi \succeq \rho, \rho \succeq \pi\}$
[TRANSITIVITY]	$\frac{\mathcal{A} \vdash \tau_1 <: \tau_2 \triangleleft C_1 \quad \mathcal{A} \vdash \tau_2 <: \tau_3 \triangleleft C_2}{\mathcal{A} \vdash \tau_1 <: \tau_3 \triangleleft C_1 \cup C_2}$
[FNSUBTYPING]	$\frac{C_1 = \{\mathcal{A}.\Phi \vdash \phi_1 \Rightarrow \phi_2\} \quad \mathcal{A} \vdash \overline{\tau^{11}} <: \overline{\tau^{21}} \triangleleft C_2 \quad \mathcal{A} \vdash \tau^{22} <: \tau^{12} \triangleleft C_3}{\mathcal{A} \vdash \langle \overline{\rho} \phi_2 \rangle \overline{\tau^{21}} \xrightarrow{r} \tau^{22} <: \langle \overline{\rho} \phi_1 \rangle \overline{\tau^{11}} \xrightarrow{r} \tau^{12} \triangleleft C_1 \cup C_2 \cup C_3}$
	Type Well-formedness Constraint Generation $\boxed{\mathcal{A} \vdash \tau \text{ ok} \triangleleft C}$
[OBJECTTYPE]	$\frac{C = \{r \in \Delta\}}{(\Delta, \Theta, \Phi) \vdash \text{Object} \langle r \rangle \text{ ok} \triangleleft C}$
[CLASSTYPE]	$\frac{CT(B) = \text{class } B \langle \overline{a} \triangleleft \overline{K} \rangle \langle \overline{\rho} \phi \rangle \triangleleft N \{ \dots \} \quad \Theta \Vdash B \langle \overline{T} \rangle \text{ ok} \quad C = \{ \overline{r} \in \Delta, \Phi \vdash [\overline{r}/\overline{\rho}](\phi) \}}{(\Delta, \Theta, \Phi) \vdash B \langle \overline{T} \rangle \langle \overline{r} \rangle \text{ ok} \triangleleft C}$
[TYPEPARAM]	$\frac{\Theta \Vdash T \text{ ok} \quad \Theta \Vdash T <: \text{Object} \quad C = \{r \in \Delta\}}{(\Delta, \Theta, \Phi) \vdash T @ r \text{ ok} \triangleleft C}$
[FNSTYPE]	$\frac{C_1 = \{r \in \Delta\} \quad \overline{\rho} \notin \Delta \quad \Delta' = \Delta \cup \{ \overline{\rho} \} \quad \mathcal{A}' = (\Delta', \Theta, \Phi \wedge \phi) \quad \Delta' \vdash \phi \text{ ok} \triangleleft C_2 \quad \mathcal{A}' \vdash \overline{\tau^1} \text{ ok} \triangleleft C_3 \quad \mathcal{A}' \vdash \tau^2 \text{ ok} \triangleleft C_4}{(\Delta, \Theta, \Phi) \vdash \langle \overline{\rho} \phi \rangle \overline{\tau^1} \xrightarrow{r} \tau^2 \text{ ok} \triangleleft C_1 \cup C_2 \cup C_3 \cup C_4}$
[REGIONTYPE]	$\frac{\Theta \Vdash T \text{ ok}}{(\Delta, \Theta, \Phi) \vdash \text{Region} \langle T \rangle \langle \pi_T \rangle \text{ ok} \triangleleft \{\}}$

■ **Figure 17** FEATHERWEIGHT BROOM: Constraint generation rules part 3 (Continuation of Fig. 16).

KafKa: Gradual Typing for Objects

Benjamin Chung

Northeastern University, Boston, MA, USA

Paley Li

Czech Technical University, Prague, Czech Republic, and
Northeastern University, Boston, MA, USA

Francesco Zappa Nardelli

Inria, Paris, France, and
Northeastern University, Boston, MA, USA

Jan Vitek

Czech Technical University, Prague, Czech Republic, and
Northeastern University, Boston, MA, USA

Abstract

A wide range of gradual type systems have been proposed, providing many languages with the ability to mix typed and untyped code. However, hiding under language details, these gradual type systems embody fundamentally different ideas of what it means to be well-typed.

In this paper, we show that four of the most common gradual type systems provide distinct guarantees, and we give a formal framework for comparing gradual type systems for object-oriented languages. First, we show that the different gradual type systems are practically distinguishable via a three-part litmus test. We present a formal framework for defining and comparing gradual type systems. Within this framework, different gradual type systems become translations between a common source and target language, allowing for direct comparison of semantics and guarantees.

2012 ACM Subject Classification Software and its engineering → Semantics

Keywords and phrases Gradual typing, object-orientation, language design, type systems

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2018.12

Supplement Material ECOOP Artifact Evaluation approved artifact available at
<http://dx.doi.org/10.4230/DARTS.4.3.10>

Funding This work received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement 695412), the NSF (award 1544542 and award 1518844) as well as ONR (award 503353).

Acknowledgements The author thank the reviewers of ECOOP, POPL, ESOP, and again ECOOP for comments that gradually improved this paper.

We are grateful to Leif Andersen, Fabian Muelbrock, Éric Tanter, Celeste Hollenbeck, Sam Caldwell, Ming-Ho Yee, Lionel Zoubritzky, Benjamin Greenman and Matthias Felleisen for their feedback.



© Benjamin Chung, Paley Li, Francesco Zappa Nardelli, and Jan Vitek;
licensed under Creative Commons License CC-BY

32nd European Conference on Object-Oriented Programming (ECOOP 2018).

Editor: Todd Millstein; Article No. 12; pp. 12:1–12:25

Leibniz International Proceedings in Informatics



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



1 Introduction

“Because half the problem is seeing the problem”

There never was a single approach to gradual typing. The field was opened by two simultaneously published papers. One, by Siek and Taha, typed individual Scheme terms using a consistency relation, casts being inserted by a type directed translation [19]. The other, by Tobin-Hochstadt and Felleisen, described a system allowing programmers to add types to individual modules, using constraint solving to determine where contracts are needed to protect typed and untyped code from each other [26]. These two approaches set the tone for a decade of research. Today, gradual type systems rely on a variety of languages, enforcement mechanisms with various guarantees; this linguistic diversity is not without consequence, however, as the very notion of what constitutes an error remains unsettled.

The type system and semantics of a programming language are necessarily tightly coupled; each has to deal with the language’s complexity. As a result, the same gradual type system may seem very different when applied to two different languages, an issue that shows up clearly with object-oriented languages. Siek and Taha’s first effort [20] presented a gradual type system for an object-oriented programming language. It related objects by generalizing the notion of consistency [19] over structural subtyping. The work had drawbacks, most notably in the handling of mutable state and aliasing – vital features of object-oriented languages. Underlying each subsequent gradual type system are different design choices on how to deal with mutability and aliasing.

The landscape of gradually typed object-oriented languages is rich and includes:

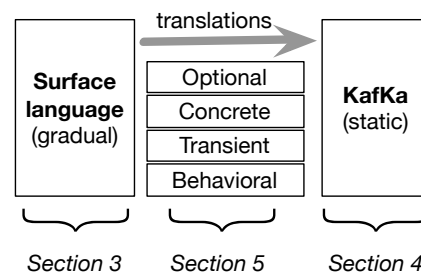
- Typed Racket: a rich gradual type system based on contracts.
- Gradualtalk: a gradual variant of Smalltalk.
- C#: a statically typed language with a dynamic type.
- Dart: a class-based language with optional types.
- Hack: a statically typed variant of PHP that allows untyped code.
- Thorn: a language with both statically typed and untyped code.
- TypeScript: JavaScript with optional types.
- StrongScript: a variant of TypeScript with nominal types.
- Nom: a language supporting dynamic types and nominal typing.
- Reticulated Python: a family of gradual type systems for Python.

These languages differ in their type systems and associated run-time enforcement strategies. There are four major approaches, labeled here as optional, concrete, behavioral, and transient. The *optional* approach, chosen by TypeScript, Dart, and Hack, amounts to static type checking followed by type erasure. Erroneous values flowing from dynamically typed code to statically typed code will not be caught. The *concrete* approach, used in C# and Nom, uses run-time subtype tests on type constructors to supplement static typing. While statically typed code executes at native speed, values are dynamically checked at typed-untyped boundaries. The *behavioral* approach of Typed Racket and Gradualtalk monitors values to ensure that they behave in accordance to their assigned types. Instead of checking higher-order and mutable values for static type tags like concrete, wrappers ensure enduring conformance of values to their declared type. The *transient* approach, specific to Reticulated Python, lies between concrete and behavioral; it adds type casts but does so only for the top level of data structures. Finally, Thorn and StrongScript combine the optional and concrete approaches, differentiating between erased types and run-time-checked types.

Static type systems for object-oriented languages are designed to prevent dynamic “method not understood” errors. For gradual type systems, however, some method not found errors

cannot be ruled out before execution. In such a gradual type system, untyped code can pass an ill-typed value to typed code, breaking soundness. The meaning of an “error” for a gradual type system, therefore, depends on how type specifications are enforced. In other words, each gradual type system may catch different “errors.” We demonstrate this with a litmus test consisting of three simple programs capable of distinguishing the four above-mentioned approaches. The litmus test programs are statically well-typed and “correct” in the sense that they run to completion without error in an untyped language. However, when executed under different gradual typing systems, they produce different errors. For intuition, consider a call, $x.m()$, where $x : C$ and C has a method m returning a D . In the concrete approach, this call will succeed. With behavioral, the call will go through, but an error may be reported if m returns a value of the wrong type. In transient, the call is similarly guaranteed to go through, but might return the wrong type without reporting an error. Finally, in optional, the call may get stuck, as x may not have a method named m ; and, if it succeeds, there is no guarantee that type D will be returned.

We propose to compare approaches to gradual typing for objects by translating a gradually typed surface language to a target language called *KafKa*. Our surface language is a *gradually* typed class-based object-oriented language similar to Featherweight Java. *KafKa* is a *statically* typed class-based object calculus with mutable state. The key difference between the two is the sound type system and casts of *KafKa*. Where the surface language allows implicit coercions, *KafKa* requires explicit casts to convert types. Casts come in two kinds: *structural casts* check for subtyping, while *behavioral casts* monitor that an object behaves as if it was of some type. Translating from surface to target language involves adding casts, the location and type of which depends on the gradual type system.



This paper makes the following contributions:

- The design of a core calculus for gradual type systems for objects.
- Translations of each gradual approach to the core calculus.
- A litmus test comprised of three programs to tell apart the gradual type systems.
- Supplementary material includes a mechanized proof of soundness of the type system of the core calculus and its proof-of-concept implementation on .Net.

Our work does not address the question of performance of the translations. Each of the semantics for gradual typing has intrinsic performance costs; but these can be mitigated by compiler and run-time optimizations, which we do not perform. *KafKa* departs from prior work (e.g. [12]) as *KafKa* is statically typed. By translating to a statically typed core, we can clearly see where wrapper-induced dynamic errors can occur. Another design choice is the use of structural subtyping in *KafKa*. This is motivated by our desire to represent behavioral and transient approaches that require structural subtyping. We do not foresee difficulties either switching to a nominal type system or providing an additional nominal subtype cast.

All of our code and proofs are available from:

github.com/BenChung/GradualComparisonArtifact.

2 Background

“If you know the enemy and know yourself...”

The intellectual lineage of gradual typing can be traced back to attempts to add types to Smalltalk and LISP. On the Smalltalk side, work on the Strongtalk optional type system [7] led to Bracha’s notion of pluggable types [6]. In Bracha’s notion of pluggable types, types exist solely to catch errors at compile-time, never affecting the run-time behavior of programs. An optional type system is *trace preserving*: that is to say, if a term e reduces to a , then adding type annotations to e does not prevent it from reducing to a [18]. This property is valuable to developers as it ensures that type annotations will not introduce errors; and thus, adding types does not increase the testing burden! Optional type systems in wide use include Hack [25], TypeScript [3] and Dart [24].

Felleisen and his students have contributed substantially to gradual typing. The Typed Scheme [27] design, that later became Typed Racket, is influenced by their earlier work on higher-order contracts and semantic casts [10, 11]. Typed Racket was envisioned as a vehicle for teaching programming, being able to explain the source of errors and avoiding surprises for beginning users were important considerations. For this reason, a value that flowed in a variable of type t , was required behave as if it belonged to that type throughout its lifetime. Whenever a higher-order or mutable value crosses a boundary between typed and untyped code, it is wrapped in a contract that monitors the value’s behavior. If the value misbehaves, blame can be assigned to the boundary that assigned it the type that was violated. The granularity of typing is the module, thus a module is either entirely typed or entirely untyped. Typed Racket’s support for objects was described by Takikawa et al. [23].

Siek and Taha coined the term gradual typing in [19] as “any type system that allows programmers to control the degree of static checking for a program by choosing to annotate function parameters with types, or not.” They formalized this idea in the lambda calculus augmented with references. To make the type system a gradual one, they defined the type consistency relation $t \sim t'$. If $t \sim t'$, then t is consistent with t' , and can therefore be used implicitly where a t' instance is expected. This enables gradual typing, as $\star \sim t$ for every t and vice versa, allowing untyped values to be passed where typed ones are expected. Siek and Taha extended this idea to an object calculus [20]. In order to do so, they combined consistency with structural subtyping, producing consistent subtyping. With consistent subtyping, consistency can be used when checking structural subtyping, allowing typed objects and untyped objects to be mixed. To explore the design space, Reticulated Python [28] was given three modes: the *guarded* mode behaves as Typed Racket with contracts applied to values. The *transient* mode performs shallow subtype checks on reads and method returns, only validating if the value obtained has matching method types. The *monotonic* mode is fundamentally different from any of the previous approaches. Monotonic cast updates the type of values in place by replacing some of the occurrences of \star with more specific types, and these updates propagate recursively through the heap until fix-point.

Other noteworthy systems include Gradualtalk [1], C# 4.0 [4], Thorn [5], Nom [15] and StrongScript [18]. Gradualtalk is a variant of Smalltalk with behavioral casts and mostly nominal type equivalence (structural equivalence can be specified on demand, but it is rarely used). It has an optional mode and a mode in which blame can be turned off.

C# 4.0 adds the type `dynamic` to C# and adds dynamically resolved method invocation. Thus C# has a dynamic sublanguage that allows developers to write unchecked code, working alongside a sound typed sublanguage in which values are always of their declared type. The implementation replaces \star by the type `object` and adds casts where needed.

	Nominal	Optional	Concrete	Behavioral	Class based	First-class Class	Soundness claim	Unboxed prim.	Subtype cast	Shallow subtype cast	Behavioral cast	Blame	Pathologies
Dart	•	•			•				•				-
Hack	•	•			•				•				-
TypeScript		•			•								-
C#	•		•		•		• ²	•	•		•		-
Thorn	•	•	•		•		• ²	•	•				-
StrongScript	•	•	•	•	•		• ²		•		•		1.1x
Nom	•		•		•		• ²	•	•			•	1.1x
Gradualtalk	• ¹			•	•		•				•	•	5x
Typed Racket				•	•	•	•			•	•	•	121x
Reticulated Python													
<i>Transient</i>		•			•		•			•			10x
<i>Monotonic</i>				•	•		•				•		27x
<i>Guarded</i>				•	•		•				•	•	21x

■ **Figure 1** Gradual type systems. (1) Opt. structural constraints. (2) Typed expressions are sound.

Thorn and StrongScript extend the C# approach with the addition of optional types (called *like types* in Thorn). Thorn is implemented by translation to the JVM. StrongScript is implemented on top of a modified version of the V8 VM. The presence of concrete types means that the compiler can optimize code (unbox data and in-line methods) and programmers are ensured that type errors will not occur within concretely typed code. Nom is similar to Thorn in that it is nominal and follows the concrete approach.

Fig. 1 reviews gradual type systems for objects. All languages are class-based, except TypeScript which has both classes and JavaScript objects. While that choice is not crucial; classes are useful as a source of type declarations. Most languages build subtyping on explicit subtype declarations, nominal typing, rather than on structural similarities. TypeScript uses structural subtyping but does not implement a run-time check for it. Anecdotal evidence suggests that TypeScript could switch to nominal subtyping with little effort, as was done for StrongScript [18]. While nominal subtyping leads to more efficient type casts, Reticulated Python’s subtype consistency relation is fundamentally structural; it would be nonsensical to use it in a nominal system.

For Racket, the heavy use of first-class classes and class generation naturally leads to structural subtyping as many of the classes being manipulated have no names and arise during computation.

The optional approach is the default for Dart, Hack, and TypeScript. Transient Reticulated Python allows any value to flow in a field, regardless of type annotations, leading to its “open world” soundness guarantee [28]. Some languages like Dart and Gradualtalk can operate in a checked and an uncheck mode. In Thorn, Nom, and C#, primitives are concretely typed; they can be unboxed without tagging. The choice of casts follows from other design decisions. The concrete approach naturally tends to use subtype tests to establish the type of values. For nominal systems, there are highly optimized algorithms.

Shallow casts are casts that only check the presence of methods but not their signature. They are used by Racket and Reticulated Python to ensure some basic form of type confor-

mance. Behavioral casts are used when information, such as a type or a blame label, must be associated with a reference or an object.

Some of the systems provide soundness guarantees. In Typed Racket, Gradualtalk and Guarded Reticulated Python, there is a guarantee that a type error that is exercised will be caught by a contract. In concrete approaches, any typed expression is guaranteed to be correct, errors occur at boundary crossings. The guarantees provided by Transient and Monotonic are somewhat harder to characterize and out of the scope of this review.

Blame assignment is a topic of investigation in its own right. Anecdotal evidence suggests that the context provided by blame helps developers pinpoint the provenance of errors. In the same way that a Java stack trace identifies the function that went wrong, blame identifies where a type assertion came from. This is especially important in behavioural gradual type systems, as type assertions become wrappers which can propagate through the heap. Blame identifies where a failing wrapper came from, a task that would otherwise require extensive backtracking debugging. Unlike stack traces, which have little run-time cost, blame tracking has a cost due to its meta-data. Blame information has to be stored whenever a wrapper is applied and is believed to cause substantial slowdowns. However, this has not been measured in detail. With the concrete approach, blame is trivially available as evaluation stops at the boundary that causes the failure [15]. We are primarily concerned with where the error arises, rather than what information is reported; thus, we do not consider blame further.

The last column of Fig. 1 lists self-reported performance pathologies. These numbers are not comparable, as they refer to different programs and different configurations of type annotations. They are not worst case scenarios either; most languages lack a sufficient corpus of code to conduct a thorough evaluation. Nevertheless, one can observe that for optional types no overhead is expected, as the type annotations are erased during compilation. Concrete types insert efficient casts and lead to code that can be optimized. The performance of the transient semantics for Reticulated Python is a worst case scenario for concrete types – i.e., there is a cast at almost every call. Finally, languages with behavioral casts are prone to significant slowdowns. Compiler optimizations for reducing these overheads are an active research topic [2, 17]. Languages such as C#, Nom, Thorn, and StrongScript are designed so that the performance of fully typed code is better than untyped code, so that mixed code performs well thanks to the relatively inexpensive nominal subtype tests.

In contemporary work, Greenman and Felleisen describe three approaches to *migratory typing* in the context of a lambda calculus extended with pairs and primitive values [12]. Their *natural embedding* corresponds to the behavioral approach, the *erasure embedding* is the optional approach, and the *locally-defensive embedding* is transient. They do not consider the concrete approach – neither objects or mutable state. They give performance results for a non-optimizing implementation of the embeddings, and the results are as expected: behavioral has extreme worst cases and transient significantly slows down fully-typed programs. While they do not evaluate object-oriented programs, these are unlikely to fare better. Our work differs in that we are trying to express a translation between object calculi using features that are readily available in most virtual machines.

3 A Family of Gradually Typed Languages and their Litmus Test

“There is no perfection only life”

There is no single, common notion of what constitutes an erroneous gradually typed program – a consequence of the varied enforcement strategies. The choice of enforcement strategy is reflected in the semantics of the language which, in turn, implies that developers have to understand the details of that strategy to avoid run-time errors. This also means that it is possible to differentiate between approaches by simply observing the run-time errors that each type system produces. We propose a litmus test consisting of three programs whose execution depends on which gradual type system is in use. Each of these programs is statically well-typed and runs without error when executed with a purely dynamic semantics. However, this varies as we use different semantics for gradual typing. We start by presenting a common surface language in which we can express our programs, and then explain why the various approaches to gradual typing yield different run-time errors.

3.1 A Common Surface Language

To normalize our presentation, we use a single surface language for all four of the gradual type systems under study. The surface language is a *gradually* typed object calculus without inheritance, method overloading or explicit type cast operations. Fig. 2 gives its syntax and an extract of its static semantics. The distinctive feature of the calculus is the presence of type \star – the dynamic type. A variable of type \star can hold any value, an invocation of a method with receiver of type \star is always statically well-typed, and an expression of type \star can appear anywhere within a typed program.

This dynamic type lets us convert our otherwise statically typed language to a gradually typed one. If a well-typed program does not use \star , then it will not get stuck on method invocation. A program where all variables are annotated as \star is fully dynamic, and any given invocation may get stuck. Gradual typing comes into play when an expression of type \star occurs as an argument to a method that expects some other type C and conversely when an argument of type C is passed to a method that expects \star . The static type system of the surface language allows such implicit coercions – using the *convertibility* relation – but run-time checks may be inserted to catch potential type mismatches. We formalize the semantics of this system later; here, we appeal to the readers’ intuition.

Before presenting the litmus tests, some details about the type system of the surface language may prove helpful. The subtyping relation is structural with the Amber rule [8] to enable recursion. $M \ K \vdash C <: D$ holds if class C has (at least) all the methods of class D and the arguments and return types are related by subtyping in the usual contra- and co-variant way; the class table K holds definitions of all classes, and M is a helper for recursion that records the subtype relations encountered so far. One noteworthy feature of subtyping is that the fields of objects do not play a role in deciding if classes are subtypes. Following languages like Smalltalk, fields are encapsulated and can only be accessed from within their defining object. Syntactically, field reads and writes are limited to the self-reference `this`.

The static type checking rules, $\Gamma \ K \vdash_s e : t$ where Γ is a type environment and K is a class table, are standard with two exceptions: method invocation and convertibility. Method invocation is always allowed when the receiver e is of type \star ; therefore, $e.m(e')$ has type \star if the argument can have type \star . Convertibility is used when statically typed and dynamically typed terms interact. The convertibility relation, written $K \vdash_s t \Rightarrow t'$, states that type t is convertible to type t' in class table K . It is used both for up-casting and for conversions of \star to non- \star types. $K \vdash_s t \Rightarrow t'$ holds when $t <: t'$, this allows up-casts. The remaining

Syntax:

$$\begin{aligned}
 k &::= \text{class } C \{ \text{fd}_{1..} \text{ md}_{1..} \} & \text{md} &::= m(x:t):t \{e\} & \text{fd} &::= f:t & t &::= \star \mid C \\
 K &::= k \ K \mid \cdot & \Gamma &::= x:t \ \Gamma \mid \cdot & M &::= C <: D \ M \mid \cdot \\
 e &::= x \mid \text{this} \mid \text{this.f} \mid \text{this.f} = e \mid e.m(e) \mid \text{new } C(e_{1..})
 \end{aligned}$$

Typing expressions:

$$\begin{array}{c}
 \frac{\Gamma(x) = t}{\Gamma K \vdash_s x : t} \qquad \frac{\Gamma(\text{this}) = C \quad f : t \in K(C)}{\Gamma K \vdash_s \text{this.f} : t} \qquad \frac{\Gamma(\text{this}) = C \quad f : t \in K(C) \quad \Gamma K \vdash_s e : t' \quad K \vdash_s t' \Rightarrow t}{\Gamma K \vdash_s \text{this.f} = e : t} \qquad \frac{\Gamma K \vdash_s e : \star \quad \Gamma K \vdash_s e' : t}{\Gamma K \vdash_s e.m(e') : \star} \\
 \\
 \frac{\Gamma K \vdash_s e : C \quad \Gamma K \vdash_s e' : t \quad m(t_1) : t_2 \in K(C) \quad K \vdash_s t \Rightarrow t_1}{\Gamma K \vdash_s e.m(e') : t_2} \qquad \frac{f_1 : t_{1..} \in K(C) \quad \Gamma K \vdash_s e_1 : t'_1.. \quad K \vdash_s t'_1 \Rightarrow t_{1..}}{\Gamma K \vdash_s \text{new } C(e_{1..}) : C}
 \end{array}$$

Convertibility:

$$\frac{\cdot K \vdash_s t <: t'}{K \vdash_s t \Rightarrow t'} \qquad \frac{}{K \vdash_s t \Rightarrow \star} \qquad \frac{}{K \vdash_s \star \Rightarrow t}$$

Subtyping:

$$\frac{}{M K \vdash \star <: \star} \qquad \frac{C <: D \in M}{M K \vdash C <: D} \qquad \frac{M' = C <: D \ M \quad \text{md} \in K(D) \implies \text{md}' \in K(C) \ . \ M' K \vdash \text{md} <: \text{md}'}{M K \vdash C <: D} \\
 \\
 \frac{M K \vdash t'_1 <: t_1 \quad M K \vdash t_2 <: t'_2}{M K \vdash m(x:t_1):t_2 \{e\} <: m(x:t'_1):t'_2 \{e'\}}$$

■ **Figure 2** Surface language syntax and type system (extract).

two rules allow implicit conversion to and from the dynamic type. To avoid collapsing the type hierarchy, convertibility is not transitive. It is through convertibility that our surface language becomes gradual.

3.2 Litmus

Using three different programs, we can differentiate between four gradual type systems. The litmus test, shown in Fig. 3, and its constituent programs are written in our surface language. Each of these programs consists of a class table and an expression whose evaluation in the context of the class table determines if the litmus test succeeds or fails.

The programs in the litmus test are designed to induce errors. This is done by arranging for values to cross typed/untyped boundaries in a way that will cause some type systems to report an error but not others. At heart, these programs can be summarized by the type boundaries that are crossed by an object. The notation $C \mid_t$ denotes an object of class C passing through a boundary that expects it to be of type t . For example, if method m expects an argument of type t , a method call $e.m(e')$ would induce the boundary $e' \mid_t$. In program **L1**, we have:

$$A \mid_{\star} \mid_I$$

	L1	L2	L3
Concrete			
Behavioral		✓	
Transient		✓	✓
Optional	✓	✓	✓

<pre> class A { m(x:A):A {this}} class I { n(x:I):I {this}} class T { s(x:I):T {this} t(x:*):* {this.s(x)}} </pre> <div style="text-align: right; background-color: #cccccc; padding: 2px;">L1</div> <pre> new T().t(new A()) </pre>	<pre> class A { m(x:A):A {this}} class Q { n(x:Q):Q {this}} class I { m(x:Q):I {this}} class T { s(x:I):T {this} t(x:*):* {this.s(x)}} </pre> <div style="text-align: right; background-color: #cccccc; padding: 2px;">L2</div> <pre> new T().t(new A()) </pre>	<pre> class C { a(x:C):C {x}} class D { b(x:D):D {x}} class E { a(x:D):D {x}} class F { m(x:E):E {x} n(x:*):* {this.m(x)}} </pre> <div style="text-align: right; background-color: #cccccc; padding: 2px;">L3</div> <pre> new F().n(new C()) .a(new C()) </pre>
---	---	---

■ **Figure 3** Litmus test. Each program consists of a class table (top) and an expression (bottom). Top left table indicates successful executions.

An instance of class *A* is first implicitly converted to \star and then to *I*; in this program classes *A* and *I* are unrelated by subtyping. In **L2**, the same sequence of conversions is applied:

$$A \mid_{\star} \mid_I$$

This time *A* and *I* both have a method *m*, but the methods have incompatible argument types. Lastly, in **L3** we start by converting a *C* to \star and then to *E* and finally back to \star :

$$C \mid_{\star} \mid_E \mid_{\star}$$

The resulting value is then used to call method *m* with an argument of class *C*. This correct as method *m* in *C* does expect an argument of that type. If the object was an instance of *E* instead, the call would not be legal because *E*'s method *m* expects a class *D* as argument.

Optional. An optional gradual type system simply erases all of the type annotations at run-time; all three programs run to completion without error.

Concrete. The concrete approach ensures that a variable of some class *C* always refers to an object of that class or of a subtype of it. To ensure this is the case, all implicit conversions imply a run-time subtype check. This causes all three programs to fail. **L1** and **L2** fail on a subtype test $K \vdash A <: I$. **L3** fails on the subtype test $K \vdash C <: E$.

Behavioral. The behavioral approach allows conversion from \star to *C* if the value is compatible to *C* and if, after that, it behaves as if it was an instance of *C*. The former is checked by a shallow cast that only looks at method names, and the latter by a wrapper that monitors further interactions. **L1** fails at because *A* does not have the method *n* expected by *I*. **L2**, however, executes without error because *A* has the method *m* expected by *I*. **L3** fails, since the instance of *C* has been applied a wrapper for *E*. When method *a* is called with a *C*, the wrapper notices that *E*'s method *a* expects a *D* and that *C* and *D* are not compatible.

Transient. The transient approach is weaker than behavioral. It retains the shallow structural checks at casts of the behavioral approach, but does not wrap values. Transient fails **L1**, for the same reason as the other two type systems, and passes **L2**, for the same reason as behavioral. **L3** succeeds because transient forgets that the C object was cast to E.

3.3 Discussion

These programs capture the behavior of implementations. The three have been expressed in TypeScript (optional), StrongScript (concrete), Typed Racket (behavioral) and Reticulated Python (transient), with the same errors.¹ What the litmus test shows is that a precise understanding of the semantics of gradually typed languages, and their run-time enforcement machinery, is crucial for developers to know if a program is “correct.” Here, all errors are false positives, since none of these programs performs an invalid operation. This underlines the fact that in gradually typed language, type specifications can lead to run-time errors just as faulty code can. Thus, type annotations must be audited and tested just like code.

These approaches induce usability trade-offs. One way to contextualize this is with the gradual guarantee of Siek et al [21]. Informally, it states that if there exists a static type assignment to an untyped program that allows the program to run to completion, any partial assignment of those types will do too. The optional approach trivially fulfills this guarantee. Transient likewise satisfies the gradual guarantee [29], as it only checks top-level structure of values at type boundaries. Unlike optional, transient does ensure that typed calls succeed; however, a call may produce a dynamic error if the receiver is of the wrong type (even in a typed context), or the return is an ill-typed value. The behavioral approach also fulfills the guarantee, as it only checks arguments and return types when wrappers are invoked. However, typed function calls can fail if they call an untyped function that returns the wrong value. Finally, the concrete approach ensures that every typed call is successful. However, this comes at the expense of the gradual guarantee – partially typed classes are not compatible with more-or-less typed ones. The guarantee is incompatible with subtyping. Suppose a program were to rely on the judgment $\{m(C) : D\} <: \{m(C) : D\}$. Relaxing the argument type to a dynamic type, $\{m(\star) : D\} <: \{m(C) : D\}$, violates subtyping. To overcome this, Reticulated Python augments subtyping with the aforementioned consistency relation. This increases the number of programs accepted by the static type system. However, it is not used by the run-time semantics and is fundamentally incompatible with the concrete approach (because any use of consistent subtyping will fail). We omit consistent subtyping.

As an alternative consider the approaches taken by Thorn or StrongScript. They have three kinds of types: \star (dynamic), C (concrete), and like C (optional), combining the concrete and optional approach into the same language. This design allows for a different kind of migration; once a program is fully annotated with optional types, they all can be converted to concrete types without introducing any run-time errors [18]. We do not model this combination directly, as the underlying details are no different from the concrete approach.

The motivation for making fields private is to simplify the system. With private fields, errors are limited to method invocation. Field accesses can be trivially checked; as they are always accessing `this`. Moreover, interposing on method invocation can easily be achieved by wrappers, whereas interposing on field access would require modifying the code of clients. This would make the formal development more cumbersome without adding insight.

¹ github.com/BenChung/GradualComparisonArtifact/examples

4 KafKa: A Core Calculus

“Aux chenilles du monde entier et aux papillons qu’elles renferment”

Even without gradual typing, comparing languages is difficult. Small differences in syntax and features can make even the most similar languages appear different. As a result, the nuances of gradual type systems are often hidden amongst irrelevant details. To enable direct comparison, we propose to translate gradually typed languages down to a common calculus designed to highlight the distinctions between designs. The target for this translation is our core language, **KafKa**. **KafKa** is a statically typed language similar to the surface language but with several features added to enable its use as a common target language.

These additions make an explicit distinction between static and dynamic operations and replace implicit conversions with explicit casts. **KafKa**’s first additions consist of two casts which are used at boundaries between typed and untyped code. The structural subtype cast, written $\langle t \rangle e$, ensures that expression e evaluates to a subtype of t . The behavioral cast, written $\blacktriangleleft t \blacktriangleright e$, creates a wrapper around the value of e that monitors e to ensure that it behaves as if it was of type t . Additionally, a syntactic distinction is made between static method invocation, written $e.m_{t \rightarrow t'}(e')$, dynamic method invocation, $e@m_{\star \rightarrow \star}(e')$. Static invocations does not get stuck, whereas dynamic invocations can. This makes explicit which function calls can fail.

KafKa was designed to align with common statically-typed class-based object-oriented compilation targets like .NET or the JVM. It maps to the intermediate language supported by these platforms. **KafKa** also requires the ability to generate new classes at run-time, a feature supported by these environments but typically not present in class-based calculi.

4.1 Syntax and Semantics

We had two requirements when designing **KafKa**: first, to be expressive enough to capture the dynamic semantics implied by each gradual type system; second, to have a type system that can express when code is known to be error free. **KafKa**’s syntax and semantics, loosely inspired by Featherweight Java [13], are shown in Fig. 4. At the top level, classes are notated as **class** $C \{ fd_1.. md_1.. \}$; methods, ranged over by md , are denoted as $m(x: t) : t \{ e \}$; and fields $f: t$. Expressions consist of:

- variables, x ;
- the self-reference **this**; and wrapper target, **that**;
- field accesses, $this.f$, and field writes, $this.f = e$;
- object creation, **new** $C(e_1..)$;
- static and dynamic method invocations;
- subtype and behavioral casts;
- and heap addresses.

The static semantics holds only a few surprises; key typing rules appear in Fig. 4. The subtyping relation is inherited from the surface language. The program typing relation (not shown here), $e \mathcal{K} \checkmark$, indicates that expression e is well-formed with respect to class table \mathcal{K} . The expression typing judgment $\Gamma \S \mathcal{K} \vdash e : t$, indicates that against Γ with heap typing \S and class table \mathcal{K} , e has type t . Unlike the surface language, **KafKa** does not rely on a convertibility relation from \star to \mathcal{C} and back; instead, explicit casts are required.

Evaluation is mostly standard with an evaluation context consisting of a class table \mathcal{K} , the expression being evaluated e , and a heap σ mapping from addresses a to objects, denoted $\mathcal{C}\{a \dots\}$. Due to the need for dynamic code generation, the class table is part of the state.

Syntax:

$e ::= x$	this	that	$k ::= \text{class } C \{ \text{fd}_{1..} \text{ md}_{1..} \}$
this.f	$\text{this.f} = e$	$\text{new } C(e_{1..})$	$\text{md} ::= m(x:t):t \{e\}$
$e.m_{t \rightarrow t'}(e)$	$e@m_{* \rightarrow *}(e)$		$\text{fd} ::= f:t$
$\langle t \rangle e$	$\blacktriangleleft t \blacktriangleright e$		$t ::= * \mid C$
a	$a.f$	$a.f = e$	$K ::= k \ K \mid \cdot$
			$\Sigma ::= a:t \ \Sigma \mid \cdot$

Static semantics:

$\frac{\Gamma \Sigma K \vdash e : C \quad m(t):t' \in K(C) \quad \Gamma \Sigma K \vdash e' : t}{\Gamma \Sigma K \vdash e.m_{t \rightarrow t'}(e') : t'}$	$\frac{\Gamma \Sigma K \vdash e : * \quad \Gamma \Sigma K \vdash e' : *}{\Gamma \Sigma K \vdash e@m_{* \rightarrow *}(e') : *}$		
$\frac{\Gamma \Sigma K \vdash e : t'}{\Gamma \Sigma K \vdash \langle t \rangle e : t}$	$\frac{\Gamma \Sigma K \vdash e : t'}{\Gamma \Sigma K \vdash \blacktriangleleft t \blacktriangleright e : t}$	$\frac{\Sigma(a) = C}{\Gamma \Sigma K \vdash a : C}$	$\frac{}{\Gamma \Sigma K \vdash a : *}$

Execution contexts:

$E ::= a.f = E$	$E.m_{t \rightarrow t'}(e)$	$a.m_{t \rightarrow t'}(E)$	$E@m_{* \rightarrow *}(e)$	\square
$a@m_{* \rightarrow *}(E)$	$\langle t \rangle E$	$\blacktriangleleft t \blacktriangleright E$	$\text{new } C(a_{1..} E e_{1..})$	

Dynamic semantics:

$K \text{ new } C(a_{1..})$	$\sigma \rightarrow K \ a' \ \sigma'$	where	$a' \text{ fresh} \quad \sigma' = \sigma[a' \mapsto C\{a_{1..}\}]$
$K \ a.f_i$	$\sigma \rightarrow K \ a_i \ \sigma$	where	$\sigma(a) = C\{a_1, \dots, a_i, a_n \dots\}$
$K \ a.f_i = a'$	$\sigma \rightarrow K \ a' \ \sigma'$	where	$\sigma(a) = C\{a_1, \dots, a_i, a_n \dots\}$ $\sigma' = \sigma[a \mapsto C\{a_1, \dots, a', a_n \dots\}]$
$K \ a.m_{t \rightarrow t'}(a')$	$\sigma \rightarrow K \ e' \ \sigma$	where	$e' = [a/\text{this } a'/x]e$ $m(x:t_1):t_2 \{e\} \in K(C)$ $\sigma(a) = C\{a_{1..}\} \quad \emptyset K \vdash t <: t_1$ $\emptyset K \vdash t_2 <: t'$
$K \ a@m_{* \rightarrow *}(a')$	$\sigma \rightarrow K \ e' \ \sigma$	where	$e' = [a/\text{this } a'/x]e$ $m(x:*) : * \{e\} \in K(C)$ $\sigma(a) = C\{a_{1..}\}$
$K \ \langle * \rangle a$	$\sigma \rightarrow K \ a \ \sigma$		
$K \ \langle D \rangle a$	$\sigma \rightarrow K \ a \ \sigma$	where	$\emptyset K \vdash C <: D \quad \sigma(a) = C\{a_{1..}\}$
$K \ \blacktriangleleft t \blacktriangleright a$	$\sigma \rightarrow K' \ a' \ \sigma'$	where	$K' \ a' \ \sigma' = \text{bcast}(a, t, \sigma, K)$
$K \ E[e]$	$\sigma \rightarrow K' \ E[e'] \ \sigma'$	where	$K \ e \ \sigma \rightarrow K' \ e' \ \sigma'$

■ Figure 4 Kafka dynamic semantics and static semantics (extract).

4.1.1 Method Invocation

KafKa has two invocation forms, the dynamic $e@m_{* \rightarrow *}(e')$ and the static $e.m_{t \rightarrow t'}(e')$, both denoting a call to method m with argument e' . There are several design issues worth discussing. First, as our calculus is a translation target, it is acceptable to require some explicit preparation for objects to be used in a dynamic context. A dynamic call is only successful if the receiver has a method of the expected name and with argument and return types of $*$. This is a design choice of KafKa, even dynamic invocation has to be well-typed

(even if this typing is trivial). Secondly, it is possible for a static invocation to call an untyped method (of type \star to \star). Consider the following class definition

```
class C {
  m(x: Int): Int { x + 2 }
  m(x:  $\star$ ):  $\star$  {  $\langle \star \rangle$  this.mInt→Int( $\langle$ Int $\rangle$ x) }
}
```

assume that class table K holds a definition for `Int`, and that we have integer constants and addition. The class demonstrates several features of `KafKa`. Its class well-formedness rules (not shown here) allow a limited form of method overloading. A class may have at most two occurrences of any method m : one “untyped” with \star as argument and return type; and one, which we call “typed”, where either its argument or return type differ from \star . The static type system enforces a single means of invoking a typed method m :

```
new C().mInt→Int(2)
```

Here, the receiver is obviously of type `C` and the argument is `Int`; thus, the call is statically well-typed. The expression will therefore evaluate the body of m . For an untyped method, there are two invocation modes:

```
new C()@m $\star$ → $\star$ (2)
```

The call executes correctly here, as `C` has an untyped m . However, in the general case, there is no guarantee that the receiver of an untyped invocation has the requested method; and therefore, a dynamic invocation can get stuck. The receiver of a dynamic invocation has type \star . When the receiver type is known to be some `C` and that class has the requested method, then the static invocation can be used:

```
new C().m $\star$ → $\star$ (2)
```

The invocation will succeed. All nuances of invocation will come in handy when translating the surface language to `KafKa`.

4.1.2 Run-time Casts

`KafKa` has two cast operations: the structural subtype cast $\langle t \rangle e$ and the behavioral cast $\blacktriangleleft t \blacktriangleright e$, both indicating the desire for the result of evaluating e to be of type t . Where the casts differ is what is meant by “has type t ”. The subtype cast checks that the result of evaluating e is an object whose class is compatible with t . If t is a class, then it will check for a subtyping relation; otherwise, if $t = \star$, the cast will succeed. As every value in the heap is tagged by its type constructor, it is always possible to perform this check. The behavioral cast is more complex; we will describe it in the remainder of this section.

The objective of the behavioral cast is to ensure that the wrapped object behaves as the target type dictates. When given the address a of some object, this cast creates a wrapper object, say a' , that enforces the invariant that a *behaves* as a value of type t . Function $\text{bcast}(a, t, \sigma, K) = K' a' \sigma'$ specifies its semantics, shown in Fig. 5. There are two cases to consider: either the target type is a class C' , or it is \star .

If the target type is C' , then $\text{bcast}(a, C', \sigma, K)$ will return an updated class table K' , a reference to the wrapped object a' , and an updated heap σ' . As long as a has every method name specified by C' , the cast itself will succeed. If a is missing a method, it is

Behavioral cast: $\text{bcast}(a, t, \sigma, K) = K' a' \sigma'$

a	Reference to wrap		a'	Wrapped reference
t	Target type to enforce		K'	Class table with wrapper
σ	Original heap		σ'	New heap
K	Original class table			

$$\text{bcast}(a, C', \sigma, K) = K' a' \sigma' \quad \text{where} \quad \left\{ \begin{array}{l} \sigma(a) = C\{a_{1..}\} \quad D, a' \text{ fresh} \quad \sigma' = \sigma[a' \mapsto D\{a\}] \\ \text{md}_{1..} \in K(C) \quad \text{names}(\text{md}'_{1..}) \subseteq \text{names}(\text{md}_{1..}) \\ \text{md}'_{1..} \in K(C') \quad \text{nodups}(\text{md}_{1..}) \quad \text{nodups}(\text{md}'_{1..}) \\ K' = K W(C, \text{md}_{1..}, \text{md}'_{1..}, D) \end{array} \right.$$

$$\text{bcast}(a, \star, \sigma, K) = K' a' \sigma' \quad \text{where} \quad \left\{ \begin{array}{l} \sigma(a) = C\{a_{1..}\} \quad \text{md}_{1..} \in K(C) \quad D, a' \text{ fresh} \\ \text{nodups}(\text{md}_{1..}) \quad K' = K W\star(C, \text{md}_{1..}, D) \\ \sigma' = \sigma[a' \mapsto D\{a\}] \end{array} \right.$$

```

W(C, md1.., md'1.., D) = class D { that: C md''1.. }
  where m(x: t1): t2 {e} ∈ md1..
        md''1 = m(x: t'1): t'2 { ◀t'2▶ this.that.mt1→t2(◀t1▶ x) } ..
              if m(x: t'1): t'2 {e'} ∈ md'1..
              m(x: t1): t2 { this.that.mt1→t2(x) } ..
              otherwise
W★(C, md1.., D) = class D { that: C md'1.. }
  where md'1 = m(x: ★): ★ { ◀★▶ this.that.mt→t'(◀t▶ x) } ..
            if m(x: t): t' {e} ∈ md1..

```

■ **Figure 5** Behavioral cast semantics.

impossible for a to implement C' correctly, and early failure is indicated. Otherwise, the metafunction continues to generate a type wrapper. Class generation itself is delegated to the W metafunction. A W invocation, $W(C, \text{md}_{1..}, \text{md}'_{1..}, D)$, takes a class C , a fresh name D , and two method lists $\text{md}_{1..}$ and $\text{md}'_{1..}$, respectively the method of C and the methods of the type to enforce. The class generated by W will have adapter methods for each method m occurring in both $\text{md}_{1..}$ and $\text{md}'_{1..}$. Type mismatches between the wrapped object and the wrapping type are resolved with more behavioral casts. For methods that do not need to be adapted (methods only in $\text{md}_{1..}$), a simple pass-through method is generated. This method simply calls the wrapped object; itself referred to by the distinguished variable `that`.

If the target type is \star , the wrapper class is simpler. It only needs to check that method arguments match the types expected by the wrapped object. This is done by another behavioral cast. Return values are cast to \star .

For example, consider the following program which has two classes C and D . Even though C and D both have method a , they are not subtypes because the arguments to their m implementations are not related.

```

(◀C▶ (◀D▶ new C()).b(2))   where K = class C {
                                m(x: ★): ★ { x }
                                n(x: ★): ★ { x }
                                }
                                class D { m(x: Int): Int { x } }

```

The program starts with a `C`, casts it to `D`, and then back to `C`. The reason we generate pass-through methods (the wrapper that enforces type `D` has a method `n`) is that without them, the method `n` would be “lost”. Without the pass-through method, it would be impossible to cast back to `C`, as the wrapper would only have an implementation for `m`. Wrappers with this semantics are referred to as opaque, as it is not possible to see methods of the underlying object through them. In contrast, `KafKa` uses transparent wrappers. To illustrate what this looks like, the following class `E` is generated by the cast from `C` to `D`:

```

class E {
  that: C
  m(x: Int): Int { ◀Int▶ that.m_{★→★}(◀★▶ x) }
  n(x: ★): ★ { that.n_{★→★}(x) }
}

```

By keeping `n` present, it is possible to return an instance of `E` to `C` again. If we were to remove `n`, then `E` would no longer be convertible back to `C` again.

4.2 Type soundness

The `KafKa` type soundness theorem ensures that a well-formed program can only get stuck at a dynamic invocation, a subtype cast, or a behavioral cast, and only there when justified.

► **Theorem 1** (*KafKa type soundness*). *For every well-formed state $K \in \sigma \checkmark$ and well-typed expression $\emptyset \Sigma K \vdash e : t$, where heap typing Σ is obtained by mapping the class of each object to the corresponding address, one of the following holds:*

- *There exists some reference a such that $e = a$.*
- *$K \in \sigma \rightarrow K' \in \sigma'$, where $K' \in \sigma' \checkmark$, $\emptyset \Sigma' K' \vdash e' : t$, σ' has all of the values of σ , Σ' has all of the types of σ' , and K' has all of the classes of K .*
- *$e = E[a@m_{★→★}(a')]$ and a refers to an object without a method m .*
- *$e = E[(C) a]$, and a refers to an object whose class is not a subtype of C .*
- *$e = E[◀C▶ a]$, and C contains a method that a does not.*

The proof is mostly straightforward, with one unusual case, centered around the `bcast` metafunction. When the `bcast` metafunction is used to generate a wrapper class, which is then instantiated, producing a new class table and heap, we must then show that the new class table is well formed, that the new heap is also well formed, and that the new wrapper is a subtype of the given type `C`. Proving these properties is relatively easy. Class table well-formedness follows by construction of the wrapper class and by well-formedness of the old class table. Heap well-formedness follows by well-formedness of the class table, construction of the new heap, and well-formedness of the old heap. Proving that the type of the wrapper is a subtype of the required type proceeds by structural induction over the required type. The proof of soundness has been formalized in `Coq` and is available in the supplementary material. The proof relies on two axioms dealing with recursive structural subtyping. We did not prove these as they have been shown in prior work [14].

4.3 Discussion

The design of `KafKa`'s two invocation forms bears discussion. In some previous works, dynamic invocation has been implemented by a combination of a cast and a statically typed call. In our case, following this approach would require creating a type for each invocation (as was done in [30]). Instead, providing a dynamic invocation form seemed more natural. The use of explicitly typed invocation is a result of our desire to be able to rule out more errors statically. Whenever a translation can generate a static invocation, the soundness result ensure that the call will succeed. But, some methods need to be called from both a typed and untyped context, to achieve this we generate two versions of the method and leverage the difference between typed and untyped calls to express invocations occurring in each context.

One of our requirements for `KafKa` was that it support transparent wrappers as these are needed for the behavioral approach. The combination of structural subtyping and dynamic class generation allows to generate subtypes on the fly. These subtypes have all the methods of the target type plus some new ones. The choice of having fields be private means that the wrappers do not have to special case field access. If fields were accessible from outside the object, some more complex rewriting would have to be used.

`KafKa` was intended to match the intermediate languages of commercial VMs. To validate this, we implemented a compiler from `KafKa` to `C#`.² The only challenge was due to subtyping. `KafKa` uses structural typing, while `C#` is nominal, and `KafKa` allows methods in subtypes to be contra-variant in argument and co-variant in return type, while `C#` requires invariance.

Implementing structural subtyping on top of a nominally typed language is tricky. Structural types create implicit subtyping relationships, which the nominal type system expects to be explicit. Prior work used reflection and complex run-time code generation [9], but this is needlessly complex for a proof of concept. Instead, we reify the implicit relationships introduced by structural subtyping into explicit nominal relationships by generating interfaces. Given two classes `C` and `D`, where $K \vdash C <: D$ holds, we generate two interfaces `CI` and `DI`, where `CI` is declared to extend `DI`. As a result, if two types are subtypes, their corresponding `C#` interfaces will be as well. The next problem is that `KafKa` allows subtype methods to be contra-variant in argument and co-variant in return types. As a result, a single method in `CI` may not be sufficient to implement `DI`. We solve this by having every class's `C#` equivalent implement every interface explicitly, with each explicit implementation delegating to the real, most general, implementation.

Despite these issues, we were able to accurately translate `KafKa` types. We translate static and dynamic invocations into corresponding `C#` invocations since `C#` also has a dynamic type. The underlying run time can then use the translated `KafKa` types to perform method dispatch, while inserting dynamic checks wherever the `KafKa` code calls for an untyped invocation. This prototype shows that `KafKa` primitives are close to those of intermediate languages. As a result, the translation of gradual type systems to `KafKa` provides insight as to how they might be implemented.

² github.com/BenChung/GradualComparisonArtifact/netImpl

5 Translating Gradual Type Systems

“Was ist mit mir geschehen? dachte er. Es war kein Traum”

Equipped with the source and target languages, we can describe the gradual-to-statically typed translation from source to KafKa. Each semantics is translated through a function mapping well-typed surface programs into well-typed KafKa terms. The translation explicitly determines which type casts need to be inserted and the invocation forms to use. A type-driven translation will insert casts where the surface language used consistency.

5.1 Class Translation

The translations for surface level classes are shown in Fig. 6. Each class in the surface language translates to a homonymous KafKa class, retaining type names through translation. The grey background denotes code generated in the translation. The notation $e; e'$ denotes sequencing.

Optional. The optional approach provides no correctness guarantees. Retaining the surface type annotations through translation would not preserve this semantics, so we erase them. The resulting class has all fields, all method arguments, and all return values typed as \star .

Transient. In the transient approach, ensure that for any method call, the receiver does have a method with the corresponding name. To encode this within the KafKa type system

Optional:

$$\begin{aligned} \mathcal{O}[\text{class } C \{ \text{fd}_1.. \text{md}_1.. \}] &= \text{class } C \{ \text{fd}'_1.. \text{md}'_1.. \} \\ \text{where } \text{fd}'_1 &= f: \star .. \text{fd}_1 = f: t.. \\ \text{md}'_1 &= m(x: \star): \star \{e'\} .. \\ \text{md}_1 &= m(x: t_1): t_2 \{e\} \quad e' = \mathcal{O}[e] \end{aligned}$$

Transient:

$$\begin{aligned} \mathcal{T}[\text{class } C \{ \text{fd}_1.. \text{md}_1.. \}] &= \text{class } C \{ \text{fd}'_1.. \text{md}'_1.. \} \\ \text{where } \text{fd}'_1 &= f: \star .. \text{fd}_1 = f: t.. \\ \text{md}'_1 &= m(x: \star): \star \{ \langle t \rangle x; e'_1 \} .. \\ \text{md}_1 &= m(x: t): t' \{e\}.. \quad e'_1 = \mathcal{T}(e)_{x:t \text{ this: } C}^* .. \end{aligned}$$

Behavioral:

$$\begin{aligned} \mathcal{B}[\text{class } C \{ \text{fd}_1.. \text{md}_1.. \}] &= \text{class } C \{ \text{fd}_1.. \text{md}'_1.. \} \\ \text{where } \text{md}'_1 &= m(x: t): t' \{e'_1\} .. \\ \text{md}_1 &= m(x: t): t' \{e_1\} .. \quad e'_1 = \mathcal{B}[e_1]_{x:t \text{ this: } C} \end{aligned}$$

Concrete:

$$\begin{aligned} \mathcal{C}[\text{class } C \{ \text{fd}_1.. \text{md}_1.. \}] &= \text{class } C \{ \text{fd}_1.. \text{md}'_1.. \text{md}''_1.. \} \\ \text{where } \text{md}'_1 &= m(x: t_1): t_2 \{e'\} .. \\ \text{md}_1 &= m(x: t_1): t_2 \{e\}.. \quad e' = \mathcal{C}(e)_{\text{this: } C; x: t_1}^{t_2} .. \\ \text{md}''_1 &= m(x: \star): \star \{ \langle \star \rangle \text{this.m}_{t_1 \rightarrow t_2}(\langle t_1 \rangle x) \} \\ &\quad \text{if } t_1 \neq \star \\ &\quad \text{empty otherwise } .. \end{aligned}$$

■ **Figure 6** Translations for classes.

requires replacing all of the argument and return types with \star . This translation allows functions to be called under any type and to return values of any type. Casts to the erased types are then effectively shallow structural checks only. As there is no guarantee that fields contain values of their type, the translation sets their type to \star .

Behavioral. The behavioral approach guarantees soundness by wrapping values that cross type-untyped boundaries. Methods are preserved by the translation but bodies are translated.

Concrete. The concrete approach ensures that variables of non- \star types refer to subtypes of the given type. Each method appearing in the original class is retained as such with its body translated. Moreover, all typed methods could be called from an untyped context, so untyped variants are generated that guard the typed functions. These variants perform subtype casts on their arguments to ensure that they were given the right types, then call the guarded typed function.

5.2 Expression Translation

To accommodate differences between the gradual typing semantics, we use two different expression translation schemes. The first is a type-agnostic one, used for the optional approach. $\mathcal{O}[\![e]\!]$ denotes optional translation, where e is the target expression, and the result is a **KafKa** term. We use this form to simplify the optional translation, as it is ambivalent about the types of the expressions it is translating; the optional semantics simply eliminates all of them.

The second translation form is type-aware, used for the three other approaches. The type-aware translation has two forms, $\mathcal{S}[\![e]\!]_{\Gamma}$ and $\mathcal{S}(e)_{\Gamma}^t$, inspired by work on bidirectional type-checking [16]. The first form, $\mathcal{S}[\![e]\!]_{\Gamma}$, is analogous to the synthetic case in bidirectional type-checking. It is used for expressions without any specific required type. The second form, $\mathcal{S}(e)_{\Gamma}^t$, is used when e must have some type t . Analogous to the analytic case of bidirectional type-checking, this form applies when some enclosing expression has an expectation of the type of e . For example, it is used in translation of method arguments, which must conform to the types of the arguments to the method. We refer to this as assertive translation. These two forms allow us to identify where consistency was needed to conclude the surface level typing judgment.

Transient:

$$\begin{aligned} \mathcal{T}(e)_{\Gamma}^t &= e' & \text{where } K, \Gamma \vdash e : t' \quad K \vdash t' <: t \quad e' = \mathcal{T}[\![e]\!]_{\Gamma} \\ \mathcal{T}(e)_{\Gamma}^t &= \langle t \rangle e' & \text{where } K, \Gamma \vdash e : t' \quad K \vdash t' \not<: t \quad e' = \mathcal{T}[\![e]\!]_{\Gamma} \end{aligned}$$

Behavioral:

$$\begin{aligned} \mathcal{B}(e)_{\Gamma}^t &= e' & \text{where } K, \Gamma \vdash e : t' \quad K \vdash t' <: t \quad e' = \mathcal{B}[\![e]\!]_{\Gamma} \\ \mathcal{B}(e)_{\Gamma}^t &= \blacktriangleleft t \blacktriangleright e' & \text{where } K, \Gamma \vdash e : t' \quad K \vdash t' \not<: t \quad e' = \mathcal{B}[\![e]\!]_{\Gamma} \end{aligned}$$

Concrete:

$$\begin{aligned} \mathcal{C}(e)_{\Gamma}^t &= e' & \text{where } K, \Gamma \vdash e : t' \quad K \vdash t' <: t \quad e' = \mathcal{C}[\![e]\!]_{\Gamma} \\ \mathcal{C}(e)_{\Gamma}^t &= \langle t \rangle e' & \text{where } K, \Gamma \vdash e : t' \quad K \vdash t' \not<: t \quad e' = \mathcal{C}[\![e]\!]_{\Gamma} \end{aligned}$$

■ **Figure 7** Assertive translation.

The assertive translation of Fig. 7 is responsible for producing well-typed terms by adding casts into expressions where static types differ. The rules closely track the convertibility

relation of the surface language. Every type-driven translation has two cases. The first case is used when the required type happens to be a supertype of the expression's actual type, in which cast upcasting can happen implicitly and no further action is required. The second case handles typed-untyped boundaries, conversions to or from \star . The concrete and transient approaches both use the subtype cast operator to protect these boundaries, though the effective semantics are different; concrete retains the types that transient erases, so subtype casts in transient check structural compatibility (e.g. are all the needed methods present) alone whereas concrete subtype casts check the entire object's types. The behavioral approach instead inserts behavioral casts at boundaries.

The translation of field access appears in Fig. 8. The optional translation only inserts a cast to \star in front of uses `this` as a technicality required for statically typing terms. The transient translation casts variables and fields to their statically expected type, as their values may be of any type. In transient, however, subtype casts only check the structure of the types. The behavioral translation and the concrete translation leave both types of access intact.

<p>Optional:</p> $\begin{aligned} \mathcal{O}[\![x]\!] &= x \\ \mathcal{O}[\![this]\!] &= \langle \star \rangle this \\ \mathcal{O}[\![this.f]\!] &= this.f \end{aligned}$ <p>Behavioral:</p> $\begin{aligned} \mathcal{B}[\![x]\!]_{\Gamma} &= x \\ \mathcal{B}[\![this]\!]_{\Gamma} &= this \\ \mathcal{B}[\![this.f]\!]_{\Gamma} &= this.f \end{aligned}$	<p>Transient:</p> $\begin{aligned} \mathcal{T}[\![x]\!]_{\Gamma} &= \langle t \rangle x && \text{where } K, \Gamma \vdash x : t \\ \mathcal{T}[\![this]\!]_{\Gamma} &= this \\ \mathcal{T}[\![this.f]\!]_{\Gamma} &= \langle t \rangle this.f && \text{where } K, \Gamma \vdash this : C \quad f : t \in K(C) \end{aligned}$ <p>Concrete:</p> $\begin{aligned} \mathcal{C}[\![x]\!]_{\Gamma} &= x \\ \mathcal{C}[\![this]\!]_{\Gamma} &= this \\ \mathcal{C}[\![this.f]\!]_{\Gamma} &= this.f \end{aligned}$
--	--

■ **Figure 8** Translations variables and field access.

The translation for assignment is shown in Fig. 9. All the approaches translate the value only differing in the expected type. Behavioral and concrete require that the result has the statically known type, transient expects \star , and the optional semantics imposes no type requirement whatsoever.

<p>Optional:</p> $\mathcal{O}[\![this.f = e]\!] = this.f = e' \quad \text{where } e' = \mathcal{O}[e]$ <p>Transient:</p> $\mathcal{T}[\![this.f = e]\!]_{\Gamma} = this.f = e' \quad \text{where } K, \Gamma \vdash this : C \quad f : t \in K(C) \quad e' = \mathcal{T}[\![e]\!]_{\Gamma}^{\star}$ <p>Behavioral:</p> $\mathcal{B}[\![this.f = e]\!]_{\Gamma} = this.f = e' \quad \text{where } K, \Gamma \vdash this : C \quad f : t \in K(C) \quad e' = \mathcal{B}[\![e]\!]_{\Gamma}^t$ <p>Concrete:</p> $\mathcal{C}[\![this.f = e]\!]_{\Gamma} = this.f = e' \quad \text{where } K, \Gamma \vdash this : C \quad f : t \in K(C) \quad e' = \mathcal{C}[\![e]\!]_{\Gamma}^t$	
---	--

■ **Figure 9** Translations for assignment.

12:20 Gradual Typing for Objects

The translation for object creation, shown in Fig. 10, follows the same reasoning. It inserts casts for each argument to be the required type according to class translation.

The translations for invocation are shown in Fig. 11. The optional approach translates all invocations to dynamic invocation, as it cannot provide any static guarantee. In the concrete and behavioral approaches, since the static types are retained, arguments must be asserted to have the statically known type. In the transient semantics, the argument type is ignored, so the argument to a statically typed method call is only required to be of type \star , but the return type is checked. In all of the systems, if the type of the receiver is \star , dynamic invocation is used.

Optional:	$\mathcal{O}[\text{new } C(e_1..)]$	$=$	$\langle \star \rangle \text{new } C(e'_1..)$	where $e'_1 = \mathcal{O}[e_1] ..$
Transient:	$\mathcal{T}[\text{new } C(e_1..)]_\Gamma$	$=$	$\text{new } C(e'_1..)$	where $f_1 : t_1 \in K(C)$ $e'_1 = \mathcal{T}(e_1)_\Gamma^\star ..$
Behavioral:	$\mathcal{B}[\text{new } C(e_1..)]_\Gamma$	$=$	$\text{new } C(e'_1..)$	where $f_1 : t_1 \in K(C)$ $e'_1 = \mathcal{B}(e_1)_\Gamma^{t_1} ..$
Concrete:	$\mathcal{C}[\text{new } C(e_1..)]_\Gamma$	$=$	$\text{new } C(e'_1..)$	where $f_1 : t_1 \in K(C)$ $e'_1 = \mathcal{C}(e_1)_\Gamma^{t_1} ..$

■ **Figure 10** Translations for object creation.

Optional:	$\mathcal{O}[e_1.m(e_2)]$	$=$	$e'_1@m_{\star \rightarrow \star}(e'_2)$	where $e'_1 = \mathcal{O}[e_1]$ $e'_2 = \mathcal{O}[e_2]$
Transient:	$\mathcal{T}[e_1.m(e_2)]_\Gamma$	$=$	$e'_1@m_{\star \rightarrow \star}(e'_2)$	where $K, \Gamma \vdash e_1 : \star$ $e'_1 = \mathcal{T}[e_1]_\Gamma$ $e'_2 = \mathcal{T}(e_2)_\Gamma^\star$
	$\mathcal{T}[e_1.m(e_2)]_\Gamma$	$=$	$\langle D_2 \rangle e'_1.m_{D_1 \rightarrow D_2}(e'_2)$	where $K, \Gamma \vdash e_1 : C$ $e'_1 = \mathcal{T}[e_1]_\Gamma$ $e'_2 = \mathcal{T}(e_2)_\Gamma^\star$ $m(D_1) : D_2 \in K(C)$
Behavioral:	$\mathcal{B}[e_1.m(e_2)]_\Gamma$	$=$	$e'_1@m_{\star \rightarrow \star}(e'_2)$	where $K, \Gamma \vdash e_1 : \star$ $e'_1 = \mathcal{B}[e_1]_\Gamma$ $e'_2 = \mathcal{B}(e_2)_\Gamma^\star$
	$\mathcal{B}[e_1.m(e_2)]_\Gamma$	$=$	$e'_1.m_{D_1 \rightarrow D_2}(e'_2)$	where $K, \Gamma \vdash e_1 : C$ $e'_1 = \mathcal{B}[e_1]_\Gamma$ $e'_2 = \mathcal{B}(e_2)_\Gamma^{D_1}$ $m(D_1) : D_2 \in K(C)$
Concrete:	$\mathcal{C}[e_1.m(e_2)]_\Gamma$	$=$	$e'_1@m_{\star \rightarrow \star}(e'_2)$	where $K, \Gamma \vdash e_1 : \star$ $e'_1 = \mathcal{C}[e_1]_\Gamma$ $e'_2 = \mathcal{C}(e_2)_\Gamma^\star$
	$\mathcal{C}[e_1.m(e_2)]_\Gamma$	$=$	$e'_1.m_{D_1 \rightarrow D_2}(e'_2)$	where $K, \Gamma \vdash e_1 : C$ $e'_1 = \mathcal{C}[e_1]_\Gamma$ $e'_2 = \mathcal{C}(e_2)_\Gamma^{D_1}$ $m(D_1) : D_2 \in K(C)$

■ **Figure 11** Translations for function invocation.

5.3 Example

We illustrate the translation with the behavior of litmus program **L3**. The operational principle of **L3** is that it creates a new object (an instance of **C**), then uses an untyped intermediate to represent it as type **E**. Type **E** ascribes the wrong type for argument x , substituting **D** for the correct type **C**.

Source:

```
class F { m(x: E): E {x}   n(x: *): * {this.m(x)} }
```

Optional:

```
class F { m(x: *): * {x}   n(x: *): * {((*) this)@m_{*→*}(x)} }
```

Transient:

```
class F { m(x: *): * {⟨E⟩ x; ⟨*⟩ x}   n(x: *): * {⟨*⟩ x; ⟨*⟩ ⟨E⟩ this.m_{*→*}(⟨*⟩ ⟨*⟩ x)} }
```

Behavioral:

```
class F { m(x: E): E {x}   n(x: *): * {◀*▶ this.m_{E→E}(◀E▶ x)} }
```

Concrete:

```
class F { m(x: E): E {x}   n(x: *): * {(*) this.m_{E→E}(⟨E⟩ x)} }
```

■ **Figure 12** Class translation for litmus test **L3**.

Two of the gradual type systems notice this invalid type. Concrete errors on **L3** because **E** is not a subtype of **C**. With behavioral, the unused type ascription is saved as a wrapper and is enforced causing a run-time error.

While this reasoning provides an intuition, it provides few detail for which we turn to our formalism. We present the translation from the top, starting with classes in Fig. 12. The optional approach does no checking whatsoever, and simply erases types. Transient also erases types, but adds argument casts on method entry. In the case of m , argument x is checked to be of type **E**, as the translation of type **E** does not include types no type error will be reported. Behavioral retains typed methods but adds behavioral casts on untyped methods. The concrete semantics retains typed methods, and adds a subtype cast when a variable of type $*$ is passed to a method that expects and **E**.

Fig. 13 presents the translation of class **E**. For the transient semantics, when x is cast to **E**, all of the types on **E** are erased. Casting to **E** is tantamount to asking for the existence of the method m . In contrast, the concrete semantics retains the types of m . A cast to **E** is equivalent to checking if a method m that takes and returns an **E** exists. This comes at the cost of the ability to migrate between untyped and typed code. Suppose that both the optional and concrete versions of **E** existed, under a different name **F**. In that program, only the concrete version of **E** could be used with the concrete version of **F**. Despite implementing the same behavior, Behavioral uses the same representation for **E** as concrete. The behavioral cast allows to use any value that behaves like an **E**.

To examine the operation of the behavioral cast in more detail, Fig. 14 depicts the wrapper classes generated at the cast from **C** to $*$ and from it to **E**. Class C_1 takes an instance of **C** and makes it safe against use as $*$. In behavioral, no typed invocations can be made on

Source:`class E { m(x: D): D {x} }`**Optional:**`class E { m(x: *): * {x} }`**Transient:**`class E { m(x: *): * {⟨E⟩ x; ⟨*⟩ x} }`**Behavioral:**`class E { m(x: E): E {x} }`**Concrete:**`class E { m(x: E): E {x} }`■ **Figure 13** Translation of E in litmus test L3.`class C { a(x: C): C {x} }``class E { a(x: D): D {x} }``class C1 { that : C``a(x: *): * {◀*▶ this.that.aC→C(◀C▶ x)} }``class C2 { that : C1``a(x: D): D {◀D▶ this.that.a*→*(◀*▶ x)} }`■ **Figure 14** Behavioral wrappers.

a value that was cast to $*$ (and not cast to some type somewhere); only untyped invocations are allowed. As a result, the wrapper need only generate an untyped version of C 's method a , which calls the underlying C instance's a (adding suitable casts). The second wrapper class C_2 takes the C_1 wrapper and casts it back to E . This wrapper takes the untyped implementation of a and wraps it again, calling it with an argument cast to $*$ and casting the return to D .

5.4 Discussion

These translations explicit the enforcement machinery of each of the four approaches. Programs can get stuck at dynamic invocation and casts. Inspecting where these are inserted gives a precise account of what constitutes an error in each gradual type system.

Our account of the behavioral approach matches its implementation in Typed Racket. However, one could imagine a slightly less restrictive implementation, one which does not have a check for method names at wrapper creation. That check is pragmatic but perhaps too strict – it will rule out programs that may be fine just because a method is missing. One could have a wrapper that simply reports an error if a missing method is called.

Performance is a perennial worry for implementers of gradual type systems. It is difficult to guess how a highly optimizing language implementation will perform, as these implementations are likely to optimize away the majority of the casts and dynamic dispatches. Consider the progress in the performance of Typed Racket reported in the literature [22, 2]. What we can tell by looking at the translations is that with optional there is no obvious benefit or cost to having type annotations. Transient has checks on reads, which are common, and typed function calls. Furthermore, those checks are needed even if the entire program is typed. Both concrete and behavioral can benefit from type information in typed code. The difference is that the cost of boundary crossing are low in the concrete approach, as it uses a subtype check whereas behavioral requires allocation of a wrapper. Wrappers may also complicate the task of devirtualization and unboxing.

6 Conclusion

This paper introduced `KafKa`, a framework for comparing the design of gradual type systems for object-oriented languages. Our approach is to provide translations with different gradual semantics from a common surface language into `KafKa`. These translations highlight the different run-time enforcement strategies deployed by the languages under study. The differences between gradual type systems are highlighted explicitly by the observable differences of their behavior in our litmus tests, demonstrating how there is no consensus on the meaning of error. These litmus tests motivated the need to have a common framework to explore the design space.

`KafKa` demonstrates that to express the different gradual approaches, one needs a calculus with two casts (structural and behavioral), two invocation forms (dynamic and static), the ability to extend the class table at run-time, and wrappers that expose their underlying unwrapped methods. We provide a mechanized proof of soundness for `KafKa` that includes run-time class generation. We also demonstrate that `KafKa` can be straightforwardly implemented on top of a stock virtual machine.

A open question for gradual type system designers is performance of the resulting implementation. Performance remains a major obstacle to adoption of approaches that attempt to provide soundness guarantees. Under the optional approach, types are removed by the translation; as a result, performance will be identical to that of untyped code. The transient approach checks types at uses, so the act of adding types to a program introduces more casts and may slow the program down (even in fully typed code). In contrast, the behavioral approach avoids casts in typed code. The price it pays for this soundness, however, is that heavyweight wrappers inserted at typed-untyped boundaries. Lastly, the concrete semantics is also sound and has low overheads, but comes at a cost in expressiveness and ability to migrate from untyped to typed.

Going forward, there are several issues we wish to investigate. We do not envision that supporting nominal subtyping within `KafKa` will pose problems, it would only take adding a nominal cast and changing the definition of classes. Then nominal and structural could coexist. A more challenging question is how to handle the intricate semantics of Monotonic Reticulated Python. For these we would need a somewhat more powerful cast operation. Rather than building each new cast into the calculus itself, it would be interesting to axiomatize the correctness requirements for a cast and let users define their own cast semantics. The goal would be to have a collection of user defined pluggable casts within a single framework.

References

- 1 Esteban Allende, Oscar Callaú, Johan Fabry, Éric Tanter, and Marcus Denker. Gradual typing for Smalltalk. *Science of Computer Programming*, 96, 2014. doi:10.1016/j.scico.2013.06.006.
- 2 Spenser Bauman, Carl Friedrich Bolz-Tereick, Jeremy Siek, and Sam Tobin-Hochstadt. Sound gradual typing: Only mostly dead. *Proc. ACM Program. Lang.*, 1(OOPSLA), 2017. doi:10.1145/3133878.
- 3 Gavin Bierman, Martin Abadi, and Mads Torgersen. Understanding TypeScript. In *European Conference on Object-Oriented Programming (ECOOP)*, 2014. doi:10.1007/978-3-662-44202-9_11.

- 4 Gavin M. Bierman, Erik Meijer, and Mads Torgersen. Adding dynamic types to C#. In *European Conference on Object-Oriented Programming (ECOOP)*, 2010. doi:10.1007/978-3-642-14107-2_5.
- 5 Bard Bloom, John Field, Nathaniel Nystrom, Johan Östlund, Gregor Richards, Rok Strnisa, Jan Vitek, and Tobias Wrigstad. Thorn: Robust, concurrent, extensible scripting on the JVM. In *Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 2009. doi:10.1145/1639950.1640016.
- 6 Gilad Bracha. Pluggable type systems. In *OOPSLA 2004 Workshop on Revival of Dynamic Languages*, 2004. doi:10.1145/1167473.1167479.
- 7 Gilad Bracha and David Griswold. Strongtalk: Typechecking Smalltalk in a production environment. In *Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 1993. doi:10.1145/165854.165893.
- 8 Luca Cardelli. Amber. In *LITP Spring School on Theoretical Computer Science*, pages 21–47. Springer, 1985.
- 9 Gilles Dubochet and Martin Odersky. Compiling structural types on the JVM: A comparison of reflective and generative techniques from Scala’s perspective. In *Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems (ICOOPLS)*, 2009. doi:10.1145/1565824.1565829.
- 10 Robert Findler and Matthias Felleisen. Contracts for higher-order functions. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, 2002. doi:10.1145/581478.581484.
- 11 Robert Bruce Findler, Matthew Flatt, and Matthias Felleisen. Semantic casts: Contracts and structural subtyping in a nominal world. In *European Conference on Object-Oriented Programming (ECOOP)*, 2004. doi:https://doi.org/10.1007/978-3-540-24851-4_17.
- 12 Ben Greenman and Matthias Felleisen. A spectrum of soundness and performance. *Proc. ACM PL (ICFP)*, to appear, 2018.
- 13 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), 2001. doi:10.1145/503502.503505.
- 14 Timothy Jones and David J. Pearce. A mechanical soundness proof for subtyping over recursive types. In *Workshop on Formal Techniques for Java-like Programs (FTJFP)*, 2016. doi:10.1145/2955811.2955812.
- 15 Fabian Muehlboeck and Ross Tate. Sound gradual typing is nominally alive and well. *Proc. ACM Program. Lang.*, 1(OOPSLA), 2017. doi:10.1145/3133880.
- 16 Benjamin C. Pierce and David N. Turner. Local type inference. In *Symposium on Principles of Programming Languages (POPL)*, 1998. doi:10.1145/345099.345100.
- 17 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), 2017. doi:10.1145/3133879.
- 18 Gregor Richards, Francesco Zappa Nardelli, and Jan Vitek. Concrete types for TypeScript. In *European Conference on Object-Oriented Programming (ECOOP)*, 2015. doi:10.4230/LIPIcs.ECOOP.2015.76.
- 19 Jeremy Siek. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, 2006. URL: http://ecee.colorado.edu/~siek/pubs/pubs/2006/siek06_gradual.pdf.
- 20 Jeremy Siek and Walid Taha. Gradual typing for objects. In *European Conference on Object-Oriented Programming (ECOOP)*, 2007. doi:10.1007/978-3-540-73589-2_2.
- 21 Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. Refined criteria for gradual typing. In *Summit on Advances in Programming Languages (SNAPL)*, 2015. doi:10.4230/LIPIcs.SNAPL.2015.274.


- 22 Asumu Takikawa, Daniel Feltey, Ben Greenman, Max New, Jan Vitek, and Matthias Felleisen. Is sound gradual typing dead? In *Symposium on Principles of Programming Languages (POPL)*, 2016. doi:10.1145/2837614.2837630.
- 23 Asumu Takikawa, T. Stephen Strickland, Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. Gradual typing for first-class classes. In *Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 2012. doi:10.1145/2398857.2384674.
- 24 The Dart Team. Dart programming language specification, 2016. URL: <http://dartlang.org>.
- 25 The Facebook Hack Team. Hack, 2016. URL: <http://hacklang.org>.
- 26 Sam Tobin-Hochstadt and Matthias Felleisen. Interlanguage migration: from scripts to programs. In *Symposium on Dynamic languages (DLS)*, 2006. doi:10.1145/1176617.1176755.
- 27 Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of typed Scheme. In *Symposium on Principles of Programming Languages (POPL)*, 2008. doi:10.1145/1328438.1328486.
- 28 Michael Vitousek, Andrew Kent, Jeremy Siek, and Jim Baker. Design and evaluation of gradual typing for Python. In *Symposium on Dynamic languages (DLS)*, 2014. doi:10.1145/2661088.2661101.
- 29 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 *Symposium on Principles of Programming Languages (POPL)*, 2017. doi:10.1145/3009837.3009849.
- 30 Tobias Wrigstad, Francesco Zappa Nardelli, Sylvain Lebesne, Johan Östlund, and Jan Vitek. Integrating typed and untyped code in a scripting language. In *Symposium on Principles of Programming Languages (POPL)*, 2010. doi:10.1145/1706299.1706343.

Dependent Types for Class-based Mutable Objects

Joana Campos

LASIGE, Faculdade de Ciências, Universidade de Lisboa, Portugal


jcampos@lasige.di.fc.ul.pt

 <https://orcid.org/0000-0002-2185-8175>

Vasco T. Vasconcelos

LASIGE, Faculdade de Ciências, Universidade de Lisboa, Portugal

vv@di.fc.ul.pt

 <https://orcid.org/0000-0002-9539-8861>

Abstract

We present an imperative object-oriented language featuring a dependent type system designed to support class-based programming and inheritance. Programmers implement classes in the usual imperative style, and may take advantage of a richer dependent type system to express class invariants and restrictions on how objects are allowed to change and be used as arguments to methods. By way of example, we implement insertion and deletion for binary search trees in an imperative style, and come up with types that ensure the binary search tree invariant. This is the first dependently-typed language with mutable objects that we know of to bring classes and index refinements into play, enabling types (classes) to be refined by indices drawn from some constraint domain. We give a declarative type system that supports objects whose types may change, despite being sound. We also give an algorithmic type system that provides a precise account of quantifier instantiation in a bidirectional style, and from which it is straightforward to read off an implementation. Moreover, all the examples in the paper have been run, compiled and executed in a fully functional prototype that includes a plugin for the Eclipse IDE.

2012 ACM Subject Classification Software and its engineering → Semantics

Keywords and phrases dependent types, index refinements, mutable objects, type systems

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2018.13

Supplement Material ECOOP Artifact Evaluation approved artifact available at <http://dx.doi.org/10.4230/DARTS.4.3.1>

Acknowledgements LASIGE Research Unit, ref. UID/CEC/00408/2013

1 Introduction

Dependent types constrain types with values that specify intrinsic properties of programs. These sorts of types can represent concisely a number of invariants and prevent some classes of errors at compile time, rather than at runtime, which constitutes a step towards more reliable software. A key reason why dependent types are interesting is that they are a smooth extension of simple types. For example, using dependent types it becomes possible to express the non-negative balance b of a bank account as simply $\text{Account}(b)$, as well as ordered data structures, such as a binary search tree $\text{BST}(l, u)$ whose elements can find a place within some minimum (l) and maximum (u) keys.

Previous work has shown how far one can go with dependent types in the context of logic and functional languages [2, 4, 18, 33, 53]. Agda [37], DML [51] and Idris [5] are noteworthy



© Joana Campos and Vasco T. Vasconcelos;

licensed under Creative Commons License CC-BY

32nd European Conference on Object-Oriented Programming (ECOOP 2018).

Editor: Todd Millstein; Article No. 13; pp. 13:1–13:28

Leibniz International Proceedings in Informatics



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



examples of functional languages that include advanced features and dependent types. Less work has been done in an imperative setting [9, 48], and none (that we know of) in class-based object-oriented programming with mutable objects.

While the benefits of a dependently-typed system in functional programming – increasing expressiveness and safety – are exactly the same for object-oriented programming, how to smoothly combine dependent types and mutable objects is still an open problem. For example: What object invariants to enforce with dependent types, and when to enforce them? How to track state in order to know which variables are modified and how? How to achieve the “safe substitutability principle” [31] with dependent types? The extra complexity that comes with developing the metatheory for a calculus that captures with dependent types the essential (but non-trivial) features of object orientation, such as mutable state and inheritance with subtyping, is challenging and requires a different approach.

In this paper, we formulate Dependent Object-oriented Language (DOL) as a smooth extension of simply typed Java-like languages that addresses the challenges of combining dependent types and object orientation. DOL offers a middle ground between traditional type systems and verification techniques, along the lines suggested by Leino [30] when discussing future challenges of the ESC static checker and the need to explore “more-than-types systems” to enforce program invariants. As argued, static verification is a powerful approach, despite being unsound, designed for “finding bugs in a program”, not for providing the guarantees of a type system, yet significantly more expressive than types. As it so happens, formal verification is not always suitable and is still too costly for mainstream adoption, often requiring prior training in logic or theorem proving. Without giving up soundness or falling back on dynamic checks, DOL is closer to existing programming methodologies, capturing via types a subset of the ESC properties.

We present a solution that allows programmers to start with standard types, writing code in an imperative style, and add more type information so as to gain additional guarantees. From simply typed, the type systems of mainstream object-oriented languages have already become more expressive and complex, namely when generics were introduced. DOL goes a step further introducing a restricted form of dependent types to enable special terms, called indices, to be parameters to classes and methods. By restricting the domain of the constraint language to that of linear inequalities over the integers, along the lines of DML [53], we render DOL’s type system decidable. The programmer simply needs to abstract the class declaration on properties they want to capture. For example, a class `Account` may be declared as follows: `class Account(b:natural){ balance: Integer(b)... }` where `natural` is a subset type that abbreviates `{x:integer | x ≥ 0}`. The index variable `b` is used to sharpen the type of fields and methods defined in `Account`, so that the typechecker can enforce through types a behaviour that forbids overdrafts. We say that `Account` defines a family of classes representing bank accounts whose instances can have many types, including the concrete type `Account(100)` obtained by instantiating `b` with `100`. Like generics, types in DOL support a variety of arguments. The difference is that in DOL the arguments to types are index terms that satisfy the specified constraints.

In certain states, some methods must not be available at the risk of violating object invariants. DOL provides support for the specification of method availability through fine-grained method signatures. For a `withdraw` method in the `Account` class, the signature should be roughly “`withdraw` takes an `Integer(m)` where $0 \leq m \leq b$ on any `Account(b)` that becomes `Account(b - m)`”. The typechecker statically tracks objects and any state change, guaranteeing that calling `withdraw` with an invalid argument leads to a type error caught by the compiler. Moreover, to enforce behavioural subtyping, a subclass may reuse `Account` by declaring `extends Account(b)` which ensures that the invariant of the superclass is preserved in the subtype.

Given that objects may be mutable, aliasing in a language that allows types to change can result in a program “getting stuck”: if an `Account` object is aliased and its balance is changed by foreign code, reading from the alias will produce an unexpected result. In DOL, type varying objects are alias protected via a linear type discipline. A distinct category of type invariant, shared objects is allowed to coexist, but cannot subvert the linear system.

As a specification of typechecking, we give a declarative type system which extends with indices the Java notion of class types that take the form of $C\bar{i}$. From the dependent type theory, the system generalises simple function spaces to dependent function spaces $\Pi a : I.T$ where the result type T can depend on the value of the argument a , restricted to special terms of index type I . Similarly, dependent sum types, written $\Sigma a : I.T$, generalise ordinary product types restricted to some constraint domain. The language also includes union types of the form $T + U$ that eliminate the need of the unsafe null value. Moreover, the type system is able to record changes to mutable state. The calculus is a significant contribution of this paper, since it features the desirable property of type soundness, expressed via subject reduction and progress.

Then, we give an algorithmic type system which modifies the rules that require guessing quantifier instantiation [15, 16] and applies bidirectional typechecking [39] in order to distinguish rules that synthesize types from those that check terms against types already known. From this precise algorithm it is straightforward to read off an implementation.

We make the following contributions:

1. In contrast to other extensions to object-oriented programming, we define types that capture both the *immutable* and *mutable* state of objects (cf. [38]). A combination of index refinements and method signatures featuring input and output types enables a smooth integration of dependent types and class-based mutable as well as immutable, shared objects.
2. We let the type of an object change throughout the program based on a sound type system whereby a linear type discipline enforces unique references to type varying objects.
3. We provide support for single class inheritance as long as the subtype satisfies the index constraints defined by its supertype, and the inherited specifications remain meaningful in the context of the subclass.
4. We give a precise algorithm that is both sound and complete. The algorithm has an implementation in a prototype compiler for DOL that includes a plugin for the Eclipse IDE, a development tool that is widely used in the context of object-oriented languages but still new for dependently-typed languages.

2 DOL by Example

A class in DOL is declared just like any other class in a Java-like language, except that index variables may be introduced in the header and be used within the class to constrain member types. The class body contains fields and methods, including a constructor method named `init`. Like Java, DOL supports single class inheritance using the optional `extends` declaration. If omitted, the class is derived from the default superclass \top , a concrete class which has no fields or methods, except for the constructor.

2.1 Bank Account

Figure 1 defines the indexed class `Account` and its subclass `PlusAccount`. Notice that if we omit the extra type annotations in the example, we get plain Java-like code, with `Account` being simply a class type. However, when indexed, the class name `Account` denotes a family of

13:4 Dependent Types for Class-based Mutable Objects

```

1 class Account(b:natural) {
2   balance: Integer(b)
3
4   init(): Account(0) =
5     balance := 0
6
7   ⟨m:natural⟩
8   [Account(b) ↔ ⟨b+m⟩]
9   deposit(amount: Integer(m)) =
10    balance := balance + amount
11
12  ⟨m:natural{m≤b}⟩
13  [Account(b) ↔ ⟨b-m⟩]
14  withdraw(amount: Integer(m)) =
15    balance := balance - amount
16
17  getBalance(): Integer(b) =
18    balance
19 }
20 class PlusAccount(s,c,b:natural)
21   extends Account(b) {
22   savings: Integer(s)
23   checking: Integer(c)
24
25   init(): PlusAccount(0,0,0) =
26     balance, savings, checking := 0
27
28   ⟨m:natural⟩
29   [PlusAccount(s,c,b) ↔ ⟨s+m,c,b+m⟩]
30   deposit(amount: Integer(m)) =
31     super.deposit(amount);
32     savings := savings + amount
33
34   ⟨m:natural⟩
35   [PlusAccount(s,c,b) ↔ ⟨s,c+m,b+m⟩]
36   deposit2Checking(amount: Integer(m)) =
37     super.deposit(amount);
38     checking := checking + amount
39
40   ⟨m:natural{m≤b ∧ b=s+c}⟩
41   [PlusAccount(s,c,b) ↔
42     ⟨max(s-m,0),min(c,c-m+s),b-m⟩]
43   withdraw(amount: Integer(m)) =
44     super.withdraw(amount);
45     if amount ≤ savings {
46       savings := savings - amount
47     } else {
48       checking := checking - amount + savings;
49       savings := 0
50     }
51 }

```

■ **Figure 1** The indexed class `Account` and its subclass `PlusAccount`.

classes. Instantiations, or concrete classes, represent bank accounts that cannot be overdrawn, and may have many types, namely `Account(0)`, `Account(1)`, ... where the occurrence of the index variable introduced in the class header is replaced by the corresponding value (an index term). State is somehow exposed in types through indices, but fields are always *private* to a class, even if we do not use the corresponding keyword.

The special `init` method behaves as a typical constructor that initialises fields, creating a fresh object assigned the proper (or concrete) type `Account(0)` (line 4). One consequence of objects having different types is that the compiler must track state changes throughout the program, namely when client code creates an account object and calls methods on it:

```

acc := new Account(); // acc: Account(0)
acc.deposit(100);    // acc: Account(100)
acc.withdraw(30)    // acc: Account(70)

```

State Modifying Methods. We give indexed signatures to methods that are defined in indexed classes. For example, the `withdraw` method (lines 12–15) must be invoked on a receiver of type `Account(b)`, accepts an amount of type `Integer(m)`, modifies the type of the receiver from the initial `Account(b)` to the final `Account(b-m)`, and does so for any amount m that is a natural number smaller or equal to the balance. In the formal language, the method type, written $\Pi m : \{x : \text{integer} \mid 0 \leq x \leq b\}.T$, is a universal type that binds the index variable m in a type T (where T represents the types of the implicit and explicit parameters and the return type from the example), so that one can mention m in T . The scope of the index variable m is therefore local; it may appear in the method signature, but not outside. The type `[Account(b) ↔ ⟨b-m⟩]`, read “`Account(b)` becomes `Account(b-m)`”, is an abbreviation for a pair of types. The first type is seen as the input type of the (implicit) receiver and the second

one is viewed as its output type. Finally, when a method does not explicitly declare a return type, the typechecker assumes the supertype `Top`.

To illustrate the precision of the types in DOL, here is a variant of the preceding example, changed by adding a second call to method `withdraw` that violates the object invariant:

```
acc := new Account(); // acc: Account⟨0⟩
acc.deposit(100);     // acc: Account⟨100⟩
acc.withdraw(70);    // acc: Account⟨30⟩
acc.withdraw(50)     // Type error: 50 > 30
```

Both `deposit` and `withdraw` are examples of methods that change state, which we sometimes call *type varying* methods, having to explicitly declare the input and output types of their implicit receivers. On the contrary, in *type invariant* methods, that is, methods whose input and output types coincide, receiver types may be omitted. The `getBalance` (lines 17–18) method provides one such example.

Base Types and Literals. Constants and operators are used in the programmer’s language only to make arithmetic and logic operations look more familiar, since they are not part of DOL’s core language. In fact, constants and operators are desugared into object references and method calls. Formally, `Integer` and `Boolean`, implemented natively, are families of classes. For example, the `Integer` “interface” includes the following types:

```
class Integer(i:integer) {
  init(): Integer(0)
  (j:integer)+(value: Integer(j)): Integer(i+j)
  (j:integer)≤(value: Integer(j)): Boolean(i≤j)
  ...
}
```

Each desugared object of a primitive class is assigned a singleton type, with the constants used in the examples representing the values on which the types depend. Technically, the argument `100` from an earlier example is an object reference of type `Integer⟨100⟩`, obtained by creating a new location, and subtraction is translated into the call `balance.minus(amount)` before typechecking.

Controlled Aliasing. Aliasing is part of what makes mutable objects useful in programming. However, shared state can be tricky to handle in a type system such as that of DOL, where the type of a variable may no longer be a fixed class type; instead, it may be a (dependent) type that changes throughout the program. In DOL, the potential sources of aliasing problems are assignment and parameter passing. We adopt a solution that uses linear control of those objects defined by type varying classes. We say that a class is type varying when at least one of its methods is type varying, giving different input and output types to its receiver. Because the `Account` class is type varying as per methods `deposit` and `withdraw`, the type system forbids creating aliases of instances of `Account`. Here is how DOL’s typechecker handles aliasing:

```
alias := acc; // alias: Account⟨30⟩
acc.withdraw(20); // Type error: acc has been consumed!
alias.withdraw(10) // alias: Account⟨20⟩
```

Instead of creating an alias, the assignment “consumes” variable `acc`, removing it from the typing context; hence, the call to `withdraw` in the second line is forbidden, with `alias` being the only variable available in the typing context.

Similarly, our type system ensures that parameters are used correctly by treating these references linearly when needed. To show that our language can be flexible, despite the linear

13:6 Dependent Types for Class-based Mutable Objects

restriction, we could add a `transferTo` method to debit some amount from the current account and credit into another account given as parameter, using the following implementation:

```
<a:natural,m:natural{m≤b}>
[Account⟨b⟩ ~> ⟨b-m⟩]
transferTo(other: Account⟨a⟩, amount: Integer⟨m⟩): Account⟨a+m⟩ =
  var local := other; // other has been consumed!
  withdraw(amount);
  local.deposit(amount);
  local
```

On the other hand, we say that a class is type invariant when its methods are type invariant, i.e. when the input and output types coincide (or are omitted) in all methods. The native `Integer` and `Boolean` provide two examples of such classes whose objects can be freely shared. Since each new assignment creates a new location, instances of these classes carry their types unchanged irrespective of being accessed or aliased.

Inheritance and Subtyping. Adapted from JML [13, 29], the `PlusAccount` class illustrates how DOL can achieve the “safe substitutability principle” [31] via indexed types. By declaring `extends Account⟨b⟩`, we make the subtype inherit the `Account`’s only field, as well as all of its methods (except the constructor). In `PlusAccount`, we declare two extra index variables, `s` and `c`, and use them to constrain fields `savings` and `checking` that hold two portions of the `balance`.

We can think of `PlusAccount` as extending the behaviour of `Account` by providing additional fields and methods. So, the `deposit` method directly inherited (if not overridden) from `Account` will be given the following type:

```
<m:natural>
[PlusAccount⟨s,c,b⟩ ~> ⟨s,c,b+m⟩]
deposit(amount: Integer⟨m⟩) = ...
```

However, we want relate the two new fields in the subclass with the superclass’s field by enforcing that $b=s+c$ via method signatures. We override the `deposit` method (lines 28–32) that adds the amount both to the account’s balance, by calling the superclass method, and the `savings` field. A new method `deposit2Checking` (lines 34–38) also adds the given amount to the `checking` field. The `withdraw` method (lines 40–50) must be redefined in order to take out the amount from each balance portion. DOL’s typechecker gives the index equations issued by types to an external constraint solver, and asks if they hold.

Common mistakes that violate the (inherited) invariant are readily detected. For example,

```
<m:natural{m≤s}>
[PlusAccount⟨s,c,b⟩ ~> ⟨max(s-m,0),min(c,c-m+s),b-m⟩]
withdraw(amount: Integer⟨m⟩) = ...
```

yields a type error, since a subtype cannot accept a *stronger* requirement, that is, it cannot accept less arguments as valid [31] (it should be clear that the constraint $m \leq s$ does not imply $m \leq b$ under the assumption that $b=s+c$). A subtler type error is found in the following variant:

```
<m:natural{m≤b}>
[PlusAccount⟨s,c,b⟩ ~> ⟨max(s-m,0),min(c,c-m+s),b-m⟩]
withdraw(amount: Integer⟨m⟩) = ...
```

The problem here is that the constraint $m \leq b$ does not relate the value of amount with the two portions of the balance, unlike the indices in the output type. Specifically, the index refinement does not provide enough evidence that allows the typechecker to conclude, after interaction with the solver, that the index term `min(c,c-m+s)` is a natural number that can safely replace the index variable `c` introduced in the class header.


```

19
20 class Node(l,k,u:integer{l ≤ k ≤ u}) {
21   key: Integer(k) // fields
22   left: Nil + Node(l,k1,u1)
23     where k1,u1:integer{l ≤ k1 ≤ u1 ≤ k}
24   right: Nil + Node(l1,k1,u)
25     where l1,k1:integer{k ≤ l1 ≤ k1 ≤ u}
26
27   ⟨v:integer⟩
28   init(value: Integer(v)): Node(v,v,v) =
29     key := value;
30     left, right := new Nil(), new Nil()
31
32   ⟨v:integer⟩
33   [Node(l,k,u) ~> ⟨min(l,v),k,max(u,v)⟩]
34   add(value: Integer(v)) = ...
35
36   ⟨v:integer⟩
37   [Node(l,k,u) ~> ⟨l,k1,u⟩
38     where k1:integer{l ≤ k1 ≤ u}]
39   deleteChild(value: Integer(v)) = ...
40 }

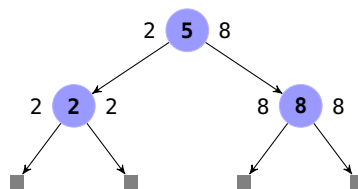
```

```

1 class BST(l,u:integer) {
2   root: Nil + Node(l,k,u)
3     where k:integer{l ≤ k ≤ u}
4
5   init(): BST(2,1) =
6     root := new Nil()
7
8   ⟨v:integer⟩
9   [BST(l,u) ~> ⟨min(l,v),max(u,v)⟩]
10  insert(value: Integer(v)) = ...
11
12  ⟨v:integer⟩
13  remove(value: Integer(v)) = ...
14 }
15
16 class Nil {
17   init(): Nil = skip
18 }

```

■ **Figure 2** Classes that implement a dependently-typed binary search tree.



■ **Figure 3** The diagrammatic representation of an object of type $\text{Node}\langle 2,5,8 \rangle$ where labels at each tree node denote the smallest and greatest keys appearing in the tree.

2.2 Binary Search Tree

Binary search trees can naturally be described by the discipline of dependent types [14, 28, 34, 47]: a binary search tree is either empty or nonempty in which case it has two subtrees that are binary search trees, and the key in the root node of the binary search tree is greater than all the keys appearing in its left subtree and smaller than all the keys appearing in its right subtree. This example shows that our type system can be precise and expressive while implementations remain as usual. The effort of programming in DOL is essentially to come up with the right type.

We implement the binary search tree in an imperative style, allowing subtrees to be modified *in place*. In Figure 2, we show the types, defining `BST` as the “public” family of classes that creates and manages both empty and nonempty trees using `Nil` (a proper class) and `Node` (a family of classes). Our binary search tree contains integer numbers included in the *loose* pair of bounds $\langle l, u: \text{integer} \rangle$ in the header of `BST` that can be used to define an *interval* $[l, u]$. Any element in a tree will find a place within the minimum (l) and maximum (u) keys.

While many approaches have been proposed [6, 8, 17] to handle Hoare’s billion dollar mistake [23], DOL provides an elegant solution using union types (cf. [24]) that enable programmers to build imperative linked data structures in a null-free style – object references are, after all, the only values in DOL. Union types (denoted by $\tau + u$) represent objects that can be of any of the specified types. Note that the lack of null in DOL means that every variable must be initialised, and that every variable of a union type must be analysed by way of a `case` construct before being used.

13:8 Dependent Types for Class-based Mutable Objects

So, class `BST` has a single field `root` that is either `Nil` or `Node`. A dependent existential quantified constructor (denoted by `where ...`) is used to keep track of the key, hidden in the field's type, so that the binary search tree invariant can be maintained. We write it as a dependent sum type in the formal language of the form $\Sigma k : \{x : \text{integer} \mid l \leq x \leq u\}. \text{Node } l \ k \ u$. Notice that this type conforms to the constraint $(l \leq k \leq u)$ issued by the signature of class `Node` that ensures the binary search order invariant.

The special `init` method (lines 5–6) creates an empty binary search tree to which we give the type `BST(2,1)`, making `root` an instance of `Nil`. When inserting a value in an empty tree, we replace an instance of `Nil` with an instance of `Node` with no children (a leaf). Then, when inserting in a nonempty tree, we recursively push the requirements of data inward, requiring that the value at each node falls within the interval $[l, u]$. For example, a type `BST(2,8)` may represent the binary search tree whose root of type `Node(2,5,8)` (depicted in Figure 3) issues the minimum and maximum keys outward. The value 5 stored in the root is not exposed, but is internally constrained by the tree bounds.

The `Node` class defines a field `key`, which holds the node value, and fields `left` and `right` that may represent the two subtrees. We use union and existential types, again pushing the data requirements inward to the types of the `left` and `right` fields. For example, the type of the `left` field (lines 22–23) enforces the fact that all the values appearing in the left subtree must be in the interval defined by $[l, k]$, so that the binary search tree invariant can be maintained. Similarly, the type of the `right` field enforces the fact that all the values appearing in this subtree must be included in $[k, u]$. The `init` method (lines 27–30) creates a leaf by accepting an integer value to be stored in the `key` field, making both `left` and `right` instances of `Nil`. By definition, leaf nodes are such that $l=k=u$. So, for example, `Node(2,2,2)` may be the type of the left subtree (a leaf) in Figure 3.

BST Insertion. We now define the `insert` method in the `BST` class, which takes as argument an integer value and provides a useful demonstration of a case discrimination construct:

```
<v:integer>
[BST(l,u) ~> <min(l,v),max(u,v)>]
insert(value: Integer<v>) =
  case root {
    Nil => root := new Node(value)
    Node => root.add(value)
  }
```

The `Node` class implements the main insertion algorithm. Its method `add` takes an argument similar to the one above, and also uses a case discrimination construct:

```
<v:integer>
[Node(l,k,u) ~> <min(l,v),k,max(u,v)>]
add(value: Integer<v>) =
  if value < key {
    case left {
      Nil => left := new Node(value)
      Node => left.add(value)
    }
  } else if value > key {
    case right {
      Nil => right := new Node(value)
      Node => right.add(value)
    }
  }
}
```

We add an element to the tree by comparing the `value` to the `key` stored at each node and recursively descending into the appropriate subtree until a leaf is reached that allows adding the new node.

The precise types given to the `BST` and `Node` classes allow the typechecker to detect a number of common programming errors. For example, the compiler will report a type error if we try to call the `add` method as follows:

```

<v:integer>
[BST(l,u) ~> <min(l,v),max(u,v)>]
insert(value: Integer<v>) =
  root.add(value)

```

Because the `root` field declares a union type, we cannot call a method directly on it; first, we must use a `case` construct to analyse its type and discover whether the object is an instance of `Nil` or `Node`, noting that, at each branch, the typechecker requires that `root` be bound to only one of the types which are subtypes of the union of types. This guarantees that either branch is taken and its execution succeeds.

Similarly, the compiler will object to the wrong conditional test below:

```

<v:integer>
[Node(l,k,u) ~> <min(l,v),k,max(u,v)>]
add(value: Integer<v>) =
  if value > key {
    case left {
      Nil => left := new Node(value)
      ...

```

Here, the compiler will report inconsistent constraints. The `case` construct is correctly used to find out that `left` is a `Nil`. Then, the assignment changes the type of the `left` field to `Node(v,v,v)` (which is the type given to it by `init`). However, the compiler assumes $v > k$ from the conditional test, after which will not be able to assert $v \leq k$ issued from the new type of the `left` field. Recall the constraints on the declared type of `left`, requiring its value be left-bounded by the minimum key ι (which has become v) and right-bounded by value k (known to be also v) such that $\iota \leq k$. Again, DOL relies on the external constraint solver to statically verify that the specified constraints hold.

BST Deletion. Deletion from the binary search tree may involve removing a key not only from the tree's leaf nodes but also from an interior node, which requires some sort of rearrangement of the tree structure. Moreover, unlike insertion, in which `min` and `max` could be used to issue the new value's standing vis-à-vis the minimum and maximum keys existing in the tree, deletion delivers the same binary search tree where the new minimum or maximum may be hidden in the subtrees.

However, we can still ensure via types that the tree after deletion is within the bounds, no matter where the key removal occurs (from a fringe or the middle of the tree). The `remove` method is implemented as usual:

```

<v:integer>
remove(value: Integer<v>) =
  case root {
    Nil => skip
    Node =>
      if root.isLeaf(value) {
        root := new Nil()
      } else {
        root.deleteChild(value)
      }
  }

```

13:10 Dependent Types for Class-based Mutable Objects

$P ::= \bar{L}$	(programs)
$L ::= \text{class } C : \Delta \text{ extends } T\{\bar{l} : \bar{T}\} \text{ is } \{\bar{M}\}$	(classes)
$M ::= m(x) = t$	(methods)
$T ::= C\bar{i} \mid \Pi a : I.T \mid \Sigma a : I.T \mid T + T$ $\mid T \times T \mid T \rightsquigarrow T \mid T \rightarrow T$	(types)
$t ::= x \mid f \mid \text{new } C() \mid f := t \mid t; t \mid m(t)$ $\mid f.m(t) \mid \text{case } f \text{ of } (C_k \Rightarrow t_k)_{k \in 1,2}$ $\mid \text{if } t \text{ then } t \text{ else } t \mid \text{while } t \text{ do } t$	(terms)
$\Delta ::= \epsilon \mid \Delta, a : I$	(index contexts)
$I ::= \text{integer} \mid \text{boolean} \mid \{a : I \mid p\}$	(index types)
$i ::= a \mid n \mid i \oplus i \mid p$	(index terms)
$p ::= \text{false} \mid \text{true} \mid \neg p \mid i \otimes i \mid p \otimes p$	(propositions)
$\oplus ::= + \mid -$	(arithmetic operators)
$\otimes ::= < \mid \leq \mid \doteq \mid \geq \mid >$	(relational operators)
$\odot ::= \wedge \mid \vee$	(logical operators)

■ **Figure 4** Top-level syntax.

3 The DOL Language

The core language is a desugared version comprising all the properties informally described in the examples. It builds on the core sequential language of Gay et al. [21], which allows us to simplify proofs while keeping them manageable. We adapt and extend that language in three ways. (1) We replace session types with dependent types and study the consequences of this idea. (2) We incorporate inheritance and nominal subtyping, a feature absent from the base language. (3) We combine linear and unrestricted objects in the formalisation, building a less restrictive type system than the original one.

3.1 Syntax

Following standard practice [21], the formal language omits some features of the practical syntax used in the examples, so as to simplify the proofs, even though our prototype includes them. Below, we summarize the main differences.

- Primitive values as used in the examples are translated into object references, which are the only values in our language, and all computations are performed by calling methods. This lightens the type system without affecting expressivity.
- All methods have exactly one parameter. A method written $m() = t$ abbreviates $m(\text{top}) = t$ where top of type Top is used as a dummy parameter. Defining methods that take an arbitrary number of parameters does not introduce any major technical challenge.
- Local variables are omitted, since they can be simulated by a parameter or extra fields.

We define the top-level syntax in Figure 4. Identifiers are drawn from the following disjoint countable sets: that of class names (denoted by B, C, D), that of fields (denoted by b, c, d), that of methods (denoted by f, g), that of object variables (denoted by x, y, o), and

that of index variables (denoted by a, b). Labels l identify class members, that can either be fields or methods. The metavariables T, U, V, W range over object types; I, J range over index types; and i, j range over index terms.

Programs P consist of collections of class declarations L . A class family, written `class C : Δ extends $T\{\bar{l} : \bar{T}\}$ is $\{\bar{M}\}$` , associates a class named C to an index context Δ , a supertype T , a sequence of member declarations $\bar{l} : \bar{T}$ (field and method signatures), and a sequence of method implementations \bar{M} . An index context maps index variables to index types, fixing the class family arity, with each entry having the form $a : I$. A concrete or proper type, written $C\bar{i}$, is obtained by instantiating a class family with indices in application position. Index variables in Δ can be used to constrain types inside the class, including that of the explicit superclass T , where T is of the form $D\bar{i}$. (As we will see later, this restriction is enforced by the typing rules.) Finally, a method is implemented separately from its signature as $m(x) = t$, where t is the method body and, for simplicity, x its single parameter.

Types. Types T either classify objects or build method signatures. They can be of the following seven forms:

- A type $C\bar{i}$ extends with indices the Java notion of class types.
- A universal dependent type constructor, written $\Pi a : I.T$, where a may occur free in T , is a type that maps elements of the index type I to elements in the type T . It is used to build method signatures.
- An existential type constructor, written $\Sigma a : I.T$, where a may occur free in T , also maps elements of the index type I to elements in the type T , with the index variable a representing some unknown value in T . It is used to represent undetermined properties of a concrete object type.
- A union type $T + T$ classifies the set of objects belonging either to the left or the right type. It is used to define a supertype grouping independently developed classes.
- A product type, written $T \times T$, is used in method signatures, with the first type classifying the current object `this`, implicitly passed to the method, and the second one classifying the only explicit parameter.
- A parameter type of the form $T \rightsquigarrow T$ relates the two components that classify the current object `this`, the input type and a possibly different output type.
- A method type, written $T \rightarrow T$, maps the type of the parameters to a return type.

Terms. Terms t are fairly standard, except for some restricted forms that allow the type system to record more precisely how the types of objects vary. The variable x denotes a parameter. There is no qualified $x.f$. Instead, field access, written f , is only defined for a shared field (a restriction enforced by the typing rules), or in combination with assignment, method calls and case constructs. This is part of the linear control of objects. All fields are private in the sense that every f always refers to a field of the current object (cf. [21]). Object creation (`new C ()`) does not take any parameters. Assignment ($f := t$) is defined in terms of a non-standard swap operation in the style of [21]. The operation assigns the value of t to the field f and returns the old value of f as its result. This prevents aliasing linear fields in terms such as $f_2 := (f_1 := t)$. The sequential term composition $(t; t)$ is standard. Method call is available both on the current object itself (a *self call*), written $m(t)$, and on a field of the current object `this`, written $f.m(t)$, but not on a parameter or an arbitrary term for that matter. This is because calling a method may change the type of the object on which the method is called. Note that the type system only records changes on the type of

13:12 Dependent Types for Class-based Mutable Objects

$T ::= \dots \mid C[F]$	(types)
$F ::= \{\bar{f} : \bar{T}\}$	(field types)
$r ::= o \mid r.f$	(paths)
$t ::= \dots \mid \text{return } t$	(terms)
$\theta ::= \epsilon \mid \theta, i/a$	(index substitutions)
$\Delta ::= \dots \mid \Delta, p$	(index contexts)
$\Gamma ::= \epsilon \mid \Gamma, x : T$	(object contexts)
$K ::= \star \mid \Pi a : I.K$	(kinds)
$h ::= \epsilon \mid h, o = R$	(heaps)
$R ::= C\{\bar{f} = \bar{o}\}$	(object records)
$S ::= (h * r, t)$	(states)
$\mathcal{E} ::= \lfloor _ \rfloor \mid f := \mathcal{E} \mid \mathcal{E}; t \mid m(\mathcal{E}) \mid f.m(\mathcal{E})$	(evaluation contexts)
$\mid \text{return } \mathcal{E} \mid \text{if } \mathcal{E} \text{ then } t \text{ else } t \mid \text{while } \mathcal{E} \text{ do } t$	

■ **Figure 5** Extended syntax, used only in the type system and operational semantics.

the current object `this`, the only one that can access its fields. To simplify, the `case` construct may only depend on a field, taking the form of `case f of $(C_k \Rightarrow t_k)_{k \in 1,2}$` where f plays the role of the binding occurrence in the branches. Conditionals and while loops are standard.

Index Refinements. Index types I comprise the integer and boolean types, as well as the subset type of the form $\{a : I \mid p\}$. Index terms i include some of the possible index constructs, namely variables, integer literals, arithmetic operations, and also propositions, which take the form of the truth values, the negation and linear inequalities. We omit functions `max` and `min` from the examples as they do not introduce any additional technical challenge.

3.2 Additional Syntax Not Available to Programmers

Figure 5 defines syntactic extensions required for the formal system only. The internal type $C[F]$ (cf. [21]) is an alternative form of an object type that contains the class name C and a record field typing F that provides types for all the fields of C , including the inherited ones. For example, $C[\{f_1 : T_1, f_2 : T_2\}]$ is the internal type of an object of C having two fields of types T_1 and T_2 , which may be defined either in C or in any of its superclasses. The internal type, used to classify the current object (`this`) in the context, cannot be the type of an arbitrary term, which never evaluates to `this` (as enforced by the typing rules). Instead, the purpose of the internal type is to allow the current object (`this`) to access its own fields, for typechecking assignment, method calls and case constructs, operations that may change its type through the field typing.

Terms evaluate to object references o , the only values in our language. To simplify, we do not define a separate syntactic category for object references. Instead, object references o are a subset of the variable names. Paths r in the style of [21] represent locations in the heap, formed by the top-level object followed by a sequence of an arbitrary number of fields. For example, if r indicates the path of the currently active object, when a method call on a

field f relative to r is entered, $r.f$ becomes the path that indicates the new current object that becomes active. The return term represents an ongoing method call during which the path changes as described above. Paths and the return term are constructs belonging to the operational semantics.

Substitutions θ map index variables to index terms. Index contexts Δ are extended to accept propositions p . Object contexts Γ map object variables to types.

Types are classified into kinds K , much the same way as terms are classified into types. Kind \star characterizes proper types, while kind $\Pi a : I.K$ classifies families of classes, i.e. types that have to be applied to index terms to form proper types. We only ever need to check for proper types (with kind \star). In fact, the only way to construct a type of a kind other than \star is by declaring an indexed class. So, in the bank account example, the class family `Account` has kind $\Pi b : \text{natural}.\star$, and an instantiation, say `Account 0`, of kind \star , denotes the proper type of an object reference.

A heap h is a mapping from object references o to object records R . We assume three special objects (`top` : `Top`, `false` : `Boolean false`, `true` : `Boolean true`) that are initially placed in the heap. The heap produced by the operation $h, (o = R)$ contains a new mapping from object reference o to record R . The operation of adding this binding to the heap h is only defined if $o \notin \text{dom}(h)$. Note that the order in h is irrelevant. Records R are instances of classes, represented by $C\{\bar{f} = \bar{o}\}$, comprising the class of the object followed by a mutable record mapping field names to object references.

The operational semantics is defined as a reduction relation on states S of the form $(h * r, t)$, consisting of a heap h , a path r that represents the current object, and a term t . Evaluation contexts \mathcal{E} are defined in the style of [46]. Intuitively, an evaluation context is a term with a hole $[_]$ at the point where the next reduction step must take place in a call-by-value evaluation order; $\mathcal{E}[t]$ is the term obtained by replacing the hole in \mathcal{E} by term t .

3.3 Static Semantics

We typecheck our language with respect to one index context Δ , and one object context Γ . The ordering is important in the index context, because of (index) variable-to-type dependencies, and irrelevant in the object context. For example, an index context $(\Delta_1, a : I, \Delta_2)$ is said to be well-formed if $a \notin \text{dom}(\Delta_1) \cup \text{dom}(\Delta_2)$ and $a \notin \text{FV}(\Delta_1)$; an index context such as $(c : \{b : \text{integer} \mid b \geq a\}, a : I)$ is ill-formed. We give the subtyping and typing rules for top-level terms in the sections that follow; we omit rules for the index language, kinding, context formation and typing. First, we give an overview of index refinements and substitution.

Index Refinements and Substitution. Our formulation of index refinements requires a way to somehow decide the semantically defined relation $\Delta \models p$ in the style of Xi and Pfenning [47, 52, 53]. The binding occurrences of index variables appear in subset types, and also in types and kinds in the object language. We say that a occurs bound in p within $\{a : I \mid p\}$, in T within $\Pi a : I.T$ and $\Sigma a : I.T$, and in K within $\Pi a : I.K$.

To simplify the proofs, and to avoid having to rename bound variables in substitution, we follow Barendregt's variable convention [3] whereby the names of bound variables must all be distinct from each other and from any other variables occurring free in terms and types.

We denote by $i_1[i_2/a]$ the capture-avoiding substitution of i_2 for the free occurrences of a in i_1 . Index substitutions are defined inductively on the structure of index terms. For example, $(i_1 + i_2)[i_3/a]$ is defined as $i_1[i_3/a] + i_2[i_3/a]$. A single index substitution is extended pointwise to multiple index substitution θ , which maps index variables to index terms, by defining $i\epsilon \triangleq i$ and $i_1([i_2/a], \theta) \triangleq (i_1[i_2/a])[\theta]$.

$$\boxed{\Delta \vdash T <: U} \text{ Under context } \Delta, \text{ type } T \text{ is a subtype of } U$$

$$\frac{\text{class } C : (\bar{a} : \bar{I}) \text{ extends } T\{_\} \text{ is } \{_\}}{\Delta \vdash C\bar{i} <: T[\bar{i}/\bar{a}]} \quad \Delta \vdash \bar{i} : \bar{I} \quad \Delta \vdash T[\bar{i}/\bar{a}] : \star \quad (\text{S-SUPER})$$

$$\frac{\Delta \models \bar{i} \doteq \bar{j} \quad \Delta \vdash C\bar{j} : \star}{\Delta \vdash C\bar{i} <: C\bar{j}} \quad (\text{S-APP}) \quad \frac{\Delta \vdash T[i/a] <: U \quad \Delta \vdash i : I}{\Delta \vdash \Pi a : I.T <: U} \quad (\text{S-IIL})$$

$$\frac{\Delta, a : I \vdash T <: U}{\Delta \vdash T <: \Pi a : I.U} \quad (\text{S-IIR}) \quad \frac{\Delta, a : I \vdash T <: U}{\Delta \vdash \Sigma a : I.T <: U} \quad (\text{S-SIL})$$

$$\frac{\Delta \vdash T <: U[i/a] \quad \Delta \vdash i : I}{\Delta \vdash T <: \Sigma a : I.U} \quad (\text{S-SIR}) \quad \frac{\Delta \vdash T_1 <: U \quad \Delta \vdash T_2 <: U}{\Delta \vdash (T_1 + T_2) <: U} \quad (\text{S-+L})$$

$$\frac{\Delta \vdash T <: U_k}{\Delta \vdash T <: (U_1 + U_2)} \quad (\text{S-+R}_k) \quad \frac{\Delta \vdash T_1 <: U_1 \quad \Delta \vdash T_2 <: U_2}{\Delta \vdash (T_1 \times T_2) <: (U_1 \times U_2)} \quad (\text{S-}\times)$$

$$\frac{\Delta \vdash \bar{T} <: \bar{U}}{\Delta \vdash C\{\{\bar{f} : \bar{T}\}\} <: C\{\{\bar{f} : \bar{U}\}\}} \quad (\text{S-RECORD}) \quad \frac{\Delta \vdash T_1 <: T_2 \quad \Delta \vdash T_2 <: T_3}{\Delta \vdash T_1 <: T_3} \quad (\text{S-TRANS})$$

■ **Figure 6** Subtyping rules.

The judgement for deriving θ is of the form $\Delta_1 \vdash \theta : \Delta_2$ where, under the assumptions in context Δ_1 , we think of Δ_2 as the input and θ as the output. The rules require that Δ_2 and θ have the same arity and that each substituent is well-formed in the context. Specifically, for each substitution i/a , there is an entry $a : I$ such that $\Delta_1 \vdash i : I$. As for index terms, application of a substitution θ to a type T , denoted by $T[\theta]$, is standard, defined inductively on the structure of T .

3.3.1 Subtyping

Term typing and method overriding rely on the subtyping relation defined as the reflexive and transitive closure of the inheritance relation as in Java, guided by the “safe substitutability principle” [31]. The judgement $\Delta \vdash T <: U$ asserts that T is a subtype of U under the assumptions in context Δ . We give the rules for subtyping in Figure 6.

All types in the top-level language are subject to subtyping, except for \rightsquigarrow and \rightarrow , since these types cannot arise from terms, and are not used to check method overriding (cf. Figures 9 and 10). The internal field typing is also subject to subtyping in order to check compatibility between fields of the same class. This relation is always derived with respect to the internal type of the current object (`this`), the only one that has access to its own fields.

S-SUPER is completely standard for object-oriented languages, adjusted to dependent types. Because class `Top` does not declare a supertype, it follows that `Top` is a supertype of every other type. By S-APP, subtyping is reflexive on class types, extended pointwise to all possible applications of the class type that satisfy the \models relation, and by rule S-TRANS, subtyping is transitive.

Regarding S-IIL and S-IIR, the left rule instantiates the index variable a to i in the subtype, while the right rule relates two types T and U provided the variable a does not appear free in T . The reasoning for S-SIL and S-SIR is similar, yet inverted. Following Barendregt’s variable convention [3], we implicitly assume that the variable a in the extended context of both S-IIR and S-SIL is distinct from all the variables already in Δ .

$$\begin{array}{c}
\boxed{\text{classof}(T) = C} \quad \text{classof}(C\bar{i}) = C \quad \text{classof}(\Sigma a : I.T) = \text{classof}(T) \\
\boxed{\text{fields}(T) = U} \quad \text{fields}(\text{Top}) = \text{Top}\{\{\}\} \quad \text{fields}(\Sigma a : I.T) = \Sigma a : I.\text{fields}(T) \\
\frac{\text{class } C : (\bar{a} : \bar{I}) \text{ extends } D\bar{j}\{\bar{f} : \bar{U}, \bar{m} : _ \} \text{ is } \{_ \} \quad \text{fields}(D\bar{j}[\bar{i}/\bar{a}]) = D[\{\bar{g} : \bar{V}\}]}{\text{fields}(C\bar{i}) = C[\{\bar{g} : \bar{V}\} \sqcup \{\bar{f} : \bar{U}[\bar{i}/\bar{a}]\}]} \\
\boxed{\text{mtype}(m, C\bar{i}) = T} \\
\frac{\text{class } C : (\bar{a} : _) \text{ extends } _ \{\dots, m : U, \dots\} \text{ is } \{_ \}}{\text{mtype}(m, C\bar{i}) = U[\bar{i}/\bar{a}]} \text{ (MT-CLASS)} \\
\frac{\text{class } C : (\bar{a} : _) \text{ extends } D\bar{j}\{\bar{l} : _ \} \text{ is } \{_ \} \quad m \notin \bar{l}}{\text{mtype}(m, C\bar{i}\bar{i}') = \text{mtype}(m, D\bar{j}[\bar{i}\bar{i}'/\bar{a}])[C\bar{i}/D]} \text{ (MT-SUPER)} \\
\boxed{\text{mbody}(m, C) = \lambda x.t} \\
\frac{\text{class } C : _ \text{ extends } _ \{_ \} \text{ is } \{\dots, \text{init}() = \bar{f} := \text{new } \bar{C}(), \dots\}}{\text{mbody}(\text{init}, C) = \bar{f} := \text{new } \bar{C}()} \text{ (MB-INIT)} \\
\frac{\text{class } C : _ \text{ extends } _ \{_ \} \text{ is } \{\dots, m(x) = t, \dots\} \quad m \neq \text{init}}{\text{mbody}(m, C) = \lambda x.t} \text{ (MB-CLASS)} \\
\frac{\text{class } C : _ \text{ extends } D\bar{j}\{_ \} \text{ is } \{\bar{M}\} \quad m \notin \bar{M}}{\text{mbody}(m, C) = \text{mbody}(m, D)} \text{ (MB-SUPER)} \\
\boxed{\text{q}(T) \text{ where } \text{q} ::= \text{un} \mid \text{lin}} \quad \text{un}(\text{Top}) \quad \frac{\text{not un}(T)}{\text{lin}(T)} \\
\frac{\text{classof}(T) = C \quad \text{class } C : _ \text{ extends } _ \{\bar{f} : _, \bar{m} : \Pi _. (\bar{T} \rightsquigarrow \bar{T} \times _ \rightarrow _) \} \text{ is } \{_ \}}{\text{un}(T)}
\end{array}$$

■ **Figure 7** Auxiliary functions and predicates.

The two rules S-+L and S-+R_k together imply that a type $T + U$ is a least upper bound of T and U . S- \times expresses that the subtyping relation is a congruence. S-RECORD checks compatibility between field typings of the same class C .

3.3.2 Typing

Auxiliary Functions and Predicates. As in Featherweight Java (FJ) [25], our typing rules rely on a few auxiliary functions and predicates. These are given in Figure 7 and described below. We denote by \sqcup the disjoint union of field types, i.e. the operation of $F_1 \sqcup F_2$ is defined by merging F_1 and F_2 if their domains are disjoint, being undefined otherwise. We write $m \notin \bar{l}$ and $m \notin \bar{M}$ to indicate that the method name m is not included, respectively, in the sequence of member names \bar{l} and method definitions \bar{M} . We denote by $T[C\bar{i}/D]$ the substitution of $C\bar{i}$ for the free occurrences of D bound to a type \rightsquigarrow in T .

The partial function $\text{classof}(T)$ looks up the class of a type T of the form $C\bar{i}$ and $\Sigma a : I.U$, being undefined for other forms. Both $\text{fields}(T)$ and $\text{mtype}(m, T)$, also partial functions, look up member types. Notice that a subclass may extend an instantiated superclass, which

13:16 Dependent Types for Class-based Mutable Objects

$$\boxed{\Delta_1; \Gamma \vdash r : T \dashv \Delta_2} \text{ Under initial contexts } \Delta_1; \Gamma, \text{ path } r \text{ has type } T, \text{ with final context } \Delta_2$$

$$\frac{\Delta \vdash \Gamma}{\Delta; \Gamma, r : T \vdash r : T \dashv \Delta} \text{ (T-REF)} \qquad \frac{\Delta; \Gamma \vdash r : C[F] \dashv \Delta}{\Delta; \Gamma \vdash r.f : F(f) \dashv \Delta} \text{ (T-FIELD)}$$

$$\frac{\Delta_1; \Gamma \vdash r : \Sigma a : I.T \dashv \Delta_2}{\Delta_1; \Gamma \vdash r : T \dashv \Delta_2, a : I} \text{ (T-UNPACK)}$$

$$\frac{\Delta; \Gamma \vdash r : C[F] \dashv \Delta \quad \Delta \vdash C[F] <: \text{fields}(C^i)}{\Delta; \Gamma \vdash r : C^i \dashv \Delta} \text{ (T-HIDE)}$$

■ **Figure 8** Typing rules for paths.

means that, because of substitutions, the types of fields and methods in the subclass may not be identical to those in the superclass. On the other hand, $\text{mbody}(m, C)$ is used only in the operational semantics. The predicate $\mathfrak{q}(T)$ assigns a qualifier \mathfrak{q} to a type T : a type is said to be unrestricted (**un**) if denotes an instance of a type invariant class, that is, a class whose methods do not change the state of the current object (the input and output types are the same); it is linear (**lin**) if its class defines at least one type varying method, which indicates that the state of the current object is modified.

Term Typing. For typing terms, we use a judgement of the form $\Delta_1; \Gamma_1 * r_1 \vdash t : T \dashv \Delta_2; \Gamma_2 * r_2$ meaning that the evaluation of term t may both extend the context Δ_1 (for example, with existential variables that arise from the types of fields, or with propositions) and change the types contained in Γ_1 (for example, by assigning values to objects, or by calling methods on them), giving rise to the final contexts $\Delta_2; \Gamma_2$. Linearity is yet another reason for a different final object context: if x is linear and is used in t , then x is consumed and does not appear in Γ_2 . The judgement includes r_1 and r_2 in the style of Gay et al. [21], which are paths needed for typing runtime terms and tracing objects in the heap. When typechecking a program, both r_1 and r_2 are always **this**, and are used exclusively to access the fields of the current class. Hence, the judgement for typing top-level terms will always have the form $\Delta_1; \Gamma_1, \text{this} : C[F_1] * \text{this} \vdash t : T \dashv \Delta_2; \Gamma_2, \text{this} : C[F_2] * \text{this}$, where Γ_1 and Γ_2 differ only in the method parameter x . If Γ_1 is $x : U$ and U is linear, then Γ_2 must be ϵ since x has been consumed by t .

The typing rules for the top-level terms (Figure 4) are given in Figure 9. They use a judgement $\Delta_1; \Gamma \vdash r : T \dashv \Delta_2$ for typing paths (Figure 8), and a definition $C.l_k$ which means T_k for class $C : \Delta$ extends $T\{l_1 : T_1, \dots, l_n : T_n\}$ is $\{\bar{M}\}$ with $1 \leq k \leq n$.

► **Definition 1** (Operations on Field Types and Object Contexts).

- If $F = \{f_1 : T_1, \dots, f_n : T_n\}$, then $F(f_k) \triangleq T_k$, and $F\{f_j \leftarrow U\} \triangleq \{f_1 : T'_1, \dots, f_n : T'_n\}$ where $T'_k = T_k$ and $T'_j = U$ for $k \neq j$ and $n \geq 1$ and $1 \leq k \leq n$ and $1 \leq j \leq n$.
- $(\Gamma, x : T)\{x \leftarrow U\} \triangleq \Gamma, x : U$.
- $\Gamma\{r.f \leftarrow T\} \triangleq \Gamma\{r \leftarrow C[F\{f \leftarrow T\}]\}$ if $\Delta; \Gamma \vdash r : C[F] \dashv \Delta$ for some Δ .

We now comment on these rules: T-UNVAR and T-LINVAR are used to access a parameter. The former is the standard rule for reading a variable, while the latter implements destructive reads. T-UNFIELD is used for field access, being defined for unrestricted types only (since the effect of reading f linear would remove it from the current object type). T-NEW is the rule for object creation, giving the new object the type from the init method signature. T-ASSIGN

$\Delta_1; \Gamma_1 * r_1 \vdash t : T \dashv \Delta_2; \Gamma_2 * r_2$

Under initial contexts $\Delta_1; \Gamma_1$ with path r_1 ,
term t has type T , with final contexts $\Delta_2; \Gamma_2$ and path r_2

$$\frac{\Delta \vdash \Gamma \quad \text{un}(T)}{\Delta; \Gamma, x : T * r \vdash x : T \dashv \Delta; \Gamma, x : T * r} \text{ (T-UNVAR)}$$

$$\frac{\Delta \vdash \Gamma \quad \text{lin}(T)}{\Delta; \Gamma, x : T * r \vdash x : T \dashv \Delta; \Gamma * r} \text{ (T-LINVAR)}$$

$$\frac{\Delta; \Gamma \vdash r.f : T \dashv \Delta \quad \text{un}(T)}{\Delta; \Gamma * r \vdash f : T \dashv \Delta; \Gamma * r} \text{ (T-UNFIELD)}$$

$$\frac{\Delta \vdash \Gamma}{\Delta; \Gamma * r \vdash \text{new } C() : C.\text{init} \dashv \Delta; \Gamma * r} \text{ (T-NEW)}$$

$$\frac{\Delta_1; \Gamma_1 * r_1 \vdash t : T \dashv \Delta_2; \Gamma_2 * r_2 \quad \Delta_2; \Gamma_2 \vdash r_2 : C[F] \dashv \Delta_2 \quad \Delta_2; \Gamma_2 \{r_2.f \leftarrow T\} \vdash r_2 : C\bar{i} \dashv \Delta_2}{\Delta_1; \Gamma_1 * r_1 \vdash f := t : F(f) \dashv \Delta_2; \Gamma_2 \{r_2.f \leftarrow T\} * r_2} \text{ (T-ASSIGN)}$$

$$\frac{\Delta_1; \Gamma_1 * r_1 \vdash t_1 : U \dashv \Delta_2; \Gamma_2 * r_2 \quad \Delta_2; \Gamma_2 * r_2 \vdash t_2 : T \dashv \Delta_3; \Gamma_3 * r_2 \quad \text{un}(U)}{\Delta_1; \Gamma_1 * r_1 \vdash t_1; t_2 : T \dashv \Delta_3; \Gamma_3 * r_2} \text{ (T-SEQ)}$$

$$\frac{\Delta_1; \Gamma_1 * r_1 \vdash t : U[\theta] \dashv \Delta_2; \Gamma_2 * r_2 \quad \Delta_2; \Gamma_2 \vdash r_2 : C\bar{i} \dashv \Delta_2 \quad \text{mtype}(m, C\bar{i}) = \Pi \Delta. (C\bar{i} \rightsquigarrow T \times U \rightarrow W) \quad \Delta_2 \vdash \Delta : \theta \quad \Delta_2; \Gamma_2 \{r_2 \leftarrow T[\theta]\} \vdash r_2 : C\bar{j} \dashv \Delta_3}{\Delta_1; \Gamma_1 * r_1 \vdash m(t) : W[\theta] \dashv \Delta_3; \Gamma_2 \{r_2 \leftarrow \text{fields}(C\bar{j})\} * r_2} \text{ (T-SELF CALL)}$$

$$\frac{\Delta_1; \Gamma_1, r_1 : C[F] * r_1 \vdash t : U[\theta] \dashv \Delta_2; \Gamma_2 * r_2 \quad \Delta_2; \Gamma_2 \vdash r_2.f : T_1 \dashv \Delta_3 \quad \text{mtype}(m, T_1) = \Pi \Delta. (T_1 \rightsquigarrow T_2 \times U \rightarrow W) \quad \Delta_3 \vdash \Delta : \theta \quad \Delta_3; \Gamma_2 \{r_2.f \leftarrow T_2[\theta]\} \vdash r_2 : C\bar{i} \dashv \Delta_3}{\Delta_1; \Gamma_1, r_1 : C[F] * r_1 \vdash f.m(t) : W[\theta] \dashv \Delta_3; \Gamma_2 \{r_2.f \leftarrow T_2[\theta]\} * r_2} \text{ (T-CALL)}$$

$$\frac{\Delta_1; \Gamma_1 \vdash r.f : (U_1 + U_2) \dashv \Delta_2 \quad \text{classof}(U_k) = C_k \quad \Delta_2; \Gamma_1 \{r.f \leftarrow U_k\} * r \vdash t_k : T \dashv \Delta_3; \Gamma_2 * r \quad C_1 \neq C_2}{\Delta_1; \Gamma_1 * r \vdash \text{case } f \text{ of } (C_k \Rightarrow t_k)_{k \in 1,2} : T \dashv \Delta_3; \Gamma_2 * r} \text{ (T-CASE)}$$

$$\frac{\Delta_1; \Gamma_1 * r_1 \vdash t : \text{Boolean } p \dashv \Delta_2; \Gamma_2 * r_2 \quad \Delta_2, p; \Gamma_2 * r_2 \vdash t_1 : T \dashv \Delta_3; \Gamma_3 * r_2 \quad \Delta_2, \neg p; \Gamma_2 * r_2 \vdash t_2 : T \dashv \Delta_3; \Gamma_3 * r_2}{\Delta_1; \Gamma_1 * r_1 \vdash \text{if } t \text{ then } t_1 \text{ else } t_2 : T \dashv \Delta_3; \Gamma_3 * r_2} \text{ (T-IF)}$$

$$\frac{\Delta_1; \Gamma_1 * r_1 \vdash t_1 : \text{Boolean } p \dashv \Delta_2; \Gamma_2 * r_2 \quad \Delta_2, p; \Gamma_2 * r_2 \vdash t_2 : \text{Top} \dashv \Delta_2; \Gamma_2 * r_2}{\Delta_1; \Gamma_1 * r_1 \vdash \text{while } t_1 \text{ do } t_2 : \text{Top} \dashv \Delta_2, \neg p; \Gamma_2 * r_2} \text{ (T-WHILE)}$$

$$\frac{\Delta_1; \Gamma_1 * r_1 \vdash t : U \dashv \Delta_2; \Gamma_2 * r_2 \quad \Delta_2 \vdash U <: T}{\Delta_1; \Gamma_1 * r_1 \vdash t : T \dashv \Delta_2; \Gamma_2 * r_2} \text{ (T-SUB)}$$

■ **Figure 9** Typing rules for terms in the top-level language.

modifies a field of the current object, acting on its type $C[F]$. Unlike the rule for assignment in Java, when a field is changed, we need to check all the other fields in F to ensure that any dependencies are satisfied. We do this with judgement $\Delta_2; \Gamma_2\{r_2.f \leftarrow T\} \vdash r_2 : C\bar{i} \dashv \Delta_2$ derived by rule T-HIDE that recovers a top-level type $C\bar{i}$ using with the updated context as its initial context. Again, unlike the standard rule for assignment, our rule returns as its result the type of the old object contained in the field as part of the linear control of objects. T-SEQ is the standard rule for the sequence operation, except that it checks the first subterm and considers its possible effects in the typing context that checks the second one.

The two rules for calling methods are rather elaborate. T-SELFCALL checks the type of the parameter as usual, but uses rule T-HIDE to obtain a top-level type for the current object r of the form $C\bar{i}$ that allows method m to be called (its signature yielding a substitution θ applied to the parameter and output types). The final object context is updated in the conclusion with a type obtained by $\text{fields}(C\bar{j})$ for r , where $C\bar{j}$ is derived by possibly unpacking the receiver output type $T[\theta]$ in the premise $\Delta_2; \Gamma_2\{r_2 \leftarrow T[\theta]\} \vdash r_2 : C\bar{j} \dashv \Delta_3$. T-CALL checks a method call on a field, combining the strategies used in T-ASSIGN and T-SELFCALL.

T-CASE makes the case distinction on a field f with a union type. Each branch is then typed with an initial context where f is bound to either the left or the right type. Two branches must have the same type and final contexts, because f can only be bound to one type. T-IF expects t in the condition to be of type Boolean p . Each branch is then typed with initial contexts asserting or negating the proposition p . T-WHILE is analogous to T-IF, yet simpler. T-SUB is the usual subsumption rule, adapted to our requirements.

Program Typing. A well-formed program relies on well-typed fields, methods and classes, which we formally define in Figure 10. The judgement $\vdash_C M$ states that a method M in a class C is well-typed. T-METHOD constructs the judgement for checking the body of a regular method, whereas T-INIT initialises all fields, including the inherited ones. The judgement $\Delta \vdash_T l : T$ checks that a member type is well-formed. T-FTYPE states that a field must be “typed” by kind \star of proper types. When checking method signatures, one of the following must hold: the method is altogether new (T-MTYPE), or a correct override of a superclass method (T-OVERRIDE). These judgements are used by T-CLASS. By T-PROGRAM a program is well-formed if each class defined in it is well-typed.

3.4 Operational Semantics

Figure 11 defines an operational semantics on states S the form $(h * r, t)$ where the object path r is used to resolve field references appearing in the term t . As usual, we denote by $t[o/x]$ the substitution of o for the free occurrences of x in t defined in the standard way.

► **Definition 2 (Operations on Heaps).** Let h be the heap of the form $h = (h_0, o = R)$ where R is the object record $C\{f_1 = o_1, \dots, f_n = o_n\}$. Then, $h(o) \triangleq R$ and $h(o).\text{class} = C$ and for all k such that $1 \leq k \leq n$,

- $R.f_k \triangleq o_k$.
- $R\{f_j \leftarrow o\} \triangleq C\{\bar{f} = \bar{o}'\}$ where $o'_k = o_k$ and $o'_j = o$ for $k \neq j$ and $1 \leq j \leq n$.
- $h\{o.f_k \leftarrow o'\} \triangleq (h_0, o = R\{f_k \leftarrow o'\})$.
- $h(r) \triangleq o_k$ if $r = o.f_k$, and $h\{r.f \leftarrow o'\} \triangleq h\{o_k.f \leftarrow o'\}$.

R-NEW creates a fresh object and adds it to the heap, after having initialised all fields. For this, it relies on the reduction of states for sequenced object creation. R-ASSIGN replaces the value of a field f of the current object located at r with a new reference, and returns

$\boxed{\vdash_C M}$ Method M is well-formed in class C

$$\begin{array}{c} \text{class } C : \Delta_1 \text{ extends } _ \{ \dots, m : \Pi \Delta_2. (T_1 \rightsquigarrow T_2 \times U \rightarrow W), \dots \} \text{ is } \{ _ \} \\ \Delta_1, \Delta_2; x : U, \text{this} : \text{fields}(T_1) * \text{this} \vdash t : W \dashv \Delta_3; \Gamma, \text{this} : C[F] * \text{this} \\ \frac{x : U \in \Gamma \Rightarrow \text{un}(U) \quad \Delta_3 \vdash C[F] <: \text{fields}(T_2) \quad m \neq \text{init}}{\vdash_C m(x) = t} \text{ (T-METHOD)} \\ \frac{\text{fields}(C.\text{init}) = C[F] \quad \epsilon; \epsilon * r \vdash \text{new } \bar{C}() : F(\bar{f}) \dashv \epsilon; \epsilon * r \quad \text{no cycles in } C}{\vdash_C \text{init}() = \bar{f} := \text{new } \bar{C}()} \text{ (T-INIT)} \end{array}$$

$\boxed{\Delta \vdash_T l : U}$ Member l has type U with supertype T

$$\begin{array}{c} \frac{\Delta \vdash U : \star}{\Delta \vdash_T f : U} \text{ (T-FTOTYPE)} \quad \frac{\text{mtype}(m, T) \text{ undefined} \quad \Delta_1 \vdash \Pi \Delta_2. (C_i \times U \times W) : \star \quad \Delta_1, \Delta_2 \vdash C_j <: T_2}{\Delta_1 \vdash_T m : \Pi \Delta_2. (C_i \rightsquigarrow T_2 \times U \rightarrow W)} \text{ (T-MTOTYPE)} \\ \frac{\text{mtype}(m, T) = \Pi \Delta'_2. (T'_1 \rightsquigarrow T'_2 \times U' \rightarrow W') \quad \Delta_1 \vdash \Pi \Delta_2. (T_1 \times T_2 \times U' \times W) <: \Pi \Delta'_2. (T'_1 \times T'_2 \times U \times W')}{\Delta_1 \vdash_T m : \Pi \Delta_2. (T_1 \rightsquigarrow T_2 \times U \rightarrow W)} \text{ (T-OVERRIDE)} \end{array}$$

$\boxed{\vdash L}$ Class declaration L is well-formed

$$\frac{\Delta \vdash T : \star \quad \Delta \vdash_T \bar{l} : \bar{T} \quad \vdash_C \bar{M}}{\vdash \text{class } C : \Delta \text{ extends } T \{ \bar{l} : \bar{T} \} \text{ is } \{ \bar{M} \}} \text{ (T-CLASS)}$$

$\boxed{\vdash P}$ Program P is well-formed

$$\frac{\vdash L_1 \quad \dots \quad \vdash L_n}{\vdash L_1 \dots L_n} \text{ (T-PROGRAM)}$$

■ **Figure 10** Typing rules for program formation.

the former object pointed by f . R-SEQ reduces to the second part of the sequence of terms, discarding the first part only after it has become an object.

R-SELF CALL is relative to a method call on the current object at r . The rule prepares the method body t with a substitution (the actual parameter for the formal one) before evaluating the term. R-CALL is the rule for a call on the object at f (relative to the current object at r), being defined in a slightly different way. The rule makes $r.f$ become the current object and wraps the method body t , prepared with the parameter substitution, in a return term that replaces the method call. Then, the body is reduced to an object in rule R-RETURN which also recovers the previous current object at r .

R-CASE $_k$ means that either branch is taken, with the first having precedence over the second, i.e. the second branch is only tried if the condition $(h(r.f).\text{class} = C_1)$ fails. The two rules R-IFTRUE and R-IFFALSE use the special true and false objects for the references that control the condition. In rule R-WHILE, the term is rewritten to a nested conditional, using top for the body of the else branch. R-CONTEXT is standard for reduction in contexts, defining which term should be evaluated next.

$$\boxed{S_1 \longrightarrow S_2} \text{ State } S_1 \text{ reduces to } S_2$$

$$\begin{array}{c}
 \text{mbody}(\text{init}, C) = \bar{f} := \text{new } \bar{C}() \\
 \frac{(h_1 * r, \text{new } \bar{C}()) \longrightarrow (h_2 * r, \bar{o}) \quad o \notin \text{dom}(h_2)}{(h_1 * r, \text{new } C()) \longrightarrow ((h_2, o = C\{\bar{f} = \bar{o}\} * r), o)} \text{ (R-NEW)} \\
 \\
 \frac{h(r).f = o_1}{(h * r, f := o_2) \longrightarrow (h\{r.f \leftarrow o_2\} * r, o_1)} \text{ (R-ASSIGN)} \\
 \\
 (h * r, o; t) \longrightarrow (h * r, t) \text{ (R-SEQ)} \quad \frac{h(r).\text{class} = C \quad \text{mbody}(m, C) = \lambda x.t}{(h * r, m(o)) \longrightarrow (h * r, t[o/x])} \text{ (R-SELF CALL)} \\
 \\
 \frac{h(r.f).\text{class} = C \quad \text{mbody}(m, C) = \lambda x.t}{(h * r, f.m(o)) \longrightarrow (h * r.f, \text{return } t[o/x])} \text{ (R-CALL)} \\
 \\
 (h * r.f, \text{return } o) \longrightarrow (h * r, o) \text{ (R-RETURN)} \\
 \\
 \frac{h(r.f).\text{class} = C_k}{(h * r, \text{case } f \text{ of } (C_k \Rightarrow t_k)_{k \in 1,2}) \longrightarrow (h * r, t_k)} \text{ (R-CASE}_k\text{)} \\
 \\
 (h * r, \text{if true then } t_1 \text{ else } t_2) \longrightarrow (h * r, t_1) \text{ (R-IF TRUE)} \\
 \\
 (h * r, \text{if false then } t_1 \text{ else } t_2) \longrightarrow (h * r, t_2) \text{ (R-IF FALSE)} \\
 \\
 \frac{t_2 = \text{if } t \text{ then } (t_1; \text{while } t \text{ do } t_1) \text{ else top}}{(h * r, \text{while } t \text{ do } t_1) \longrightarrow (h * r, t_2)} \text{ (R-WHILE)} \\
 \\
 \frac{(h_1 * r_1, t_1) \longrightarrow (h_2 * r_2, t_2)}{(h_1 * r_1, \mathcal{E}[t_1]) \longrightarrow (h_2 * r_2, \mathcal{E}[t_2])} \text{ (R-CONTEXT)}
 \end{array}$$

■ **Figure 11** Reduction rules for states.

4 Type Soundness

In order to establish type soundness, we need an additional set of relations that describe heaps and runtime states. This is given in Figure 12. For typing the heap, we use a judgement of the form $\Delta; \Gamma \vdash h$ that states that under contexts $\Delta; \Gamma$ the heap h is well-formed. By rule T-EMPTYHEAP, a heap is constructed from typing contexts containing assumptions and types for all the objects relative to locations added to the heap by rule T-HEAP. The latter ensures that each heap entry has the prescribed field typing. The most important feature of this rule is that all aliases of linear references are explicitly forbidden by the rightmost premise. For typing sequenced objects \bar{o} as part of a runtime state, the judgement relies on T-UNVAR and T-LINVAR as appropriate to type each object. In particular, for each linear o_k in o_1, \dots, o_n , with $1 \leq k \leq n$, the initial typing context contains o_k and the final one of the extended heap does not, meaning that a heap that contains multiple references to the same linear object is not typable. The similar inverse argument justifies the existence of cyclic structures in the heap. Rule T-HEAPHIDE is used as needed in order to replace an internal object type by an equivalent top-level one (cf. [21]).

Finally, we use a judgement $\Delta_1; \Gamma_1 \vdash S : T \dashv \Delta_2; \Gamma_2 * r$ to type states and formalize the main invariant of subject reduction. By T-STATE, given a state S of the form $(h * r_1, t)$, the heap h must be compatible with a context Γ_1 under the assumptions in Δ_1 , which are the initial contexts that type the runtime term t , knowing from the leftmost premises that h is

$\boxed{\Delta; \Gamma \vdash h}$ Under contexts $\Delta; \Gamma$, heap h is well-formed

$$\frac{\Delta \vdash \Gamma}{\Delta; \Gamma \vdash \epsilon} \text{ (T-EMPTYHEAP)}$$

$$\frac{\Delta; \Gamma_1 \vdash h \quad \Delta; \Gamma_1 * o \vdash \bar{o} : F(\bar{f}) \dashv \Delta; \Gamma_2, o : C[F] * o}{\Delta; \Gamma_2, o : C[F] \vdash h, (o = C\{\bar{f} = \bar{o}\})} \text{ (T-HEAP)}$$

$$\frac{\Delta; \Gamma, o : C[F] \vdash h \quad \Delta; \Gamma, o : C[F] \vdash o : C\bar{i} \dashv \Delta}{\Delta; \Gamma, o : C\bar{i} \vdash h} \text{ (T-HEAPHIDE)}$$

$\boxed{\Delta_1; \Gamma_1 \vdash S : T \dashv \Delta_2; \Gamma_2 * r}$ Under initial contexts $\Delta_1; \Gamma_1$, state S has type T , with final contexts $\Delta_2; \Gamma_2$ and path r

$$\frac{\text{dom}(\Gamma_1) \subseteq \text{dom}(h) \quad \Delta_1; \Gamma_1 \vdash h \quad \Delta_1; \Gamma_1 * r_1 \vdash t : T \dashv \Delta_2; \Gamma_2 * r_2}{\Delta_1; \Gamma_1 \vdash (h * r_1, t) : T \dashv \Delta_2; \Gamma_2 * r_2} \text{ (T-STATE)}$$

■ **Figure 12** Typing rules for heaps and states.

complete, i.e. for any $o \in \text{dom}(h)$ we have $\text{children}_h(o) \subseteq \text{dom}(h)$, and that $\text{dom}(\Gamma_1) \subseteq \text{dom}(h)$, i.e. every object that has a type in Γ_1 appears in h along with all of its children.

► **Definition 3** (Initial Heap and Object Context). In any well-formed program ($\vdash P$), h_0 and Γ_0 represent the initial heap and object context such that $h_0 = (\text{top} = \text{Top}\{\}, \text{false} = \text{Boolean}\{\}, \text{true} = \text{Boolean}\{\})$ and $\Gamma_0 = (\text{top} : \text{Top}, \text{false} : \text{Boolean false}, \text{true} : \text{Boolean true})$.

Main Results. By standard techniques [46], we now prove the expected results.

► **Theorem 4** (Subject Reduction). *Suppose that P is a well-formed program ($\vdash P$). In this context, let $\Gamma_0 \subseteq \Gamma_1$ and $h_0 \subseteq h_1$, and $S_1 = (h_1 * r_1, t)$. If $\Delta_1; \Gamma_1 \vdash S_1 : T \dashv \Delta_2; \Gamma_2 * r_2$ and $S_1 \longrightarrow S_2$, then $\Delta'_1; \Gamma'_1 \vdash S_2 : T \dashv \Delta_2; \Gamma'_2 * r_2$ for some Δ'_1, Γ'_1 and Γ'_2 such that $\Delta_1 \subseteq \Delta'_1$ and $\Gamma_2 \subseteq \Gamma'_2$.*

Proof sketch. In order to build this result, we need to prove a number of basic lemmas, namely inversion of the term typing relation, exchange for object contexts, weakening for index contexts, substitution for objects in term typing, substitution for indices, substitution for class types, and agreement of judgements. We also prove soundness and instantiation of function `mtype` as well as two lemmas (opening and closing) for the replacement of an internal object type by an equivalent top-level one when typing the heap. We then show that in well-formed DOL programs the types of objects describe their runtime values, that there never exists more than one reference to a linear object, and that all aliasing never produce a value of unexpected type. ◀

► **Theorem 5** (Progress). *Suppose that P is a well-formed program ($\vdash P$). In this context, let $\Gamma_0 \subseteq \Gamma_1$ and $h_0 \subseteq h_1$.*

1. *If $\Delta_1; \Gamma_1 \vdash (h_1 * r_1, t_1) : T \dashv \Delta_2; \Gamma_2 * r_2$, then t_1 is an object reference or $(h_1 * r_1, t_1) \longrightarrow (h_2 * r_2, t_2)$.*
2. *If $\Delta_1; \Gamma_1 \vdash (h_1 * r, \bar{t}) : \bar{T} \dashv \Delta_2; \Gamma_2 * r$, then $(h_1 * r, \bar{t}) \longrightarrow (h_2 * r, \bar{t}')$.*

Proof sketch. By mutual induction on the structure of t and the length of \bar{t} . ◀

$$\boxed{\Delta_1 \vdash T <: U \dashv \Delta_2} \quad \text{Under initial context } \Delta_1, \text{ type } T \text{ is a subtype of } U, \text{ with final context } \Delta_2$$

$$\frac{\text{class } C : (\bar{a} : \bar{I}) \text{ extends } T\{_ \} \text{ is } \{_ \} \quad \Delta_1 \vdash T[\bar{i}/\bar{a}] <: D\bar{j} \dashv \Delta_2 \quad C \neq D}{\Delta_1 \vdash C\bar{i} <: D\bar{j} \dashv \Delta_2} \quad (\text{AS-SUPER})$$

$$\frac{\Delta \triangleright T^\circ : \star}{\Delta \vdash T^\circ <: \text{Top} \dashv \Delta} \quad (\text{AS-TOP}) \qquad \frac{\Delta_1 \vdash \bar{i} \equiv \bar{j} \dashv \Delta_2}{\Delta_1 \vdash C\bar{i} <: C\bar{j} \dashv \Delta_2} \quad (\text{AS-APP})$$

$$\frac{\Delta_1, \hat{a} : I \vdash U[\hat{a}/a] <: T^\circ \dashv \Delta_2}{\Delta_1 \vdash \Pi a : I.U <: T^\circ \dashv \Delta_2} \quad (\text{AS-III}) \qquad \frac{\Delta_1, a : I \vdash T <: U \dashv \Delta_2}{\Delta_1 \vdash T <: \Pi a : I.U \dashv \Delta_2} \quad (\text{AS-IIR})$$

$$\frac{\Delta_1, a : I \vdash T <: U \dashv \Delta_2}{\Delta_1 \vdash \Sigma a : I.T <: U \dashv \Delta_2} \quad (\text{AS-SL}) \qquad \frac{\Delta_1, \hat{a} : I \vdash T^\circ <: U[\hat{a}/a] \dashv \Delta_2}{\Delta_1 \vdash T^\circ <: \Sigma a : I.U \dashv \Delta_2} \quad (\text{AS-SR})$$

$$\boxed{\Delta_1; \Gamma_1 \vdash t \uparrow T \dashv \Delta_2; \Gamma_2} \quad \text{Under initial contexts } \Delta_1; \Gamma_1, \text{ term } t \text{ synthesizes type } T, \text{ with final contexts } \Delta_2; \Gamma_2$$

$$\boxed{\Delta_1; \Gamma_1 \vdash t \downarrow T \dashv \Delta_2; \Gamma_2} \quad \text{Under initial contexts } \Delta_1; \Gamma_1, \text{ term } t \text{ checks against input type } T, \text{ with final contexts } \Delta_2; \Gamma_2$$

$$\frac{\Delta_1; \Gamma_1 \vdash \text{this}.f \uparrow T_1 \dashv \Delta_2 \quad \text{mtype}(m, T_1) = \Pi(\bar{a} : \bar{I}).(T_1 \rightsquigarrow T_2 \times U \rightarrow W) \quad \Delta_2, \bar{a} : \bar{I}; \Gamma_1 \vdash t \downarrow U[\bar{a}/\bar{a}] \dashv \Delta_3; \Gamma_2 \quad \bar{a} \text{ fresh} \quad \Delta_3; \Gamma_2\{\text{this}.f \leftarrow T_2[\bar{a}/\bar{a}]\} \vdash \text{this} \uparrow_h C\bar{i} \dashv \Delta_4}{\Delta_1; \Gamma_1 \vdash f.m(t) \uparrow W[\bar{a}/\bar{a}] \dashv \Delta_3; \Gamma_2\{\text{this}.f \leftarrow T_2[\bar{a}/\bar{a}]\}} \quad (\text{AT-CALL})$$

$$\frac{\Delta_1; \Gamma_1 \vdash \text{this}.f \uparrow (U_1 + U_2) \dashv \Delta_2 \quad \text{classof}(U_k) = C_k \quad \Delta_2; \Gamma_1\{\text{this}.f \leftarrow U_k\} \vdash t_k \uparrow T_k \dashv \Delta_{k+2}; \Gamma_{k+2} \quad C_1 \neq C_2}{\Delta_1; \Gamma_1 \vdash \text{case } f \text{ of } (C_k \Rightarrow t_k)_{k \in 1,2} \uparrow (T_1 + T_2) \dashv (\Delta_3; \Gamma_3 \cdot \Delta_4; \Gamma_4)} \quad (\text{AT-CASE})$$

■ **Figure 13** Selected algorithmic rules. In the subtyping rules, T° means that T is not a type Π, Σ , or $+$. In rules AS-III and AS-SR, index variable \hat{a} is fresh.

5 Algorithmic Typechecking

We develop an algorithmic system in two steps. The first step is to introduce an existential index variable (written \hat{a} with the hat in the style of Dunfield and Krishnaswami [14, 15, 16]) into the initial index context whenever there is a need to make a guess at the appropriate index term i . The oracular rules in the declarative system are S-III and S-SR, T-HIDE, T-SELF-CALL, T-CALL and T-MTYPE. In the algorithmic type system, each declarative judgement has a corresponding algorithmic judgement that takes an initial index context and yields a final index context, possibly augmented with knowledge about what index terms have to be. Instead of guessing, the algorithmic system adds judgements to instantiate existential index variables of the form $\Delta_1 \vdash \hat{a} := i \dashv \Delta_2$, and to equate index terms, namely $\Delta_1 \vdash i \equiv j \dashv \Delta_2$. The second step is to apply bidirectional typechecking [39] in order to distinguish rules that synthesize types from those that check terms against types already known, a technique that easily supports subtyping and index refinements. In the process, we also eliminate the nondeterminism associated with the subtyping and typing rules for paths.

The system uses the syntax and meta-variables of the declarative system (Figures 4 and 5). In addition to *solved* existential index variable declarations $a : I$, index contexts in the algorithmic system may also contain *unsolved* existential index variable declarations $\hat{a} : I$. Similarly to index contexts in the declarative system, index contexts in the algorithmic system are ordered sequences.

We give some of the algorithmic rules in Figure 13. The algorithmic subtyping rules use judgements of the form $\Delta_1 \vdash T <: U \dashv \Delta_2$ where the final index context Δ_2 may carry information about solved existential index variables. In the bidirectional typechecking algorithm [39], we alternate between synthesizing types and checking terms against types already known. Bidirectional typechecking in DOL is formalised by replacing the typing judgement of the form $\Delta_1; \Gamma_1 * r_1 \vdash t : T \dashv \Delta_2; \Gamma_2 * r_2$ with the following two judgements: $\Delta_1; \Gamma_1 \vdash t \uparrow T \dashv \Delta_2; \Gamma_2$ for synthesizing and $\Delta_1; \Gamma_1 \vdash t \downarrow T \dashv \Delta_2; \Gamma_2$ for checking.

► **Definition 6** (Complete Index Contexts). A complete index context, denoted by ϕ , is an algorithmic index context such that for every existential index variable \hat{a} in $\text{dom}(\phi)$, $\phi(\hat{a}) \triangleq \hat{a} : I \doteq i$.

Soundness. To show that the algorithmic system is sound with respect to the original system, we are given an algorithmic judgement, with an initial index context Δ_1 and a final index context Δ_2 , and ϕ as a solved extension of context Δ_2 , and hence $\text{dom}(\Delta_1) \subseteq \text{dom}(\phi)$ (by transitivity). Applying ϕ as a substitution to the given algorithmic judgement produces a declarative judgement, which is the result we want to obtain.

► **Theorem 7** (Soundness of Algorithmic Subtyping). *If $\Delta_1 \triangleright T : \star$ and $\Delta_1 \triangleright U : \star$ and $T[\Delta_1] = T$ and $U[\Delta_1] = U$ and $\Delta_1 \vdash T <: U \dashv \Delta_2$ and ϕ extends Δ_2 , then $\Delta_2[\phi] \vdash T[\phi] <: U[\phi]$.*

► **Theorem 8** (Soundness of Algorithmic Typing). *Let ϕ be a complete index context that extends Δ_2 .*

1. *If $\Delta_1; \Gamma \vdash r \uparrow T \dashv \Delta_2$, then $\Delta_1[\phi]; \Gamma[\phi] \vdash r : T[\phi] \dashv \Delta_2[\phi]$.*
2. *If $\Delta_1; \Gamma_1 \vdash t \uparrow T \dashv \Delta_2; \Gamma_2$, then $\Delta_1[\phi]; \Gamma_1[\phi] * \text{this} \vdash t : T[\phi] \dashv \Delta_2[\phi]; \Gamma_2[\phi] * \text{this}$.*
3. *If $\Delta_1; \Gamma_1 \vdash t \downarrow T \dashv \Delta_2; \Gamma_2$ and $\Delta_1 \triangleright T : \star$, then $\Delta_1[\phi]; \Gamma_1[\phi] * \text{this} \vdash t : T[\phi] \dashv \Delta_2[\phi]; \Gamma_2[\phi] * \text{this}$.*

Completeness. To prove completeness of the algorithmic system, we somehow do the reverse of soundness: from a declarative derivation, which has no existential index variables, we obtain a complete index context along an algorithmic derivation. In completeness of algorithmic *subtyping*, we are given an initial index context Δ_1 and a complete index context ϕ_1 that extends it. In completeness of algorithmic *typing*, in addition we are given a final context Δ'_1 that may extend Δ_1 (with the result of unpacking or with propositions, for example) such that $\text{dom}(\Delta_1) \subseteq \text{dom}(\Delta'_1)$ and $\text{dom}(\Delta'_1) = \text{dom}(\phi_1)$, and hence $\text{dom}(\Delta_1) \subseteq \text{dom}(\phi_1)$. We show that we can build an algorithmic derivation with a final context Δ_2 . However, the algorithmic rules generate fresh index variables that may not be in Δ_1, Δ'_1 or ϕ_1 . So, completeness will also produce a complete index context ϕ_2 that extends both Δ_2 and ϕ_1 such that $\text{dom}(\Delta_2) = \text{dom}(\phi_2)$.

► **Theorem 9** (Completeness of Algorithmic Subtyping). *Let ϕ_1 be a complete index context that extends Δ_1 such that $\text{dom}(\Delta_1) = \text{dom}(\phi_1)$. If $\Delta_1[\phi_1] \vdash T[\phi_1] : \star$ and $\Delta_1[\phi_1] \vdash U[\phi_1] : \star$ and $\Delta_1[\phi_1] \vdash T[\phi_1] <: U[\phi_1]$, then $\Delta_1 \vdash T[\Delta_1] <: U[\Delta_1] \dashv \Delta_2$ and there exists ϕ_2 that extends both Δ_2 and ϕ_1 such that $\text{dom}(\Delta_2) = \text{dom}(\phi_2)$.*

► **Theorem 10** (Completeness of Algorithmic Typing). *Let ϕ_1 and Δ'_1 be index contexts that extend Δ_1 such that $\text{dom}(\Delta_1) \subseteq \text{dom}(\Delta'_1)$ and $\text{dom}(\Delta'_1) = \text{dom}(\phi_1)$.*

1. *If $\Delta_1[\phi_1] \vdash T[\phi_1] : \star$, $\Delta_1[\phi_1]; \Gamma[\phi_1] \vdash r : T[\phi_1] \dashv \Delta'_1[\phi_1]$, then $\Delta_1; \Gamma[\Delta_1] \vdash r : T[\Delta_1] \dashv \Delta_2$ and there exists ϕ_2 that extends both Δ_2 and ϕ_1 such that $\text{dom}(\Delta_2) = \text{dom}(\phi_2)$.*

2. If $\Delta_1[\phi_1] \vdash T[\phi_1] : \star$ and $\Delta_1[\phi_1]; \Gamma_1[\phi_1] * r_1 \vdash t : T[\phi_1] \dashv \Delta'_1[\phi_1]; \Gamma_2[\phi_1] * r_2$, then depending on t either $\Delta_1; \Gamma_1[\Delta_1] \vdash t \uparrow T[\Delta_1] \dashv \Delta_2; \Gamma_2[\Delta_1]$ or $\Delta_1; \Gamma_1[\Delta_1] \vdash t \downarrow T[\Delta_1] \dashv \Delta_2; \Gamma_2[\Delta_1]$ and there exists ϕ_2 that extends both Δ_2 and ϕ_1 such that $\text{dom}(\Delta_2) = \text{dom}(\phi_2)$.

Implementation. Our prototype ships with an IDE developed as an Eclipse plugin based on the Xtext framework, where the examples can be typechecked, compiled and run. The IDE support includes: a code editor assistant for DOL programs, on-the-fly error checking, and target code generation in the form of Java classes. Our typechecker is a direct implementation of the algorithmic type system, extended with integer and boolean literals, local variables and all the syntactic sugar from the examples. Constraint checking is performed as part of typechecking via a direct interface to the Z3 constraint solver [12].

6 Related Work and Discussion

At the basis of index refinements lies the notion of dependent type developed by Martin-Löf [32], and first applied to proof assistants (logical frameworks) such as AUTOMATH [44], the Calculus of Constructions [11], NuPRL [10], Lego [54] and the Edinburgh Logical Framework [22]. While full dependent types are an appealing feature to integrate in programming languages, the price is increased complexity of typechecking. Unlike index refinements, full dependent types do not restrict the domain of variables appearing in types. When added to (possibly nonterminating) programming languages, the task of determining type equivalence becomes as difficult as determining term equivalence (which is undecidable in general).

Some programming languages offer different strategies to handle nonterminating programs. Cayenne [2] is a functional programming language in the style of Haskell with an undecidable dependent type system. A semi-decidable approach forces the typechecker to terminate within a number of prescribed steps, eventually providing the user with an answer. Epigram [33] builds on a tactic-driven proof engine, similar to that of the Coq proof assistant, requiring correctness proofs to be specified. Unlike Cayenne, Epigram rules out general recursive programs, avoiding nontermination and any form of effects, thus making typechecking decidable. Recursion is supported by the structure of dependent types which are inductive families with inductive indices.

The Ynot tool is an extension of the functional dependently-typed language included in Coq with support for side-effects via Hoare Type Theory (HTT) and Separation Logic [35, 36]. HTT introduces an indexed monadic type in the style of a Hoare triple to reason about mutation. While DOL's varying types may have similarities with the Hoare type, our approach does not involve the complexity of higher-order abstraction. As HTT, the F* language [43], designed for program verification, employs the monad technique generalising it to multiple monads. This ML-style functional language uses dependent and refinement types to specify effectful programs, and supports automated and interactive proofs. A related approach is provided by RSP1 [45] that allows programming with proofs in an imperative setting. The language offers decidable typechecking by banning impure operations from types with the purpose of letting the user prove arbitrary properties of programs. All these languages provide SMT-based automation and handle effectful programming. In that regard, they are close to DOL, yet they differ substantially in their aim to combine programming and theorem proving, which our language does not support. Targeting the C programming language, Deputy [9] also handles mutation using a Hoare-inspired typing rule ensuring that assignment results in a well-typed state. For decidability, Deputy combines compile time and runtime checking, as opposed to our approach in which typechecking is performed statically.

Index refinements as formulated by Xi and Pfenning [53] reduce typechecking to a constraint satisfaction problem on terms belonging to index sorts. Their approach (which we adopt) offers the additional advantage of relative simplicity of the type system, as well as requiring fewer annotations, when compared to full dependent type systems. Xi later formulated Xanadu [48], a language with a C-like syntax combining imperative programming with index refinements, and ATS [50] which also supports DML-style dependent types and linear types (named viewpoints). While closely related, DOL extends the ideas of Xanadu to class-based objects that exhibit state and behaviour. Our language also handles object-oriented programming features such as modular development, inheritance with subtyping, which Xanadu does not deal with. A proposal for building an object-oriented system on top of DML was formulated by Xi [49]. The language includes inheritance without subtyping, simulated via existentially quantified dependent types. Xi's object model is simpler than ours, since objects are not regarded as records of fields (they merely respond to messages), and the language does not include imperative features. Ω mega [41] and Liquid Types [40] offer two more examples of functional languages with a strict phase separation; the latter is implemented in DSolve, a tool that automatically infers dependent types from an OCaml program and a set of logical qualifiers. Cyclone [26] is a type-safe extension of the C programming language, combining static analysis and runtime checks. It offers domain-specific indexed types for the purpose of safe multi-threading and memory management.

Another reference is Dependent JavaScript (DJS) [7], which introduces refinement types with predicates from an SMT-decidable logic in a dynamic real-world language. In DJS, imperative updates involve the presence of mutation: the types of variables are changed by assignment, for instance. The challenge is handled using *flow-sensitive heap types*, which allow tracking variable types, in combination with refinement types. The result is an increase in the language expressiveness by using type annotations inside JavaScript comments that account for side-effects. DJS employs the *alias types* approach [42] for strong updates in combination with thawing/freezing locations, an alternative to DOL's linear approach.

Other forms of dependent types include X10's constrained types [38], designed around the notion of constraints on the *immutable* state of objects. The core language proposed extends the purely functional FJ [25]. While appealing, constrained types currently cannot enforce invariants on the mutable state of objects. Dependent classes [20] provide another approach in the object-oriented setting. A class can be seen as forming a family of collaborating objects, much like a type family in traditional dependent type theory. The model is complex, since it also involves inheritance, and type soundness is hard to prove. Like DOL, full dependent classes and its lightweight version [27] support class-based programming and inheritance. A similar model is provided by Scala's path-dependent types [1] that unify nominal and structural type systems by allowing objects to contain type members. Dependent types in this model are expressed not in type signatures but in type placements. An abstract type refers to a type that must be defined by subclasses, becoming dependent on the instance it refers to. None of these languages supports an imperative style of programming, whereas DOL is designed to handle both mutable and immutable objects.

A kind of *typestate* seems to arise from having state exposed in (method) types and relying on input and output types that define pre- and post-conditions. In this sense, DOL relates to recent work on a typestate-oriented programming language [19] by Garcia et al. As DOL, Featherweight Typestate (FT) is a nominal object-oriented language with mutable state but whose types are enriched with state permissions. However, there are important technical differences between FT and DOL. The former is based on transitions that specify sequences of method calls explicitly, whereas DOL's method availability is less explicit. FT also allows flexible aliasing control by way of *access permissions* specified in types, whereas DOL uses only linear objects (adding a better alias control is seen as an orthogonal issue).

Extensions. We have left out of DOL’s formalisation some features that are desirable in practice. In particular, we need (1) richer index languages in domains of interest, possibly at the cost of decidable typechecking, and (2) alternatives to the current strategy for handling aliases. To relax the notion of uniqueness, we could, for instance, introduce an indirection from the main type context to a compile-time heap of objects in the style of alias types [42]. The possibility of aliasing indexed types would require a pointer to the object, allowed to be freely duplicated, while the type describing the object state must remain linear. However, to record type indirections, this approach would require additional type annotations.

References

- 1 Nada Amin, Tiark Rompf, and Martin Odersky. Foundations of path-dependent types. In *OOPSLA*. ACM Press, 2014.
- 2 Lennart Augustsson. Cayenne a language with dependent types. In *ICFP*, pages 239–250. ACM Press, 1998.
- 3 Hendrik Pieter Barendregt. *The Lambda Calculus – Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1984.
- 4 Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of Agda – A functional language with dependent types. In *TPHOLs*, volume 5674 of *LNCS*, pages 73–78. Springer, 2009.
- 5 Edwin Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23:552–593, 2013.
- 6 Patrice Chalin and Perry R. James. Non-null references by default in Java: Alleviating the nullity annotation burden. In *ECOOP*, pages 227–247. Springer, 2007.
- 7 Ravi Chugh, David Herman, and Ranjit Jhala. Dependent types for JavaScript. In *OOPSLA*, pages 587–606. ACM Press, 2012.
- 8 Maciej Cielecki, Jędrzej Fulara, Krzysztof Jakubczyk, and Lukasz Jancewicz. Propagation of jml non-null annotations in Java programs. In *Principles and Practice of Programming in Java*, pages 135–140. ACM Press, 2006.
- 9 Jeremy Condit, Matthew Harren, Zachary R. Anderson, David Gay, and George C. Necula. Dependent types for low-level programming. In *ESOP*, volume 4421 of *LNCS*, pages 520–535. Springer, 2007.
- 10 Robert L. Constable, Stuart F. Allen, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, Douglas J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, Scott F. Smith, James T. Sasaki, and S. F. Smith. *Implementing mathematics with the Nuprl proof development system*. Prentice Hall, 1986.
- 11 Thierry Coquand and Gérard P. Huet. The calculus of constructions. *Inf. Comput.*, 76(2/3):95–120, 1988.
- 12 Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.
- 13 Krishna Kishore Dhara and Gary T. Leavens. Forcing behavioral subtyping through specification inheritance. In *International Conference on Software Engineering*, pages 258–267, 1996.
- 14 Joshua Dunfield. *A Unified System of Type Refinements*. PhD thesis, Carnegie Mellon University, 2007. CMU-CS-07-129.
- 15 Joshua Dunfield and Neelakantan R. Krishnaswami. Complete and easy bidirectional typechecking for higher-rank polymorphism. In *ICFP*, pages 429–442. ACM Press, 2013.
- 16 Joshua Dunfield and Neelakantan R. Krishnaswami. Sound and complete bidirectional typechecking for higher-rank polymorphism with existentials and indexed types. *CoRR*, abs/1601.05106, 2016.

- 17 Manuel Fähndrich and K. Rustan M. Leino. Declaring and checking non-null types in an object-oriented language. In *OOPSLA*, pages 302–312. ACM Press, 2003.
- 18 Cormac Flanagan. Hybrid type checking. In *POPL*, pages 245–256, 2006.
- 19 Ronald Garcia, Éric Tanter, Roger Wolff, and Jonathan Aldrich. Foundations of typestate-oriented programming. *ACM Trans. Program. Lang. Syst.*, 36(4):12:1–12:44, 2014.
- 20 Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. Dependent classes. In *OOPSLA*, pages 133–152. ACM Press, 2007.
- 21 Simon J. Gay, Nils Gesbert, António Ravara, and Vasco Thudichum Vasconcelos. Modular session types for objects. *Logical Methods in Computer Science*, 11(4), 2015.
- 22 Robert Harper, Furio Honsell, and Gordon D. Plotkin. A framework for defining logics. *J. ACM*, 40(1):143–184, 1993.
- 23 Tony Hoare. Null references: The billion dollar mistake. QCon, 2009.
- 24 Atsushi Igarashi and Hideshi Nagira. Union types for object-oriented programming. In *Symposium on Applied Computing*, pages 1435–1441. ACM Press, 2006.
- 25 Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *TOPLAS*, 23(3):396–450, 2001.
- 26 Trevor Jim, J. Gregory Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In *USENIX*, pages 275–288. USENIX, 2002.
- 27 Tetsuo Kamina and Tetsuo Tamai. Lightweight dependent classes. In *GPCE*, pages 113–124. ACM Press, 2008.
- 28 Kenneth L. Knowles. *Executable Refinement Types*. PhD thesis, University of California, 2014.
- 29 Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. *SIGSOFT Softw. Eng. Notes*, 31(3):1–38, 2006.
- 30 K. Rustan M. Leino. Extended static checking: A ten-year perspective. In *Informatics – 10 Years Back. 10 Years Ahead*, volume 2000 of *LNCS*, pages 157–175. Springer, 2001.
- 31 Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *TOPLAS*, 16(6):1811–1841, 1994.
- 32 Per Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis-Napoli, 1984.
- 33 Conor McBride. Epigram: Practical programming with dependent types. In *Advanced Functional Programming*, volume 3622 of *LNCS*, pages 130–170. Springer, 2004.
- 34 Conor McBride. How to keep your neighbours in order. In *ICFP*, pages 297–309. ACM Press, 2014.
- 35 Aleksandar Nanevski, Greg Morrisett, Avraham Shinnar, Paul Govereau, and Lars Birkedal. Ynot: dependent types for imperative programs. In *ICFP*, pages 229–240, 2008.
- 36 Aleksandar Nanevski, J. Gregory Morrisett, and Lars Birkedal. Hoare type theory, polymorphism and separation. *Journal of Functional Programming*, 18(5-6):865–911, 2008.
- 37 Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, 2007.
- 38 Nathaniel Nystrom, Vijay Saraswat, Jens Palsberg, and Christian Grothoff. Constrained types for object-oriented languages. In *OOPSLA*, pages 457–474. ACM Press, 2008.
- 39 Benjamin C. Pierce and David N. Turner. Local type inference. *ACM Trans. Program. Lang. Syst.*, 22(1):1–44, 2000.
- 40 Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. Liquid types. In *PLDI*, pages 159–169. ACM Press, 2008.
- 41 Tim Sheard and Nathan Linger. Programming in Omega. In *Central European Functional Programming School*, volume 5161 of *LNCS*, pages 158–227. Springer, 2007.

- 42 Frederick Smith, David Walker, and J. Gregory Morrisett. Alias types. In *ESOP*, volume 1782 of *LNCS*, pages 366–381. Springer, 2000.
- 43 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^* . In *POPL*, pages 256–270. ACM Press, 2016.
- 44 Diederik T. van Daalen. *The Language Theory of Automath*. PhD thesis, Technische Hogeschool Eindhoven, Eindhoven, 1980.
- 45 Edwin Westbrook, Aaron Stump, and Ian Wehrman. A language-based approach to functionally correct imperative programming. In *ICFP*, LNCS, pages 268–279. ACM Press, 2005.
- 46 Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.
- 47 Hongwei Xi. *Dependent Types in Practical Programming*. PhD thesis, Carnegie Mellon University, Pittsburgh, 1998.
- 48 Hongwei Xi. Imperative programming with dependent types. In *LICS*, pages 375–387. IEEE Press, 2000.
- 49 Hongwei Xi. Unifying object-oriented programming with typed functional programming. In *PEPM*, pages 117–125. ACM Press, 2002.
- 50 Hongwei Xi. Applied type system: Extended abstract. In *TYPES*, pages 394–408. Springer, 2004.
- 51 Hongwei Xi. Dependent ML: an approach to practical programming with dependent types. *Journal of Functional Programming*, 17(2):215–286, 2007.
- 52 Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. In *PLDI*, pages 249–257. ACM Press, 1998.
- 53 Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *POPL*, pages 214–227. ACM Press, 1999.
- 54 Luo Zhaohui and Robert Pollack. The LEGO proof development system: A user’s manual. Technical Report ECS-LFCS-92-211, University of Edinburgh, 1992.

Static Typing of Complex Presence Constraints in Interfaces

Nathalie Oostvogels¹

Vrije Universiteit Brussel, Brussels, Belgium
noostvog@vub.ac.be

Joeri De Koster

Vrije Universiteit Brussel, Brussels, Belgium
jdekoste@vub.ac.be

Wolfgang De Meuter

Vrije Universiteit Brussel, Brussels, Belgium
wdmeuter@vub.ac.be

Abstract

Many functions in libraries and APIs have the notion of optional parameters, which can be mapped onto optional properties of an object representing those parameters. The fact that properties are optional opens up the possibility for APIs and libraries to design a complex “dependency logic” between properties: for example, some properties may be mutually exclusive, some properties may depend on others, etc. Existing type systems are not strong enough to express such dependency logic, which can lead to the creation of invalid objects and accidental usage of absent properties. In this paper we propose TypeScript_{IPC}: a variant of TypeScript with a novel type system that enables programmers to express complex presence constraints on properties. We prove that it is sound with respect to enforcing complex dependency logic defined by the programmer when an object is created, modified or accessed.

2012 ACM Subject Classification Theory of computation → Type theory, Software and its engineering → Object oriented languages, Software and its engineering → Data types and structures

Keywords and phrases type checking, interfaces, dependency logic

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2018.14

Supplement Material ECOOP Artifact Evaluation approved artifact available at <http://dx.doi.org/10.4230/DARTS.4.3.3>

1 Introduction

Static type checking enables the compile-time detection of type errors in programs, which would otherwise occur at run-time. To enable static type checking, developers have to include *type declarations* in their code. These type declarations also serve as documentation, which facilitates reasoning over code. Early type systems only describe the basic type of the values that could be stored in a variable, but throughout the years more complex types have been introduced, such as intersection types [26], union types, linear types [16] and dependent types [22]. Using these more expressive types, developers can express more sophisticated programs while retaining the compile-time guarantee that their code is correct.

¹ Funded by a PhD Fellowship of the Research Foundation - Flanders (FWO)



■ **Table 1** Twitter API documentation for sending private messages⁴.

Property name	Optional?	Description
<code>text</code>	required	The text of your direct message.
<code>user_id</code>	optional	ID of the user who should receive the direct message.
<code>screen_name</code>	optional	Screen name of the user who should receive the direct message.
Note: One of <code>user_id</code> or <code>screen_name</code> are required. ⁵		

Dynamically typed languages have given rise to new challenges in type systems, such as flow-sensitivity and optional types. One such challenge in particular is using the absence or presence of parameters to encode information. For example, a search function might require that at least one filter is specified, or objects might only be considered valid if a group of properties are all present or all absent. For singular properties, optional types can already express this. However, in order to fully resolve this challenge using static type systems, these *inter-property constraints* must be made explicit.

These types of constraints are common for Web APIs [24], where the presence of a property can determine the structure of other properties in the object of which it is a member, or where the presence of a property even *excludes* other properties. However, inter-property constraints also exist in programming languages and libraries. We show several examples of inter-property constraints, classified into three categories:

- **Exclusive constraints:** exactly one of a set of properties must be present. In the Twitter API, users can be identified by either their `user_id` or their `screen_name`. Another example is found in the Python standard library, where the function `os.utime`² sets both the access and modification time of a file. The documentation describes that the function takes two optional parameters to set the time: `times` and `ns`, moreover it states that “*It is an error to specify tuples for both `times` and `ns`*”.
- **Dependent constraints:** constraints on a property depend on the presence or the value of another property. For example, properties explaining details of a picture (name, description) should not be present if the picture property itself is not present either. In Chart.js, a library for designing charts in JavaScript, the documentation for `lines` in a chart states that “*If the `steppedLine` value is set to anything other than `false`, `lineTension` will be ignored*”.³
- **Group constraints:** a group of properties should either all be present or not present in an object. For example, latitude and longitude properties of a GPS location should always occur (or be omitted) together.

We will use a running example from the Twitter API specification to demonstrate that state-of-the-art interfaces do not suffice to describe inter-property constraints. Table 1 shows the specification for sending a private message, with a typical translation to a TypeScript interface in Listing 1. Every object that contains the input data for sending a private message should adhere to the `PrivateMessage` interface.

The accompanying note in Table 1 indicates that there is an *exclusive* constraint imposed

² <https://docs.python.org/3/library/os.html#os.utime>

³ <http://www.chartjs.org/docs/latest/charts/line.html#stepped-line>

⁴ At the time of writing, the note below the table was explicitly mentioned in the API. Recently, the description has changed — omitting the note — but the constraint still holds.

⁵ <https://developer.twitter.com/en/docs/direct-messages/sending-and-receiving/api-reference/new-message>

on the user properties. However, in TypeScript (and also in other languages) it is impossible to express that *exactly one* of `user_id` and `screen_name` is required. The question marks after `user_id` and `screen_name` in Listing 1 denote that these properties are *optional*, but this means that the type system accepts objects containing none or both of the user properties. Similarly, a group constraint with latitude and longitude properties cannot be expressed: one can mark both properties as optional, but the type system will not reject the program when only one property is provided.

```

1 interface PrivateMessage {
2   text: string;
3   user_id?: number;
4   screen_name?: string;
5 }

```

■ **Listing 1** TypeScript interface for the specification in Table 1.

The lack of support for inter-property constraints in existing programming languages causes errors to be delegated to the runtime. In the best case, the API or library provides a detailed error message, stating which properties were incompatible. Sometimes no error message is returned at all, and a silent choice is made instead: if both user properties are provided, Twitter silently chooses the screen name over the user ID.

Existing type systems are incapable of expressing inter-property constraints and statically checking these constraints both at construction time and during updates. In this paper we describe a type system that can express such complex presence constraints over multiple properties of an object. We show how interfaces with support for inter-property constraints can be incorporated in programming languages in Section 2, and describe the key features of the type system in Section 3. Sections 4 and 5 present the formalisations of the language, as a variant of TypeScript. We prove that the type system enforces both type safety and constraint integrity (Section 6). Sections 7 and 8 discuss related work and future work, respectively. Section 9 contains concluding remarks.

2 Programming with Inter-property Constraints

In this section, we propose a syntax for expressing inter-property constraints and explain intuitively how they can be used. Unless otherwise noted, every code snippet in the rest of this paper is written in `TypeScriptIPC`, our version of TypeScript with support for inter-property constraints. The syntax of `TypeScriptIPC` differs little from the syntax of TypeScript. Instead, the type system makes optimal use of the information provided by the program about the structure of objects.

2.1 Definition of interfaces with constraints

To handle inter-property constraints, the interface declaration syntax needs to be extended. Listing 2 shows an example of an interface declaration, revisiting the Twitter specification we showed in Table 1. Interfaces now consist of two parts: next to the traditional property name–type declarations, they also contain a list of constraints over the presence and absence of those properties. The syntax of constraints is as follows:

$$c \in \text{Constraints} ::= \text{present}(n) \mid (c) \mid c \wedge c \mid c \vee c \mid \neg c \mid c \rightarrow c \mid c \leftrightarrow c \mid c \text{ xor } c$$

As opposed to TypeScript and many other languages — where properties are required by default and can be made optional with a `?` annotation — properties in `TypeScriptIPC` are optional by default and are made required by adding a `present(n)` constraint.

14:4 Static Typing of Complex Presence Constraints in Interfaces

Lines 2–4 list the three properties for `PrivateMessage`, and their types in `TypeScriptIPC`. Lines 6 and 7 denote the constraints on the presence of those three properties. To improve the expressiveness of interfaces, constraints on the presence of a property can be combined with logical operators. The `PrivateMessage` interface lists two presence constraints: line 6 requires the presence of the `text` property and line 7 is the inter-property constraint from our running example. Objects can only be of an interface type if all its constraints are satisfied.

```
1 interface PrivateMessage {
2   text: string;
3   user_id: number;
4   screen_name: string;
5 } constraining {
6   present(text);
7   present(user_id) xor present(screen_name);
8 }
```

■ **Listing 2** Twitter private messaging API data expressed as interface with constraints.

The constraint definition language does not list optional properties as an explicit constraint operation, as this can be expressed by the following constraint: $\text{present}(n) \vee \neg\text{present}(n)$, which is a tautology.

Listing 3 shows another example of inter-property constraints, describing an interface of a picture object with required caption (line 7) and optional geolocation. However, the `lat` and `long` properties are dependent on the `picture` property: if the picture itself is not provided, the location should be omitted as well. In other words: the presence of the location properties implies that the picture must be present as well. These constraints are defined on lines 8 and 9. The fourth constraint on line 10 requires that the latitude and longitude properties are present or absent *together*.

```
1 interface Picture {
2   caption: string;
3   picture: string;
4   lat: number;
5   long: number;
6 } constraining {
7   present(caption);
8   present(lat) → present(picture);
9   present(long) → present(picture);
10  present(lat) ↔ present(long);
11 }
```

■ **Listing 3** Interface with dependent and group inter-property constraints.

Interfaces with inter-property constraints can also benefit from interface inheritance. For example, let us consider the case where we want a stricter version of the `PrivateMessage` interface in which only the screen name is allowed. Instead of creating a new interface, the existing interface can also be extended with extra constraints. Listing 4 shows an interface in which all properties and constraints of `PrivateMessage` are inherited, with an additional `present(screen_name)` constraint. As the `xor` constraint from `PrivateMessage` is still applicable, this interface implicitly forbids the presence of a `user_id` property.

```
1 interface PrivateMessageStrict extends PrivateMessage {
2   // reuse properties from PrivateMessage
3 } constraining {
4   present(screen_name);
5 }
```

■ **Listing 4** Extending `PrivateMessage` to require the screen name property.

2.2 Object creation

Listing 5 shows how three objects are created and assigned to three variables of type `PrivateMessage`. Even though the interface contains inter-property constraints, nothing changes for the programmer on a syntactical level. To type check this code snippet properly, the type system has to verify that the interface constraints are satisfied for that object. In the example, the first object (`msg1`) satisfies all constraints, including the exclusive constraint: only `user_id` is passed along as identification for the user. However, the type system has to generate errors for `msg2` and `msg3`, as they both violate the exclusive constraint.

```

1 var msg1: PrivateMessage = {text: "Hello", user_id: 42}; // correct
2 var msg2: PrivateMessage = {text: "Hello"}; // error: none present
3 var msg3: PrivateMessage = {text: "Hello",
4                             user_id: 42,
5                             screen_name: "Alice"}; // error: both present

```

■ **Listing 5** Creating objects with inter-property constraints.

The type system also needs to ensure that no constraints are violated when expressions with different interface types are assigned to each other, or when an instance of an interface is assigned to a variable with a regular object literal type.

2.3 Property access

When inter-property constraints are involved, reading object properties requires extra caution. The type system should only allow the access of a property when that property is guaranteed to be present. For example, the property `text` in the `PrivateMessage` interface is a required property and thus it is certain this property is always present in objects of type `PrivateMessage`.

By contrast, the type system should reject programs where other properties of a `PrivateMessage` object are accessed. The exclusive constraint guarantees that exactly one of `user_id` and `screen_name` will be present, but it is not known *which* property actually is. The function `getUserId` (defined in Listing 6) tries to read the `user_id` of a `PrivateMessage`, which generates a type error as this property access is unsafe.

To prevent errors from accessing undefined properties, programmers must verify whether properties are present before using them. For example, the function `getUser` first performs a test to check whether `user_id` is present. Inside the true branch, access to the user ID (line 6) must be allowed. Additionally, because there is an inter-property constraint between `user_id` and `screen_name`, the `screen_name` property is guaranteed to be absent even though we did not explicitly test for it. The inverse holds in the false branch.

Similarly, in the function `getLocation` (which retrieves the longitude and latitude of a picture), the type system has to allow the access of `long`, which follows directly from the if statement. On top of that, the type system should also accept accessing the properties `lat` and `picture`, which are both guaranteed to be present if `long` is present.

```

1 function getUserId(msg: PrivateMessage) : number {
2   return msg.user_id; // error: user_id is not guaranteed to be present
3 }
4 function getUser(msg: PrivateMessage) {
5   if (msg.user_id !== undefined) {
6     msg.user_id; // :: number (present due to if statement)
7     msg.screen_name; // :: undefined (not present due to xor constraint)
8   } else {
9     msg.user_id; // :: undefined (not present due to if statement)
10    msg.screen_name; // :: string (present due to xor constraint)

```

14:6 Static Typing of Complex Presence Constraints in Interfaces

```
11 }
12 }
13 function getLocation(picture: Picture) {
14   if (picture.long !== undefined) {
15     picture.long; // :: number (present due to if statement
16     picture.lat; // :: number (present due to group constraint)
17     picture.picture; // :: string (present due to dependent constraint)
18   }
19 }
```

■ Listing 6 Accessing properties

2.4 Property updates

As with every object-oriented type system, the assignment of a new value to a property of an object should only succeed when the value is of the correct type. Inter-property constraints add an extra complication: assigning to a property might invalidate an inter-property constraint.

Updating a property that was already guaranteed to be present is safe: the previous section showed that the type system will only assign the intended type to properties that are known to be present. Line 2 in Listing 7 illustrates this with the `text` property. The update of the `user_id` property on line 4 will fail, however: the type system disallows the property access, as explained in the previous section.

Note that it is not allowed to assign the value `undefined` to properties of any type except `Undefined`, as this would make a required property absent (line 3). This principle is known as the *strict null-checking* mode of TypeScript. In Listing 7, it is only allowed to assign `undefined` to `screen_name` (line 8), as this property is known to be absent inside the consequent of the if statement.

```
1 function setMsg(msg: PrivateMessage, text: string, user_id: number) {
2   msg.text = text; // ok
3   msg.text = undefined; // error: assigning undefined to present property
4   msg.user_id = user_id; // error: property with unknown presence status
5
6   if (msg.user_id !== undefined) {
7     msg.user_id = user_id; // ok
8     msg.screen_name = undefined; // ok
9   }
10 }
```

■ Listing 7 Updating properties.

The examples of Listing 7 only modify one property at a time. However, an inter-property constraint often requires the modification of several properties at once, as the object could be in a type-incorrect state inbetween several assignments. Let us consider the case in Listing 8 where a programmer wants to switch from user ID to screen name. The type system rejects this program, as it breaks the rules imposed by the strict-null checking mode. This behaviour is desirable: inbetween lines 3 and 4, the inter-property constraint of `msg` is violated: it contains neither user ID nor screen name.

```
1 var msg: PrivateMessage = {text: "Hello", user_id: 42};
2 if (msg.user_id !== undefined) {
3   msg.user_id = undefined;
4   msg.screen_name = "Alice";
5 }
```

■ Listing 8 Changing an inter-property constraint is not possible with separate assignments.

Our solution is to enable updating of multiple properties simultaneously, such that the object is never in an invalid state between consecutive assignment statements. We propose an `assign(i, o)` operator⁶ that returns a *copy* of object *i*, in which the properties from the object *o* are added or updated. Listing 9 shows how the `assign` operator switches from `user_id` to `screen_name`. Note that `assign` is functional: instead of modifying its first arguments, it returns a new object.

```

1 var msg: PrivateMessage = {text: "Hello", user_id: 42};
2 var msg2: PrivateMessage =
3   assign(msg, {user_id: undefined, screen_name: "Alice"}); // correct
4 var msg3: PrivateMessage =
5   assign(msg, {user_id: undefined}); // incorrect

```

■ **Listing 9** Using multi-assign to switch from user ID to screen name.

While programmers can update any subset of the properties of an object, not all combinations are correct, as the `msg3` example above shows. Intuitively, if an inter-property constraint exists between two or more properties, they should all appear together in the call to `assign`. The properties of an object can thus be divided into one or more “clusters”. For example a `Picture` object has a trivial cluster for `caption`, and a separate cluster for the `long`, `lat` and `picture` properties.

3 Verifying Constraints in TypeScript

The addition of constraints to interfaces has consequences on several facets of the type system. In the following sections, we explain how the type system of `TypeScriptIPC` deals with the creation, modification, and access of properties of interfaces with constraints. Because the constraint language expresses constraints with logical connectives, the type system uses several concepts from propositional logic to guarantee correctness.

3.1 Object literals have to satisfy constraints

The type system only accepts the assignment of an object literal to a variable with an interface type when that object satisfies the interface constraints. Using terminology from propositional logic, the type system requires that the object literal is a *valuation* [15] that satisfies the logical formulas of the interface (constraints). More specifically, an object literal defines a valuation, assigning truth values (presence and absence of properties) to proposition symbols (property names). Moreover, for every valuation *v* there exists a unique function \hat{v} which takes a proposition (here: the constraints) and returns true or false.

3.2 Constraints dictate property presence

As with other type systems, interface declarations contain a list of properties with their types. However, looking up a property of an interface may only succeed when the interface contains a constraint indicating that property is present. Of course, with complex inter-property constraints, these constraints may not be *directly* present in the constraint set. Instead, the type system relies on *logical entailment* (denoted \models) to verify whether a `present(n)` constraint follows from a set of constraints. Calculating logical entailments can be efficiently automated using deductive systems such as the Gentzen system [15]. Returning to the `PrivateMessage`

⁶ `assign` resembles the `Object.assign` function in JavaScript, but does not modify its input object.

14:8 Static Typing of Complex Presence Constraints in Interfaces

example, the type system verifies the following logical entailment for accessing the `text` property:

$$\{\text{present}(\text{text}); \text{present}(\text{user_id}) \text{ xor } \text{present}(\text{screen_name})\} \models_{\ell} \text{present}(\text{text})$$

Similarly, inter-property constraints can also guarantee the *absence* of a property. In the case where neither the presence or absence of a property can be derived from the constraints, the type system should conservatively reject the access of that property. This also follows from the logical entailment. For example, the type checker rejects the function `getUserId` of Listing 6, because neither the presence nor the absence of `user_id` is a logical consequence of the interface constraints:

$$\begin{aligned} \{\text{present}(\text{text}); \text{present}(\text{user_id}) \text{ xor } \text{present}(\text{screen_name})\} &\not\models_{\ell} \text{present}(\text{user_id}) \\ \{\text{present}(\text{text}); \text{present}(\text{user_id}) \text{ xor } \text{present}(\text{screen_name})\} &\not\models_{\ell} \neg \text{present}(\text{user_id}) \end{aligned}$$

3.3 Explicit property presence tests

In dynamic languages, it is common to perform runtime property presence tests. These presence tests can provide the type system with more information about the object being tested: in one branch it is certain that the property is present, while it is guaranteed to be absent in the other. For the `true` branch in the function `getUser` of Listing 6, the type system simply adds the new information (`present(user_id)`) to the set of constraints, to allow the access of the `user_id` property.

That extra information can trigger other inter-property constraints, thus guaranteeing the presence or absence of other properties. Using logical entailment, the type system can prove that `screen_name` will not be present:

$$\left\{ \begin{array}{l} \text{present}(\text{text}); \\ \text{present}(\text{user_id}) \text{ xor } \text{present}(\text{screen_name}); \\ \text{present}(\text{user_id}); \end{array} \right\} \models_{\ell} \neg \text{present}(\text{screen_name})$$

Similarly, the presence check on longitude in `getLocation` guarantees that the longitude is present, but also suffices to safely access latitude (by combining the constraint `present(long) ↔ present(lat)` with `present(long)`) and the picture itself (combining constraints `present(long) → present(picture)` and `present(long)`).

3.4 Interface–interface compatibility

Normally, an instance of interface I_0 is considered assignable to a variable with as type another interface I_1 if I_0 contains at least every property and method in the other interface. However, with the addition of constraints we must also take care that no instance of I_0 violates the constraints in I_1 . To guarantee that all constraints of I_1 are satisfied, every constraint from I_1 must be a *logical entailment* of the constraints in I_0 . Properties which are absent from I_0 result in extra `¬present(n)` constraints at the left-hand side of the entailment.

For example, assigning a variable with a more strict interface type `PrivateMessage2` (defined in Figure 1) to a variable of type `PrivateMessage`, gives rise to the following logical entailment. Next to the constraints of `PrivateMessage`, the left side of the logical entailment contains an extra constraint due the absence of the screen name in `PrivateMessage2`. Without the third constraint, the logical entailment would not be valid.

$$\left\{ \begin{array}{l} \text{present}(\text{text}); \\ \text{present}(\text{user_id}); \\ \neg \text{present}(\text{screen_name}) \end{array} \right\} \models_{\ell} \begin{array}{l} \text{present}(\text{text}) \wedge \\ \text{present}(\text{user_id}) \text{ xor } \text{present}(\text{screen_name}) \end{array}$$

<pre> 1 interface PrivateMessage1 { 2 text: string; 3 user_id: number; 4 screen_name: string; 5 } constraining { 6 present(text); 7 present(user_id); 8 present(screen_name); 9 }</pre>	<pre> interface PrivateMessage2 { text: string; user_id: number; } constraining { present(text); present(user_id); }</pre>
---	--

■ **Figure 1** Other versions of the PrivateMessage interface.

As for properties, one might expect that I_0 may contain a superset of the properties in I_1 . However, this can lead to constraint violations: consider the following example, with two variations on the `PrivateMessage` interface (defined in Figure 1).

```

1 var msg1: PrivateMessage1 = {text:"Hello",user_id:42,screen_name:"Alice"};
2 var msg2: PrivateMessage2 = msg1;
3 var msg3: PrivateMessage = msg2;
```

On line 2, a variable of type `PrivateMessage1` is assigned to a variable of type `PrivateMessage2` and line 3 assigns a variable of type `PrivateMessage2` to a variable of the default `PrivateMessage` interface: both assignments would be allowed, as no constraints are violated. However, line 3 would result in an object of type `PrivateMessage` that contains both `user_id` and `screen_name`, violating its constraints.

Evidently, width subtyping is irreconcilable with a type system that requires the absence of properties. Therefore, the type system has to counter-intuitively require that the interface I_0 only contains properties other than those in I_1 when those properties are guaranteed to be absent. This is not the case for the second assignment (line 2) in the example:

$$\{\text{present}(\text{text}); \text{present}(\text{user_id}); \text{present}(\text{screen_name})\} \not\sqsubseteq_{\ell} \neg \text{present}(\text{screen_name})$$

3.5 Updated objects have to satisfy constraints

To verify that all constraints are still satisfied after a simultaneous update of multiple properties, the type system again uses valuations. However, as the update only affects a subset of the properties, the object literal in the second argument only serves as a valuation for a subset of the constraints.

Consider the following example of an interface that indicates both the sender (with the `s_*` properties) and the receiver (`r_*`). Logically, these properties form separate clusters that are not affected by each other.

<pre> 1 interface PrivateMessage3 { 2 text: string; 3 r_user_id: number; 4 r_screen_name: string; 5 s_user_id: number; 6 s_screen_name: string; 7 } constraining { 8 present(text); 9 r_user_id xor r_screen_name; 10 s_user_id xor s_screen_name; 11 }</pre>	<pre> var msg:PrivateMessage3 = {text: "Hello", r_user_id: 42, s_user_id: 43}; var msg2 = assign(msg, {r_user_id: undefined, r_screen_name: "Alice"}); \[\]</pre>
--	---

The `assign` at the right side only updates the receiver of the private message. Therefore, the constraints for the sender side do not have to be taken into account: the `assign` operation

type checks if the object literal is a valid valuation of the constraint on line 9. This is the case, as `undefined` is interpreted as an absent property. Of course, the types of properties in the object literal must conform to those defined in the interface (with the exception of undefined properties). Note that an update is only valid when all properties of the cluster are updated.

4 TypeScript_{IPC}: A Variant of TypeScript with Constraints

Section 2 showed how constraints on the presence of properties can be added to TypeScript's interfaces and Section 3 gave an informal idea of how the type system statically enforces that constraints stay satisfied throughout the program. In this section, we formalise these ideas in TypeScript_{IPC}, a variant of TypeScript.

TypeScript is an extension of JavaScript which adds optional static typing. It provides extra features over JavaScript such as structural typing and named interfaces. To ensure compatibility with existing JavaScript code, type annotations in TypeScript are optional which enables developers to gradually convert existing JavaScript code to TypeScript.

This section introduces TypeScript_{IPC}. The syntax, semantics and type rules presented in this section build upon those presented by Bierman et al. [7]. They present the type system in two parts: the first is a safe calculus (called safeFTS) which contains the core features of TypeScript, including structural typing, contextual types and the lack of block scoping in JavaScript. The second part expands safeFTS to a production-ready calculus, which is unsafe.

TypeScript_{IPC} reuses most of safeFTS's features, which are based upon TypeScript 0.9.5. However, as checking the presence or absence of properties is a key feature of TypeScript_{IPC}, we use the subtyping rules from the strict null checking mode in TypeScript 2.0. These make it illegal to assign `null` and `undefined` to variables of any other type, unless explicitly allowed.

Our variant of TypeScript with constraints will focus on objects and interfaces. Contextual typing and constructs to deal with the lack of block scoping are omitted for clarity. As they are orthogonal to object creation and interfaces, they can be trivially added to the language presented in this paper.

4.1 Syntax

Figure 2 presents the syntax of TypeScript_{IPC}, which is based on the syntax presented in [7]. It features basic language expressions such as identifiers, literals, assignment and binary operators. Literals can be numbers n , strings s , or one of the following constants: `true`, `false`, `null` and `undefined`, where `null` indicates the empty object and `undefined` is returned when accessing a property that is not present in an object.

Objects are defined using object literals, which map property names to values. Multiple properties of an object can be updated at once using `assign`. This function returns a new object that contains all properties of the first argument. Properties from the second argument are either updated (when already present in the first argument) or added (otherwise). Function expressions are similar to those in JavaScript, but with type annotations for the parameters. Expressions can be cast to a type, but only when the cast is known to be correct. Statements and variable declarations are straightforward. TypeScript_{IPC} only features variable declarations where the type and the value for the variable are provided.

The empty sequence is denoted with \bullet , a concatenation is denoted using a comma, and a sequence of expressions is written as \bar{e} . A sequence of property assignments $\{\bar{n} : \bar{e}\}$ is an

$e, f \in \text{Expressions}$	$::=$	x	(Identifier)
		l	(Literal)
		$\{\bar{a}\}$	(Object literal)
		$e = f$	(Assignment operator)
		$\text{assign}(e, \{\bar{a}\})$	(Assign operator)
		$e \otimes f$	(Binary operator)
		$e.n$	(Property access)
		$e(\bar{f})$	(Function call)
		$\langle T \rangle e$	(Type assertion)
		$\text{function } (\bar{x} : \bar{T}) : S \{ \bar{s} \}$	(Function expression)
$a \in \text{Property assignments}$	$::=$	$n : e$	(Property assignment)
$s, t \in \text{Statements}$	$::=$	$e;$	(Expression statement)
		$\text{if } (e) \{ \bar{s} \} \text{ else } \{ \bar{t} \}$	(If statement)
		$\text{return};$	(Return statement)
		$\text{return } e;$	(Return value statement)
		$\text{var } x : T = e$	(Variable declaration)

■ **Figure 2** Syntax of TypeScript_{IPC}.

abbreviation for $\{n_1 : e_1, \dots, n_n : e_n\}$, with n the length of the sequence. Similarly, $(\bar{x} : \bar{T})$ is a sequence of function arguments $(x_1 : T_1, \dots, x_n : T_n)$.

To reduce the size and complexity of our formalisation, we omit parts of safeFTS that do not contribute to the necessary adaptations for inter-property constraints. More specifically, TypeScript_{IPC} does not support computed property accesses, untyped identifiers, call signatures without parameter types or return types, and untyped and uninitialised variable declarations.

Figure 3 shows that TypeScript_{IPC} has three kinds of types: the top type **any**, primitive types and object types. An object type is represented by either a literal type or an interface type. Note that functions are represented as callable objects that contain one field with its type of the form $(\bar{x} : \bar{S}) : T$. A sequence of types is denoted as \bar{T} , and the sequence of properties and call signatures is analogous to their corresponding value sequences.

Interfaces play a key role in expressing inter-property constraints, and their declaration in TypeScript_{IPC} is different from other languages:

$$D \in \text{Declarations} ::= \begin{cases} \text{interface } I \{ \bar{n} : \bar{T} \} \text{ constraining } \{ \bar{c} \} \\ \text{interface } I \text{ extends } \bar{I} \{ \bar{n} : \bar{T} \} \text{ constraining } \{ \bar{c} \} \text{ } (\bar{I} \text{ non-empty}) \end{cases}$$

TypeScript_{IPC} interfaces first list the property (field or method) names, together with their types as usual. However, constraints on the presence of a property are specified in the **constraining** section, using the syntax presented in Section 2.1. By default, all properties are optional unless marked as **present**. In addition, the **constraining** section can impose inter-property constraints on properties of the interface. Interfaces can inherit properties and constraints from other interfaces. TypeScript_{IPC} does not allow interfaces to define properties with the same name as any of their superinterfaces. Furthermore, all properties are public.

To retrieve the properties and constraints from a given interface, we define two auxiliary functions *properties* and *constraints*. Analogous to the inheritance of properties, constraints from the superinterfaces are simply accumulated.

14:12 Static Typing of Complex Presence Constraints in Interfaces

$R, S, T \in \text{Types}$	$::=$	any	
		P	
		0	
$P \in \text{Primitive types}$	$::=$	number	
		string	
		boolean	
		void	
		Null	
		Undefined	
$O \in \text{Object types}$	$::=$	I	(Interface type)
		L	(Literal type)
$L \in \text{Object literal types}$	$::=$	$\{\bar{M}\}$	
$M, N \in \text{Type members}$	$::=$	$n:T$	(Property)
		$(\bar{x} : \bar{S}) : T$	(Call signature)

■ **Figure 3** Types of TypeScript_{IPC}.

$$\text{Property lookup (1)} \frac{\Sigma_i(\mathbf{I}) = \text{interface } \mathbf{I} \{ \bar{n} : \bar{T} \} \text{ constraining } \{ \bar{c} \}}{\text{properties}(\mathbf{I}) = \{ \bar{n} : \bar{T} \}}$$

$$\text{Property lookup (2)} \frac{\Sigma_i(\mathbf{I}) = \text{interface } \mathbf{I} \text{ extends } \bar{\mathbf{I}} \{ \bar{n} : \bar{T} \} \text{ constraining } \{ \bar{c} \}}{\text{properties}(\mathbf{I}) = \{ \bar{n} : \bar{T} \} \cup \text{properties}(\bar{\mathbf{I}})}$$

$$\text{Constraint lookup (1)} \frac{\Sigma_i(\mathbf{I}) = \text{interface } \mathbf{I} \{ \bar{n} : \bar{T} \} \text{ constraining } \{ \bar{c} \}}{\text{constraints}(\mathbf{I}) = \{ \bar{c} \}}$$

$$\text{Constraint lookup (2)} \frac{\Sigma_i(\mathbf{I}) = \text{interface } \mathbf{I} \text{ extends } \bar{\mathbf{I}} \{ \bar{n} : \bar{T} \} \text{ constraining } \{ \bar{c} \}}{\text{constraints}(\mathbf{I}) = \{ \bar{c} \} \cup \text{constraints}(\bar{\mathbf{I}})}$$

Before analysis starts, all interface declarations are gathered and stored in a mapping Σ_i of interface names \mathbf{I} to their respective declaration D . As in safeFTS, a program is a pair (Σ_i, \bar{s}) containing an interface table and a sequence of statements. TypeScript_{IPC} requires every interface to satisfy a set of sanity conditions:

1. For every $\mathbf{I} \in \text{dom}(\Sigma_i)$, $\Sigma_i(\mathbf{I}) = \text{interface } \mathbf{I} \{ \bar{n} : \bar{T} \} \text{ constraining } \{ \bar{c} \}$ or $\Sigma_i(\mathbf{I}) = \text{interface } \mathbf{I} \text{ extends } \bar{\mathbf{I}} \{ \bar{n} : \bar{T} \} \text{ constraining } \{ \bar{c} \}$;
2. for every interface name \mathbf{I} appearing anywhere in Σ_i , it is the case that $\mathbf{I} \in \text{dom}(\Sigma_i)$;
3. there are no cycles in the dependency graph induced by the **extends** clauses of the interface declarations defined in Σ_i ;
4. for every interface name \mathbf{I} in $\text{dom}(\Sigma_i)$, there exists at least one valuation (that assigns truth values (indicating presence or absence) to proposition symbols (property names)) that satisfies the constraints ($\text{constraints}(\mathbf{I})$);
5. for every interface name \mathbf{I} in $\text{dom}(\Sigma_i)$, none of the properties of \mathbf{I} is allowed to be of type **any** or **Undefined**.

The first three sanity conditions are common, and almost identical to those in safeFTS, the latter two are specifically for interfaces with inter-property constraints. The fourth condition prevents the declaration of interfaces with inherent contradictions, and the fifth condition prevents the assignment of **undefined** to an object property, which — at runtime — is equal to an absent property.

4.2 Type System

In this section we present the type system of $\text{TypeScript}_{\text{IPC}}$. Figure 4 shows the type rules of $\text{TypeScript}_{\text{IPC}}$, which are based on those of safeFTS. For clarity, we omit contextual typing and JavaScript’s lack of block scoping from the typing rules, which are orthogonal extensions to the contribution in this paper. The typing judgement is written as follows: $\Gamma \vdash e : T$, where given an environment Γ the expression e is of type T . Γ maps variables to types ($\bar{x} : \bar{T}$) and is extended as follows: $\Gamma, x : T$. For sequences, we write $\Gamma \vdash \bar{e} : \bar{T}$ as shorthand for $\Gamma \vdash e_1 : T_1, \dots, \Gamma \vdash e_n : T_n$, with n the length of the sequence. $\bar{S} \leq T$ is an abbreviation for $S_1 \leq T, \dots, S_n \leq T$ and we write $\bar{S} \leq \bar{T}$ as shorthand for $S_1 \leq T_1, \dots, S_n \leq T_n$.

The rules that do not (directly) deal with interfaces are standard: I-Id looks up a variable in the environment. I-Number, I-String, I-Bool, I-Null and I-Undefined all type check a constant. The type of an object literal is a mapping of all property names onto the type of their expression (I-ObjLit). In I-Op, the type system checks that the parameters have the expected type.

4.2.1 Property lookup

I-Prop first retrieves the type of the object, and then determines the type of the property using the *lookup* function:

$$lookup(S, n) = \begin{cases} lookup(\text{Number}, n) & \text{if } S = \text{number} \\ lookup(\text{Boolean}, n) & \text{if } S = \text{boolean} \\ lookup(\text{String}, n) & \text{if } S = \text{string} \\ T & \text{if } S = \{\bar{M}_0, n : T, \bar{M}_1\} \\ lookup(\text{Object}, n) & \text{if } S = \{\bar{M}\} \text{ and } n \notin \bar{M} \\ T & \text{if } S = I \text{ and } n : T \in \text{properties}(I) \\ & \text{and } \text{constraints}(I) \models_{\ell} \text{present}(n) \\ \text{Undefined} & \text{if } S = I \text{ and } n : T \in \text{properties}(I) \\ & \text{and } \text{constraints}(I) \models_{\ell} \neg \text{present}(n) \end{cases}$$

Properties of primitive types are looked up in their associated interface type (lines 1–3). Looking up a property in an object literal type is as expected (line 4). When the property is not found in the object literal type, the *lookup* function searches the property in the Object type (line 5). The last two lines show how a property is looked up in a $\text{TypeScript}_{\text{IPC}}$ interface. Simply looking up the property in the list of interface properties does not suffice: as shown in Section 3.2, the *constraints* on an interface type dictate the presence of its properties. If the property is guaranteed to be present, *lookup* returns its type, otherwise it returns **Undefined**. If neither the presence nor the absence of a property can be guaranteed, the *lookup* function is not defined.

4.2.2 Assignment Compatibility

In I-Assign, a new expression may only be assigned to an expression when the new expression has a type that is *assignable to* the type of the original expression. Similarly, I-Call uses the assignment compatibility relationship to check that the parameters of the function call have the correct type. When type checking a function definition, I-Func extends the environment as usual with the type declarations for the parameters, and type **any** for the **this** variable. The return types of the function body must all be assignable to the declared return type. As

$$\begin{array}{c}
 \text{I-Id} \frac{}{\Gamma, x:T \vdash x:T} \quad \text{I-Number} \frac{}{\Gamma \vdash n : \text{number}} \quad \text{I-String} \frac{}{\Gamma \vdash s : \text{string}} \\
 \\
 \text{I-Bool} \frac{}{\Gamma \vdash \text{true}, \text{false} : \text{boolean}} \quad \text{I-Null} \frac{}{\Gamma \vdash \text{null} : \text{Null}} \\
 \\
 \text{I-Undefined} \frac{}{\Gamma \vdash \text{undefined} : \text{Undefined}} \quad \text{I-ObjLit} \frac{\Gamma \vdash \bar{e} : \bar{T}}{\Gamma \vdash \{\bar{n} : \bar{e}\} : \{\bar{n} : \bar{T}\}} \\
 \\
 \text{I-Op} \frac{\Gamma \vdash e : S_0 \quad \Gamma \vdash f : S_1 \quad S_0 \otimes S_1 = T}{\Gamma \vdash e \otimes f : T} \quad \text{I-Prop} \frac{\Gamma \vdash e : S \quad \text{lookup}(S, n) = T}{\Gamma \vdash e.n : T} \\
 \\
 \text{I-Assign} \frac{\Gamma \vdash e : S \quad \Gamma \vdash f : T \quad T \leq S}{\Gamma \vdash e = f : T} \quad \text{I-Call} \frac{\Gamma \vdash e : \{(\bar{x} : \bar{S}) : \bar{R}\} \quad \Gamma \vdash \bar{f} : \bar{T} \quad \bar{T} \leq \bar{S}}{\Gamma \vdash e(\bar{f}) : \bar{R}} \\
 \\
 \text{I-Func} \frac{\Gamma, \text{this} : \text{any}, \bar{x} : \bar{T} \vdash \bar{s} : \bar{R} \quad \bar{R} \leq S}{\Gamma \vdash \text{function}(\bar{x} : \bar{T}) : S \{ \bar{s} \} : \{(\bar{x} : \bar{T}) : S\}} \quad \text{I-Assert} \frac{\Gamma \vdash e : S \quad S \leq T}{\Gamma \vdash \langle T \rangle e : T} \\
 \\
 \text{I-AssertInf} \frac{\Gamma \vdash \{\bar{n} : \bar{e}\} : \{\bar{M}\} \quad \{\bar{M}_p\} = \{n : T \mid n : T \in \{\bar{M}\} \wedge T \neq \text{Undefined}\} \quad \{\bar{M}_p\} \subseteq \text{properties}(\mathbb{I}) \quad c_p = \{\text{present}(n) \mid n : T \in \{\bar{M}_p\}\} \quad \{\bar{M}_{np}\} = \text{properties}(\mathbb{I}) \setminus \{\bar{M}_p\} \quad c_{np} = \{\neg \text{present}(n) \mid n : T \in \{\bar{M}_{np}\}\} \quad v = c_p \cup c_{np} \quad \hat{v}(\text{constraints}(\mathbb{I})) = \text{true}}{\Gamma \vdash \langle \mathbb{I} \rangle \{\bar{n} : \bar{e}\} : \mathbb{I}} \\
 \\
 \text{I-UpdateObj} \frac{\Gamma \vdash e : \{\bar{M}\} \quad \Gamma \vdash \{\bar{n} : \bar{e}\} : \{\bar{N}\}}{\Gamma \vdash \text{assign}(e, \{\bar{n} : \bar{e}\}) : \{\bar{M}\} \uplus \{\bar{N}\}} \\
 \\
 \text{I-UpdateInf} \frac{\Gamma \vdash e : \mathbb{I} \quad \Gamma \vdash \langle \mathbb{I}' \rangle \{\bar{n} : \bar{e}\} : \mathbb{I}' \quad \mathbb{I}' = \text{slice}(\mathbb{I}, \bar{n}, \text{constraints}(\mathbb{I})) \quad \bar{n} \in \text{dom}(\text{properties}(\mathbb{I})) \quad \bar{n} = \text{dom}(\text{properties}(\mathbb{I}'))}{\Gamma \vdash \text{assign}(e, \{\bar{n} : \bar{e}\}) : \mathbb{I}}
 \end{array}$$

■ **Figure 4** Type rules of TypeScript_{IPC}.

only safe casts are allowed in TypeScript_{IPC}, casting an expression to another type is only allowed when the original type is assignable to the cast type (I-Assert).

The assignment compatibility relation is defined in Figure 5, and is based on the rules of safeFTS. In safeFTS, interfaces are replaced by corresponding object literals. When an interface (indirectly) references itself in its field declarations, this can lead to an infinite type expansion. To deal with this, safeFTS defines assignment compatibility as a coinductive relation, which guarantees termination. In TypeScript_{IPC}, on the other hand, interfaces cannot be replaced by object literals, as interfaces may also contain constraints. Thus, assignment compatibility for interface fields with interface types in TypeScript_{IPC} must be checked against the interface definition instead of via a coinductive relation.

First, assignment compatibility is transitive (A-Trans) and reflexive (A-Refl). Any type can be assigned to **any** (A-AnyR). **null** can only be assigned to itself or **any**, and **undefined** can only be assigned to itself, **any** or **void** (A-Undefined). For assigning primitive types, A-Prim looks up their interface type. An object literal type can be assigned to another object literal type when all the properties of the source object are also present on the target object, and properties are assignable pairwise (A-Object). A-Prop defines that assigning

properties to each other is invariant. Assigning call signatures is contra-/co-variant (A-CS and A-CS-Void). A-Interface is as discussed in Section 3.4: interfaces must be at least as strict as the target interface to be considered assignment-compatible, and common properties should have the same type. Extra properties on I_0 are not allowed, unless their absence can be proven from the constraints. A-IntObj allows assigning an interface to an object when the constraints on the interface guarantee that all properties are present.

Due to width subtyping, the type of an object does not guarantee that *only* those properties are present at runtime (as can be seen in A-Object). However, width subtyping conflicts with inter-property constraints, that may require properties to be absent: the assignment of an object to an interface could possibly invalidate the interface constraints at runtime. Therefore, there is no assignment compatibility rule for assigning an object to an interface: TypeScript_{IPC} only allows the casting of a *literal* object to an interface. This is covered by the rule I-AssertInf (covered in Section 4.2.3). By only allowing object literals (instead of all object literal types), the type system has an exact view of the properties that are present and can thus guarantee that the interface constraints are satisfied.

A small study⁷ on web APIs indicates that this is not a severe restriction. The study explored a list of GitHub projects that use an SDK to send requests to the Twitter and YouTube API. In 163 of the 180 studied API calls, the data was provided as an object literal. In 14 out of the 17 cases where the data argument was not an object literal, the object was defined directly above the API call.

Note that, as a consequence, the examples in Section 2 that create objects with inter-property constraints (Listing 5) are only accepted by the type checker if they are first typecast to `PrivateMessage`.

4.2.3 Creating and updating

The rule I-AssertInf covers the case where an object literal is cast to an interface. As explained in Section 3.1, the cast only succeeds when the properties of the object have the correct type *and* the presence and absence of properties form a valid valuation of the constraints. A property is considered absent when it is not in the object literal, or when its type is `Undefined`.

I-UpdateInf and I-UpdateObj cover updating multiple properties of an object at once, using the functional `assign` function (see Section 3.5). When the type of the first argument of `assign` is an object literal type, I-UpdateObj simply combines (updates or adds, when the property is already present resp. not present in the first argument) the properties of the second argument with the first, using \uplus . More caution is required when the type of `e` is an interface, as updating properties could invalidate the constraints. As the second argument does not necessarily contain every property of the interface, it does not suffice to check whether the new properties satisfy all the constraints. To solve this, I-UpdateInf uses the *slice* function (defined below) to generate an interface that only contains constraints concerning the properties that are being updated. Given this generated interface, rule I-AssertInf is reused to verify whether the updated properties satisfy the applicable subset of constraints. An `assign` fails if any of the updated properties are not declared in the interface `I`, or when not all properties of `I'` are part of the second argument of `assign`.

To preserve soundness, `assign` does not modify its first argument; instead it returns a fresh object. Allowing `assign` to mutate the object would impose severe usage restrictions (such as in Flow [10] and RSC [34]), or requires tracking aliases (such as in DJS [11]).

⁷ <http://soft.vub.ac.be/~noostvog/typescriptipc/olrestriction.pdf>

$$\begin{array}{c}
 \text{A-Trans} \frac{R \leq S \quad S \leq T}{R \leq T} \qquad \text{A-Refl} \frac{S \vdash \diamond}{S \leq S} \qquad \text{A-AnyR} \frac{S \vdash \diamond}{S \leq \text{any}} \\
 \\
 \text{A-Undefined} \frac{}{\text{Undefined} \leq \text{void}} \qquad \text{A-Prim} \frac{\mathcal{I}(P) \leq T}{P \leq T} \\
 \\
 \text{A-Object} \frac{\{\bar{M}_0, \bar{M}_1\} \vdash \diamond \quad \bar{M}_1 \leq \bar{M}_2}{\{\bar{M}_0, \bar{M}_1\} \leq \{\bar{M}_2\}} \qquad \text{A-Prop} \frac{}{n : T \leq n : T} \\
 \\
 \text{A-CS} \frac{\bar{T} \leq \bar{S} \quad R_1 \neq \text{void} \quad R_0 \leq R_1}{(\bar{x} : \bar{S}) : R_0 \leq (\bar{y} : \bar{T}) : R_1} \qquad \text{A-CS-Void} \frac{\bar{T} \leq \bar{S} \quad R \vdash \diamond}{(\bar{x} : \bar{S}) : R \leq (\bar{y} : \bar{T}) : \text{void}} \\
 \\
 \text{A-Interface} \frac{\begin{array}{l} \forall n : S \in \text{properties}(\mathbb{I}_0) \wedge n : T \in \text{properties}(\mathbb{I}_1) : S = T \\ c_0 = \{\neg \text{present}(n) \mid n : T \in \text{properties}(\mathbb{I}_0) \setminus \text{properties}(\mathbb{I}_1)\} \\ c_1 = \{\neg \text{present}(n) \mid n : T \in \text{properties}(\mathbb{I}_1) \setminus \text{properties}(\mathbb{I}_0)\} \\ \text{constraints}(\mathbb{I}_0) \cup c_1 \vDash_\ell \wedge \text{constraints}(\mathbb{I}_1) \wedge c_0 \end{array}}{\mathbb{I}_0 \leq \mathbb{I}_1} \\
 \\
 \text{A-IntObj} \frac{\text{properties}(\mathbb{I}) \leq \{\bar{M}\} \quad \{\bar{n} : \bar{T}\} = \{\bar{M}\} \quad \text{constraints}(\mathbb{I}) \vDash_\ell \text{present}(\bar{n})}{\mathbb{I} \leq \{\bar{M}\}}
 \end{array}$$

■ **Figure 5** Assignment compatibility for types in TypeScript_{IPC}.

slice returns the transitive closure of all properties and constraints of the given interface which are affected by the properties being updated. Formally, *slice* is defined as follows. It uses an auxiliary function *fv* which takes a constraint and returns all referenced properties.

$$\text{slice}(\mathbb{I}, \bar{p}, \bar{c}) = \begin{cases} \text{interface } \mathbb{I}' \{ \bar{p} \} \text{ constraining } \{ \bar{c} \} & \text{if } (\bar{p}, \bar{c}) \equiv (\bar{p}', \bar{c}') \\ \text{slice}(\mathbb{I}, \bar{p}', \bar{c}') & \text{otherwise} \end{cases}$$

$$\begin{aligned}
 \text{where } \bar{c}' &= \bar{c} \cup \{c \mid c \in \text{constraints}(\mathbb{I}) \wedge fv(c) \cap \bar{p} \neq \emptyset\} \\
 \bar{p}' &= \bar{p} \cup \{fv(c) \mid c \in \bar{c}'\}
 \end{aligned}$$

4.2.4 Sequence typing

Finally, Figure 6 shows the type rules for sequences, which are of the form $\Gamma \vdash \bar{s} : \bar{R}$, where given an environment Γ the sequence of statements \bar{s} has a set of return types \bar{R} . These return types are collected from all **return** statements in the sequence. This is used by the type system to verify whether the types of all return statements in a function are assignable to the declared return type.

All rules are default and identical to those in safeFTS, except for the type rules for **if** statements. As with latent predicates in occurrence typing [33], the type system uses the presence tests inside conditions of **if** statements to refine interface types in the branches. I-IfPresenceInterface shows the case where the condition contains a property presence test (cfr. Section 3.3) for a property of an object with an interface type.

The function *addConstraint* adds the constraints to the interface, and performs a satisfiability check to verify that there are no inconsistent constraints in the extended constraint

$$\begin{array}{c}
\text{I-EmpSeq} \frac{}{\Gamma \vdash \bullet : \bullet} \qquad \text{I-ExpSt} \frac{\Gamma \vdash e : S \quad \Gamma \vdash \bar{s} : \bar{R}}{\Gamma \vdash e; \bar{s} : \bar{R}} \\
\\
\text{I-IfPresenceInterface} \frac{\Gamma \vdash x : I \quad n : S \in \text{properties}(I) \quad \Gamma \vdash \bar{s} : \bar{R} \quad \begin{array}{l} I^- = \text{addConstraint}(I, \neg \text{present}(n)) \quad \Gamma \uplus x : I^- \vdash \bar{c}_1 : \bar{T}_1 \\ I^+ = \text{addConstraint}(I, \text{present}(n)) \quad \Gamma \uplus x : I^+ \vdash \bar{c}_2 : \bar{T}_2 \end{array}}{\Gamma \vdash \text{if } (x.n \equiv \text{undefined}) \{ \bar{c}_1 \} \text{ else } \{ \bar{c}_2 \}; \bar{s} : \bar{T}_1, \bar{T}_2, \bar{R}} \\
\\
\text{I-IfGeneral} \frac{\Gamma \vdash e : S \quad \Gamma \vdash \bar{c}_1 : \bar{T}_1 \quad \Gamma \vdash \bar{c}_2 : \bar{T}_2 \quad \Gamma \vdash \bar{s} : \bar{R}}{\Gamma \vdash \text{if } (e) \{ \bar{c}_1 \} \text{ else } \{ \bar{c}_2 \}; \bar{s} : \bar{T}_1, \bar{T}_2, \bar{R}} \qquad \text{I-Return} \frac{\Gamma \vdash \bar{s} : \bar{R}}{\Gamma \vdash \text{return}; \bar{s} : \text{void}, \bar{R}} \\
\\
\text{I-ReturnVal} \frac{\Gamma \vdash e : T \quad \Gamma \vdash \bar{s} : \bar{R}}{\Gamma \vdash \text{return } e; \bar{s} : T, \bar{R}} \\
\\
\text{I-ITVarDec} \frac{\Gamma \vdash e : T \quad T \leq S \quad \text{noDup}(\Gamma, x : S) \quad \Gamma \uplus x : S \vdash \bar{s} : \bar{R}}{\Gamma \vdash \text{var } x : S = e; \bar{s} : \bar{R}}
\end{array}$$

■ **Figure 6** Sequence type rules in $\text{TypeScript}_{\text{IPC}}$.

set. In the case of inconsistencies (ie. when the formula $\text{present}(n) \wedge \neg \text{present}(n)$ can be proven for any n), addConstraint will return the bottom type `Undefined`, preventing access to an invalid object. The definition of addConstraint is straightforward and omitted for lack of space. Note that the type assignment for e is *overwritten* in both branches using \uplus , leaving type assignments for other variables as-is. Although Figure 6 only defines rules for a single pattern of conditional expressions, the type rule can be generalised to inequalities and combined logical expressions, like in [33]. If statements without presence tests are covered by `I-IfGeneral`.

5 Operational Semantics of $\text{TypeScript}_{\text{IPC}}$

TypeScript is a superset of JavaScript that adds typing. However, after compilation, TypeScript emits JavaScript code in which all types are erased, which means that the semantics of TypeScript (and $\text{TypeScript}_{\text{IPC}}$) are the same of those of JavaScript. However, we provide the operational semantics of $\text{TypeScript}_{\text{IPC}}$, which will be used in Section 6 to prove its soundness.

A heap H is a partial function from locations (l) to heap objects (o). A heap object is either a closure or an object map. A closure represents a function, and is a pair containing a lambda expression (where $\text{function}(\bar{x})\{\bar{s}\}$ is shortened to $\lambda \bar{x}.\{\bar{s}\}$) and a scope chain L . An object map represents an object literal, and is a partial function from variables (x) to values (v). A variable is either a program variable x , a property name n or the internal properties `@this` or `@interface`. A value is a location l or a literal l . A result r is a value or a reference, and a reference is a pair containing a location and a variable.

An empty heap is indicated by `emp`, a heap cell by $l \mapsto o$, a heap lookup by $H(l, x)$, a heap update by $H[l \mapsto o]$ and the union of two disjoint heaps is indicated by $H_1 * H_2$. $H[l, x \mapsto v]$ updates or extends an object map l with the element x . $H(l, x) \downarrow$ is true iff $H(l, x)$ is defined. We define a helper function $\gamma(H, r)$ that returns r if r is a value, otherwise (i.e. r is a reference (l, x)) it returns $H(l, x)$ if defined and `undefined` otherwise. `null` is a distinguished location, and may not be in the domain of the heap.

The evaluation rules use a *scope chain* to model the treatment of variables in JavaScript: JavaScript resolves variables dynamically against a scope object. A scope chain is a list of locations of the scope objects, and $l : L$ is a concatenation of a location l to a scope chain L . A program is evaluated with a scope chain containing only the global JavaScript object l_g . For each function call, a new scope object is created and prepended to the beginning of the scope chain. After evaluating the function call, that scope object is removed from the scope chain. The variable lookup function σ is defined as follows:

$$\sigma(H, l : L, x) = \begin{cases} l & \text{if } H(l, x) \downarrow \\ \sigma(H, L, x) & \text{otherwise} \end{cases}$$

The evaluation of an expression e is written as follows: $\langle H_1, L, e \rangle \Downarrow \langle H_2, r \rangle$, with H_1 as initial heap and L as scope chain, evaluating to heap H_2 with result r . As we often need to evaluate expressions to values instead of references, we define $\langle H_1, L, e \rangle \Downarrow_v \langle H_2, v \rangle$ as the combination $\langle H_1, L, e \rangle \Downarrow \langle H_2, r \rangle$ and $\gamma(H_2, r) = v$.

Figure 7 shows the semantics for evaluating expressions in TypeScript_{IPC}. The evaluation rules of TypeScript_{IPC} are almost identical to those in safeFTS, but omit block scoping. E-ObLit uses an auxiliary function *new* to create a new location in the object map, E-Update uses the auxiliary function *clone* to duplicate an object, and E-Prop' uses the auxiliary function *box* to box primitive values. Note that we do not create bindings for all local variables up front: they are added to the local scope as they are declared and initialised. E-Update and E-TypeAssertInf are new. E-Update evaluates the functional update of multiple properties at once, and E-TypeAssertInf covers the casting of an object literal to an interface. Next to evaluating the object literal (as in E-ObLit), the internal property `@interface` indicates that the expression is of interface type I . In the next section, this property is used for linking the run-time interface in a location to the declared type in the program. In E-Call, the auxiliary functions *This* and *act* are used:

$$\text{This}(H, (l, x)) = \begin{cases} l & \text{if } H(l, @\text{this}) \downarrow \\ l_g & \text{otherwise} \end{cases}$$

$$\text{act}(l, \bar{x}, \bar{v}, l') = l \mapsto (\{\bar{x} \mapsto \bar{v}, @\text{this} \mapsto l'\})$$

The evaluation relation for statement sequences is written as $\langle H_1, L, \bar{s}_1 \rangle \Downarrow \langle H_2, s \rangle$, where s is a statement result (i.e. either `return;`, `return v;` or `;`). These rules are omitted for brevity. Unlike safeFTS, the branches of if statements introduce a new scope, so variables declared there are not visible outside.

6 Soundness

The novelty of the TypeScript_{IPC} type system lies in its guarantee that *all* constraints imposed on objects are guaranteed to be satisfied throughout the execution of the program, including those over multiple properties. This property is captured in Lemma 1.

Our proof of type soundness is structured identically to [7], albeit without support for block typing and contextual typing. We define a heap type Σ as a partial function from heap locations to types [3, 8] (either function types, object literal types, or interface types). Next, we introduce a number of judgments. First, we define a well-formedness judgment for heaps $H \models \diamond$ and a judgment that a heap H and scope chain L are compatible, written $H, L \models \diamond$. This judgment requires that all scope objects in the scope chain exist on the heap. We use a judgment $\Sigma \models H$ to denote that the heap H is compatible with the heap type Σ .

$$\begin{array}{c}
\text{E-Id} \frac{\sigma(H, L, \mathbf{x}) = l}{\langle H, L, \mathbf{x} \rangle \Downarrow \langle H, (l, \mathbf{x}) \rangle} \qquad \text{E-Lit} \frac{}{\langle H, L, \mathbf{1} \rangle \Downarrow \langle H, \mathbf{1} \rangle} \\
\text{E-this} \frac{\sigma(H, L, \text{@this}) = l_1 \quad H(l_1, \text{@this}) = l}{\langle H, L, \text{@this} \rangle \Downarrow \langle H, l \rangle} \qquad \text{E-Op} \frac{\langle H_0, L, \mathbf{e}_1 \rangle \Downarrow_v \langle H_1, \mathbf{1}_1 \rangle \quad \langle H_1, L, \mathbf{e}_2 \rangle \Downarrow_v \langle H_2, \mathbf{1}_2 \rangle}{\langle H_0, L, \mathbf{e}_1 \otimes \mathbf{e}_2 \rangle \Downarrow \langle H_2, \mathbf{1}_1 \otimes \mathbf{1}_2 \rangle} \\
\text{E-ObLit} \frac{H_1 = H_0 * [l \mapsto \text{new}()] \quad \langle H_1, L, \mathbf{e}_1 \rangle \Downarrow_v \langle H'_1, v_1 \rangle \quad H_2 = H'_1[(l, \mathbf{n}_1) \mapsto v_1] \quad \dots \quad \langle H_m, L, \mathbf{e}_m \rangle \Downarrow_v \langle H'_m, v_m \rangle \quad H = H'_m[(l, \mathbf{n}_m) \mapsto v_m]}{\langle H_0, L, \{\mathbf{n}_1 : \mathbf{e}_1, \dots, \mathbf{n}_m : \mathbf{e}_m\} \rangle \Downarrow \langle H, l \rangle} \\
\text{E-Assign} \frac{\langle H_0, L, \mathbf{e}_1 \rangle \Downarrow \langle H_1, (l, \mathbf{x}) \rangle \quad \langle H_1, L, \mathbf{e}_2 \rangle \Downarrow_v \langle H_2, v \rangle}{\langle H_0, L, \mathbf{e}_1 = \mathbf{e}_2 \rangle \Downarrow \langle H_2[(l, \mathbf{x}) \mapsto v], v \rangle} \\
\text{E-Update} \frac{\langle H_0, L, \mathbf{e} \rangle \Downarrow_v \langle H'_0, l \rangle \quad H_1 = H'_0 * [l_r \mapsto \text{clone}(l)] \quad \langle H_1, L, \mathbf{e}_1 \rangle \Downarrow_v \langle H'_1, v_1 \rangle \quad \dots \quad \langle H_m, L, \mathbf{e}_m \rangle \Downarrow_v \langle H'_m, v_m \rangle \quad H = H'_m[(l_r, \mathbf{n}_m) \mapsto v_m]}{\langle H_0, L, \text{assign}(\mathbf{e}, \{\mathbf{n}_1 : \mathbf{e}_1, \dots, \mathbf{n}_m : \mathbf{e}_m\}) \rangle \Downarrow \langle H, l_r \rangle} \\
\text{E-Prop} \frac{\langle H_0, L, \mathbf{e} \rangle \Downarrow_v \langle H_1, l \rangle \quad l \neq \text{null}}{\langle H_0, L, \mathbf{e}.n \rangle \Downarrow \langle H_1, (l, \mathbf{n}) \rangle} \qquad \text{E-Prop}' \frac{\langle H_0, L, \mathbf{e} \rangle \Downarrow_v \langle H_1, \mathbf{1} \rangle \quad H_2 = H_1 * [l \mapsto \text{box}(\mathbf{1})]}{\langle H_0, L, \mathbf{e}.n \rangle \Downarrow \langle H_2, (l, \mathbf{n}) \rangle} \\
\text{E-Call} \frac{\langle H_0, L_0, \mathbf{e} \rangle \Downarrow \langle H_1, r \rangle \quad H(l_1) = \langle \lambda \bar{x}. \{\bar{s}\}, L_1 \rangle \quad \langle H_1, L_0, \mathbf{e}_1 \rangle \Downarrow_v \langle H_2, v_1 \rangle \quad \dots \quad \langle H_n, L_0, \mathbf{e}_n \rangle \Downarrow_v \langle H_{n+1}, v_n \rangle \quad H' = H_{n+1} * \text{act}(l, \bar{x}, \bar{v}, l_2) \quad \langle H', l : L_1, \bar{s} \rangle \Downarrow \langle H'', \text{return } v; \rangle}{\langle H_0, L_0, \mathbf{e}(\mathbf{e}_1, \dots, \mathbf{e}_n) \rangle \Downarrow \langle H'', v \rangle} \\
\text{E-CallUndef} \frac{\langle H_0, L_0, \mathbf{e} \rangle \Downarrow \langle H_1, r \rangle \quad H(l_1) = \langle \lambda \bar{x}. \{\bar{s}\}, L_1 \rangle \quad \langle H_1, L_0, \mathbf{e}_1 \rangle \Downarrow_v \langle H_2, v_1 \rangle \quad \dots \quad \langle H_n, L_0, \mathbf{e}_n \rangle \Downarrow_v \langle H_{n+1}, v_n \rangle \quad H' = H_{n+1} * \text{act}(l, \bar{x}, \bar{v}, l_2) \quad \langle H', l : L_1, \bar{s} \rangle \Downarrow \langle H'', \text{return}; \rangle}{\langle H_0, L_0, \mathbf{e}(\mathbf{e}_1, \dots, \mathbf{e}_n) \rangle \Downarrow \langle H'', \text{undefined} \rangle} \\
\text{E-Func} \frac{H_1 = H_0 * [l \mapsto \langle \lambda \bar{x}. \{\bar{s}\}, L \rangle]}{\langle H_0, L, \text{function}(\bar{x})\{\bar{s}\} \rangle \Downarrow \langle H_1, l \rangle} \qquad \text{E-TypeAssert} \frac{\langle H_0, L, \mathbf{e} \rangle \Downarrow \langle H_1, r_1 \rangle}{\langle H_0, L, \langle T \rangle \mathbf{e} \rangle \Downarrow \langle H_1, r_1 \rangle} \\
\text{E-TypeAssertInf} \frac{H_1 = H_0 * [l \mapsto \{\text{@interface} \mapsto \mathbf{I}\}] \quad \langle H_1, L, \mathbf{e}_1 \rangle \Downarrow_v \langle H'_1, v_1 \rangle \quad H_2 = H'_1[(l, \mathbf{n}_1) \mapsto v_1] \quad \dots \quad \langle H_m, L, \mathbf{e}_m \rangle \Downarrow_v \langle H'_m, v_m \rangle \quad H = H'_m[(l, \mathbf{n}_m) \mapsto v_m]}{\langle H_0, L, \langle \mathbf{I} \rangle \{\bar{\mathbf{n}} : \bar{\mathbf{e}}\} \rangle \Downarrow \langle H, l \rangle}
\end{array}$$

■ **Figure 7** Operational semantics of TypeScript_{IPC}.

This compatibility also requires that the constraints of interface types are satisfied, which we prove in Lemma 2. Finally, we depend on a function $context(\Sigma, L)$ which builds a typing judgment describing the variables in the scope chain L , using the types in Σ . The \uplus operator ensures that only the inner-most type for a variable is used: if a variable is present on both sides, the right instance is returned. Because E-TypeAssertInf attaches an `@interface` label to all interface variables in the heap, Σ can reconstruct interface types as well as function types and object literal types.

$$\begin{aligned} context(\Sigma, []) &= \{\} \\ context(\Sigma, l : L) &= context(\Sigma, L) \uplus \Sigma(l) \end{aligned}$$

We combine the judgments above to write $\Sigma \models \langle H, L, e \rangle : T$ to mean $\Sigma \models H$; $H, L \models \diamond$; and $context(\Sigma, L) \vdash e : T$. We define an analogous judgment for statements, as $\Sigma \models \langle H, L, \bar{s} \rangle : \bar{T}$. Finally, we add a judgment on the result of evaluation of expressions, written $\Sigma \models \langle H, r \rangle : T$.

Before we can prove the safety properties of our type system with respect to evaluation, we first show that the constraints of an interface type accurately predict the presence or absence of its properties at runtime.

► **Lemma 1** (Constraint–presence correlation). *The type system of $TypeScript_{IPC}$ guarantees that if the constraints of an interface contain a constraint $present(n)$, it is certain that the property n is present at runtime in objects with that interface type. Similarly: if there is a constraint $not(present(n))$, it is certain that the property n will not be present.*

Proof. There are three cases to consider:

Case 1: Construction Interfaces can only be constructed in three ways, which all ensure that the correlation holds:

Case 1a: I-AssertInf. When an object literal is cast to an interface, the interface constraints are verified against the properties in the object literal. The correlation is thus informed by the exact properties of the runtime object (E-TypeAssertInf) and enforced by the type system.

Case 1b: I-Assign. When an instance of interface I_0 is assigned to a variable of type interface I_1 , the type system requires that the constraints are satisfied via the assignment compatibility rule A-Interface. The correlation holds for the source object (with type I_0) and the compatibility rule asserts that the properties of I_1 must be respectively present or absent. Therefore, the correlation must hold after the cast as well. At runtime, nothing changes.

Case 1c: I-Assert. Analogous to Case 1b: assignment compatibility dictates the presence and absence of properties in the source object. Nothing changes at runtime.

Case 2: Property assignment The assignment of new values to object properties either happens on a per-property basis (Case 2a), or multiple properties at once using `assign` (Case 2b).

Case 2a: I-Assign. When a new value is assigned to a property n of an interface, two typing rules are relevant: I-Prop (including the *lookup* function) and I-Assign. At runtime, the E-Assign rule simply overwrites the object property, so it is up to the type system to enforce the correlation. We assume the correlation holds before the assignment, so the constraints of the interface must state one of the following:

present(n): the *lookup* function of I-Prop returns the type of n and I-Assign then allows the assignment of another value (following the typing rules). As this will

only update the value of a property that is already present, this does not change the presence of n in the object, thus the correlation holds.

\neg **present**(n): the *lookup* function of I-Prop returns type **Undefined**. The assignment compatibility required by I-Assign will fail as no type is assignable to **Undefined**, except for **undefined**, in which case the property will remain absent. Again, the correlation holds.

Neither: the *lookup* function of I-Prop is not defined in this case, so the program does not typecheck. Without this safety guard in place, the correlation would not hold.

Case 2b: I-Update. The **assign** function updates multiple properties of an object. Again, we assume that the correlation holds before the assignment. The **assign** function returns a new object, of the same type as the first argument, in which the properties of the second argument are updated. Properties can become absent or present (by resp. assigning **undefined** or a value different from **undefined**), or simply change value. The assignment is only accepted by the type checker if the second argument of **assign** is assignable to the generated interface which covers its properties. Therefore, a change in presence for those properties is only allowed if the input interface did not already require their presence or absence. At runtime, rule E-Update first clones the object and then the properties are overwritten by those of the second argument. The correlation holds for both the generated interface (because of assignment compatibility and isolation) and the rest of the object.

Case 3: After a presence test In case of an if statement that tests the presence of an interface property, the newly gained information is added to the constraints of the type in both branches (function *addConstraint* in I-IfPresenceInterface). Here the property follows from the program flow: if the field presence test succeeds the type system can only conclude that the **present** constraint applies, and vice versa when the presence test fails. Outside of the if statement, the **present** constraint is discarded again. Even though the runtime value does not change, this is again an example of the properties of the runtime value informing the the type system and thus the correlation. \blacktriangleleft

From Lemma 1, we can prove that a well-typed program does not violate constraints at runtime. We add an additional condition to the heap–heap type compatibility rule stated above as $\Sigma \models H$: (the *fields* function returns field names of an object at runtime)

\blacktriangleright **Lemma 2** (Correctness of interface types at runtime). *For heap locations tagged as interface types, i.e. those where $\Sigma(l) = I$, the following is required:*

1. Every interface object is tagged as such:
 $H(l, @interface) = I' \wedge I' \leq I$;
2. All properties are correctly typed:
 $\forall n \in fields(l) : n : T \in properties(I) \wedge H, \Sigma \vdash (l, n) : T' \wedge T' \leq T$.
3. The constraints are satisfied by a valuation over the presence or absence of properties:
 $v = c_p \cup c_{np}$ and $\hat{v}(constraints(I)) = true$

$$where\ c_p = \{present(n) \mid n \in fields(l)\}$$

$$where\ c_{np} = \{\neg present(n) \mid n \in properties(I)$$

$$\wedge (\neg H(l, n) \downarrow \vee H(l, n) = undefined)\}$$

$$where\ fields(l) = \{n \mid H(l, n) \downarrow \wedge n \neq @interface \wedge H(l, n) \neq undefined\}$$

This lemma is not only unaffected by explicit property presence tests, it guarantees it because of property 3. Assuming an object (with interface type I) is well-formed before the presence

test, then the strengthened interface type I' in the taken branch must more closely resemble the state of the runtime object.

Proof. By induction on the evaluation rules. Most rules do not directly modify the heap, so we only focus on the rules that potentially invalidate this condition.

E-TypeAssertInf This evaluation rule is responsible for instantiating interface types on the heap, given an object literal. Property 1 follows from the evaluation rule. Properties 2 and 3 follow directly from the type system.

E-Assign There are three sub-cases: e_1 can either resolve to a variable reference, an object property, or an interface property:

- In case of a variable reference to an interface I , the three properties follow directly from assignment compatibility between I and the interface type I' assigned to e_2 .
- In case of a property belonging to an object: the three properties cannot be invalidated.
- In case of an interface property: it depends on whether this expression is trying to add a new property or update a present property. The type system assigns type `Undefined` to properties which are guaranteed to be absent, and rejects programs that access properties whose presence is unknown.

For property update, we prevent users from modifying the `@interface` property (preserving property 1). Properties 2 and 3 are guaranteed by assignment compatibility.

E-Update This rule first clones the source object (for which all properties are already satisfied) before assigning the new fields. Property 1 follows from the evaluation rule: the `@interface` tag is cloned along with other fields. We now consider the generated interface I' in `I-UpdateInf`. *slice* ensures that the interface contains the smallest possible subset of constraints and properties such that all constraints in I either do not mention any properties from I' or are part of the constraints in I' . For the fields in I' , the properties 2 and 3 are guaranteed by the `I-UpdateInf` rule. For fields *not* in I' , properties 2 and 3 continue to hold, as they cannot be affected by the `assign` operation by definition.

E-ObLit This rule creates a new object on the heap, but cannot invalidate existing interface types on the heap.

E-Prop', E-Func These rules create a heap location for respectively properties of literal objects and a closure, but neither can affect existing interface types on the heap.

E-Call, E-CallUndef The heap modifications made by these two rules are limited to evaluation of sub-expressions or the allocation of a new scope object to hold the new function's local variables. In the latter case, we rely on the fact that extension cannot affect existing interface types on the heap. ◀

Finally, we can combine Lemma 2 with the existing proof of `safeFTS` to obtain proof of type safety in the presence of constraints.

► **Theorem 3** (Subject reduction).

- If $\Sigma \models \langle H, L, e \rangle : T$ and $\langle H, L, e \rangle \Downarrow \langle H', r \rangle$
then $\exists \Sigma', T'$ such that $\Sigma \subseteq \Sigma', \Sigma' \models \langle H', r \rangle : T'$ and $T' \leq T$.
- If $\Sigma \models \langle H, L, \bar{s} \rangle : \bar{T}$ and $\langle H, L, \bar{s} \rangle \Downarrow \langle H', s \rangle$
then $\exists \Sigma', T'$ such that $\Sigma \subseteq \Sigma', \Sigma' \models \langle H', s \rangle : T'$ and $T' \leq \text{return}(\bar{T})$.

7 Related Work

To the best of our knowledge, `TypeScriptIPC` is the first language that statically verifies *all* aspects of programming with inter-property constraints: defining, initialising, accessing *and*

updating objects with inter-property constraints. In this section, we give an overview of existing work related to various aspects of the type system presented in this paper.

Dependent and refinement types

Dependently typed languages [5, 36] allow programmers to write more expressive types, by parametrising types on values. There are no restrictions on what dependent types can express, which comes at the cost of decidability. Refinement types are a restricted form of dependent types where types are “refined” with predicates that are statically decidable, for example through SMT solvers. Refinement types have been used to verify many different properties [35, 14, 29, 23, 6, 11, 34]. We limit our discussion of refinement types to the applications that are close to our work: refinement types for dynamic programming languages and object-oriented programming languages.

DJS [11] extends a subset of JavaScript with dependent types, which allows (with some modifications) the expression of inter-property constraints over object properties. However, DJS requires extensive knowledge on heap typing from the developer. This significant annotation overhead is acknowledged in the paper. Contrast this to TypeScript_{IPC}, which proposes a lightweight extension to regular TypeScript interfaces.

In [34], Vekris et al. introduce RSC, a lightweight refinement system for TypeScript. RSC allows invariants to be imposed in classes and objects, including inter-property constraints on properties. However, the soundness of these invariants is guaranteed by restricting invariants to be imposed on *immutable* properties. Flanagan et al. introduce Hoop [13], a hybrid object-oriented programming language with refinement types and object invariants. Hoop requires refinements and variants to be pure and therefore refinements can only be placed on immutable data. In [23], Nystrom et al. introduce a form of dependent types for objects in X10. Again, constraints can only be imposed on immutable fields. To conclude, although refinement type systems are often able to express inter-property constraints, none of them support inter-property constraints after the initialisation phase: updating properties that are part of inter-property constraints is impossible. In contrast, TypeScript_{IPC} allows single-property updates of objects, *and* guarantees that the constraints remain satisfied.

Type refinements

The type system of TypeScript_{IPC} extracts property presence information from conditional expressions. This concept is known as occurrence typing [32, 33] or type refinement, which narrows (or *strengthens*) variable types based on predicates in conditional expressions. Several static type systems for dynamic languages such as TypeScript [2], Hack [1], Flow [10], λ_S [17] and [20] support refining types using tests on the type of a value. Recently, a hybrid occurrence-refinement type system was proposed in [21]. As this paper demonstrates, occurrence typing can also be applied to objects with inter-property constraints.

Constraint-based programming

The constraint-centric interfaces introduced in this paper should not be confused with constraint-based programming [30]. Constraint-based programming is a discipline that finds solutions for a number of variables given constraints over these variables. By contrast, TypeScript_{IPC} uses constraints and flow information to determine the most specific presence information for properties of objects.

Type systems for dynamic languages

In recent years, several formalisations for TypeScript have been proposed. As already mentioned earlier, TypeScript_{IPC} is based on earlier work [7] by Bierman et al., who formalised both sound and unsound features of TypeScript, including features such as contextual typing and the lack of block scoping in JavaScript. There exist several other approaches for adding gradual typing to dynamic languages such as TypeScript [27, 28] and Dart [19]. These approaches focus on improving the combination between sound and unsound parts of type systems for dynamic languages, which is orthogonal to the goal of our paper: enabling programmers to express inter-property constraints and statically enforcing them.

There already exist several research efforts that focus on the dynamic nature of objects in JavaScript [4, 31, 18, 9], providing a static type system that verifies the usage of objects, such as property additions, accesses and updates. The focus of this paper is not on supporting JavaScript’s object types, but on extending object types with inter-property constraints. Accessing and updating object properties with inter-property constraints is allowed, but only when it does not invalidate the object constraints.

Optional object properties

TypeScript_{IPC} is not the first language to impose constraints on the presence of an object property. In TypeScript, objects (and methods) can contain optional properties (and parameters). In strict null checking mode, the type of an optional property in TypeScript is automatically transformed to a union type, combining the original type with `Undefined`. Similarly, programmers can only assign `null` to value types in C# if that type is indicated as a nullable type. To support the notion of required and optional properties in Java, there also exist Java frameworks that provide support for `@NonNull` annotations (such as [12, 25]). However, all of these languages and frameworks are restricted to single-property constraints (types and presence) and cannot express inter-property constraints.

8 Future Work

This paper introduces the concept of constraints in programming languages. Going forward, we would like to further expand the expressiveness of constraint-centric interfaces. So far, TypeScript_{IPC} only supports inter-property constraints on the presence of properties. In the future, we plan to add support for *value-dependent constraints*, where the presence of a property depends on the value of another property. The introduction already listed an example of a value-dependent constraint in the Chart.js library: “If the `steppedLine` value is set to anything other than `false`, `lineTension` will be ignored”. Another example can be found in the Google Maps API for rendering directions⁸, where “the `infoWindow` property is ignored when the property `suppressInfoWindows` is set to `true`”. To enable value-dependent constraints, we plan on using TypeScript’s *literal types* that limit types to a set of predefined values.

In this paper we only considered constraints as applied to interfaces, but constraints could also be imposed on the parameters of a function definition. Listing 10 shows the (simplified) function `utime` from the Python standard library, which imposes a NAND constraint on two of its parameters.

⁸ <https://developers.google.com/maps/documentation/javascript/reference/3/directions>

```
1 function utime(path: string, times: array, ns: array) {
2   //...
3 } constraining {
4   present(path);
5   ¬(present(ns) ^ present(times));
6 }
```

■ **Listing 10** Hypothetical example of a function with inter-parameter constraints.

Finally, this paper highlighted the need for updating multiple properties at once. In the future, we plan on updating multiple object properties in place without increasing the annotation burden, by means of alias tracking or stronger heap types.

9 Conclusion

This paper shows how complex constraints on the presence of interface properties can be statically enforced in programming languages. We introduced a type system with *constraint-centric interfaces*, which express constraints on the presence of properties in the desired pattern.

To achieve this, the type system is extended with four new features: 1) Interfaces carry constraints on their properties; 2) The type system uses if statements to enrich variable types of interfaces used in the condition with extra information about property presence; 3) Accessing and updating a property of an object is only allowed when the constraints can statically guarantee its presence; 4) Finally, a novel procedure `assign` allows the (functional) updating of multiple properties at once, which is necessary to safely update properties that are part of an inter-property constraint.

Implementation. The implementation of TypeScript_{IPC} is available at <https://github.com/noostvog/TypeScriptIPC>.

References

- 1 Hack. <http://hacklang.org/>.
- 2 TypeScript 2.0 Release Notes. <https://www.typescriptlang.org/docs/handbook/release-notes/typescript-2-0.html>.
- 3 Martin Abadi and Luca Cardelli. *A theory of objects*. Springer Science & Business Media, 2012.
- 4 Christopher Anderson, Paola Giannini, and Sophia Drossopoulou. Towards type inference for JavaScript. In *European conference on Object-oriented programming*, pages 428–452. Springer, 2005.
- 5 Lennart Augustsson. Cayenne: a language with dependent types. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming, ICFP '98*, pages 239–250, New York, NY, USA, 1998. ACM.
- 6 Jesper Bengtson, Karthikeyan Bhargavan, Cédric Fournet, Andrew D Gordon, and Sergio Maffeis. Refinement types for secure implementations. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 33(2):8, 2011.
- 7 Gavin Bierman, Martín Abadi, and Mads Torgersen. Understanding TypeScript. In *European Conference on Object-Oriented Programming*, pages 257–281. Springer, 2014.
- 8 Gavin M Bierman, MJ Parkinson, and AM Pitts. MJ: An imperative core calculus for Java and Java with effects. Technical report, University of Cambridge, Computer Laboratory, 2003.


- 9 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*, pages 410–429. ACM, 2016.
- 10 Avik Chaudhuri, Panagiotis Vekris, Sam Goldman, Marshall Roch, and Gabriel Levi. Fast and precise type checking for JavaScript. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):48, 2017.
- 11 Ravi Chugh, David Herman, and Ranjit Jhala. Dependent Types for JavaScript. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 587–606, New York, NY, USA, 2012. ACM.
- 12 Manuel Fähndrich and K. Rustan M. Leino. Declaring and Checking Non-null Types in an Object-oriented Language. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '03, pages 302–312, New York, NY, USA, 2003. ACM.
- 13 Cormac Flanagan, Stephen N Freund, and Aaron Tomb. Hybrid types, invariants, and refinements for imperative objects. *FOOL/WOOD*, 6, 2006.
- 14 Tim Freeman and Frank Pfenning. Refinement Types for ML. In *In Proceedings of the SIGPLAN'91 Symposium on Language Design and Implementation*. Citeseer, 1991.
- 15 Jean H Gallier. *Logic for computer science: foundations of automatic theorem proving*. Courier Dover Publications, 2015.
- 16 Jean-Yves Girard. Linear logic. *Theoretical computer science*, 50(1):1–101, 1987.
- 17 Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. Typing Local Control and State Using Flow Analysis. In *European Symposium on Programming*, pages 256–275. Springer, 2011.
- 18 Phillip Heidegger and Peter Thiemann. Recency types for analyzing scripting languages. In *European conference on Object-oriented programming*, pages 200–224. Springer, 2010.
- 19 Thomas S Heinze, Anders Møller, and Fabio Strocchio. Type safety analysis for Dart. In *Proceedings of the 12th Symposium on Dynamic Languages*, pages 1–12. ACM, 2016.
- 20 Vineeth Kashyap, John Sarracino, John Wagner, Ben Wiedermann, and Ben Hardekopf. Type Refinement for Static Analysis of JavaScript. In *Proceedings of the 9th Symposium on Dynamic Languages*, DLS '13, pages 17–26, New York, NY, USA, 2013. ACM.
- 21 Andrew M. Kent, David Kempe, and Sam Tobin-Hochstadt. Occurrence Typing Modulo Theories. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 296–309, New York, NY, USA, 2016. ACM.
- 22 Per Martin-Löf and Giovanni Sambin. *Intuitionistic type theory*, volume 9. Bibliopolis Napoli, 1984.
- 23 Nathaniel Nystrom, Vijay Saraswat, Jens Palsberg, and Christian Grothoff. Constrained types for object-oriented languages. In *In OOPSLA'08: Proceedings of the 23rd ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*. Citeseer, 2008.
- 24 Nathalie Oostvogels, Joeri De Koster, and Wolfgang De Meuter. Inter-parameter Constraints in Contemporary Web APIs. In *International Conference on Web Engineering*, pages 323–335. Springer, 2017.
- 25 Matthew M Papi, Mahmood Ali, Telmo Luis Correa Jr, Jeff H Perkins, and Michael D Ernst. Practical pluggable types for Java. In *Proceedings of the 2008 international symposium on Software testing and analysis*, pages 201–212. ACM, 2008.
- 26 Garrel Pottinger. A type assignment for the strongly normalizable λ -terms. *To HB Curry: essays on combinatory logic, lambda calculus and formalism*, pages 561–577, 1980.

- 27 Aseem Rastogi, Nikhil Swamy, Cédric Fournet, Gavin Bierman, and Panagiotis Vekris. Safe & Efficient Gradual Typing for TypeScript. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 167–180, New York, NY, USA, 2015. ACM.
- 28 Gregor Richards, Francesco Zappa Nardelli, and Jan Vitek. Concrete types for TypeScript. In *LIPICs-Leibniz International Proceedings in Informatics*, volume 37. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.
- 29 Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. Liquid types. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 159–169, New York, NY, USA, 2008. ACM.
- 30 Francesca Rossi, Peter Van Beek, and Toby Walsh. *Handbook of constraint programming*. Elsevier, 2006.
- 31 Peter Thiemann. Towards a type system for analyzing javascript programs. In *European Symposium On Programming*, pages 408–422. Springer, 2005.
- 32 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.
- 33 Sam Tobin-Hochstadt and Matthias Felleisen. Logical Types for Untyped Languages. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, pages 117–128, New York, NY, USA, 2010. ACM.
- 34 Panagiotis Vekris, Benjamin Cosman, and Ranjit Jhala. Refinement Types for TypeScript. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 310–325, New York, NY, USA, 2016. ACM.
- 35 Hongwei Xi and Frank Pfenning. Eliminating Array Bound Checking Through Dependent Types. In *In Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1998.
- 36 Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 214–227. ACM, 1999.

Mailbox Types for Unordered Interactions


Ugo de'Liguoro

Università di Torino, Dipartimento di Informatica, Torino, Italy
deligu@di.unito.it

 <https://orcid.org/0000-0003-4609-2783>

Luca Padovani

Università di Torino, Dipartimento di Informatica, Torino, Italy
luca.padovani@unito.it

 <https://orcid.org/0000-0001-9097-1297>

Abstract

We propose a type system for reasoning on protocol conformance and deadlock freedom in networks of processes that communicate through unordered mailboxes. We model these networks in the mailbox calculus, a mild extension of the asynchronous π -calculus with first-class mailboxes and selective input. The calculus subsumes the actor model and allows us to analyze networks with dynamic topologies and varying number of processes possibly mixing different concurrency abstractions. Well-typed processes are deadlock free and never fail because of unexpected messages. For a non-trivial class of them, junk freedom is also guaranteed. We illustrate the expressiveness of the calculus and of the type system by encoding instances of non-uniform, concurrent objects, binary sessions extended with joins and forks, and some known actor benchmarks.

2012 ACM Subject Classification Theory of computation \rightarrow Type structures, Theory of computation \rightarrow Process calculi, Software and its engineering \rightarrow Concurrent programming structures, Software and its engineering \rightarrow Message passing

Keywords and phrases actors, concurrent objects, first-class mailboxes, unordered communication protocols, behavioral types, protocol conformance, deadlock freedom, junk freedom

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2018.15

Related Version Proofs and additional examples are found in the related technical report [17], <https://arxiv.org/abs/1801.04167>.

Acknowledgements The authors are grateful to the anonymous reviewers for their valuable feedback. The second author misses his beloved cat Arturo, who passed away on January 8, 2018. Much of this paper was written next to Arturo during the last weeks of his long illness.

1 Introduction

Message passing is a key mechanism used to coordinate concurrent processes. The order in which a process consumes messages may coincide with the order in which they arrive at destination (*ordered processing*) or may depend on some intrinsic property of the messages themselves, such as their priority, their tag, or the shape of their content (*out-of-order or selective processing*). Ordered message processing is common in networks of processes connected by point-to-point *channels*. Out-of-order message processing is common in networks of processes using *mailboxes*, into which processes concurrently store messages and from which one process selectively receives messages. This communication model is typically found in the various implementations of actors [26, 1] such as Erlang [3], Scala and Akka actors [24], CAF [8] and Kilim [52]. Non-uniform, concurrent objects [50, 48, 15] are also



© Ugo de'Liguoro and Luca Padovani;

licensed under Creative Commons License CC-BY

32nd European Conference on Object-Oriented Programming (ECOOP 2018).

Editor: Todd Millstein; Article No. 15; pp. 15:1–15:28

Leibniz International Proceedings in Informatics



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

15:2 Mailbox Types for Unordered Interactions

```
1 class Account(var balance: Double) extends ScalaActor[AnyRef] {
2   private val self = this
3   override def process(msg: AnyRef) {
4     msg match {
5       case dm: DebitMessage =>
6         balance += dm.amount
7         val sender = dm.sender.asInstanceOf[Account]
8         sender.send(ReplyMessage.ONLY)
9       case cm: CreditMessage =>
10        balance -= cm.amount
11        val sender = cm.sender.asInstanceOf[ScalaActor[AnyRef]]
12        val recipient = cm.recipient.asInstanceOf[Account]
13        recipient.send(new DebitMessage(self, cm.amount))
14        receive {
15          case rm: ReplyMessage =>
16            sender.send(ReplyMessage.ONLY)
17        }
18        case _: StopMessage => exit()
19        case message =>
20          val ex = new IllegalArgumentException("Unsupported message")
21          ex.printStackTrace(System.err)
22    }
23  }
24 }
```

■ **Listing 1** An example of Scala actor taken from the *Savina* benchmark suite [32].

examples of out-of-order message processors. For example, a busy lock postpones the processing of any `acquire` message until it is released by its current owner. Out-of-order message processing adds further complexity to the challenging task of concurrent and parallel application development: storing a message into the wrong mailbox or at the wrong time, forgetting a message in a mailbox, or relying on the presence of a particular message that is not guaranteed to be found in a mailbox are programming mistakes that are easy to do and hard to detect without adequate support from the language and its development tools.

The Scala actor in Listing 1, taken from the *Savina* benchmark suite [32], allows us to illustrate some of the subtle pitfalls that programmers must carefully avoid when dealing with out-of-order message processing. The `process` method matches messages found in the actor's mailbox according to their type. If a message of type `DebitMessage` is found, then `balance` is incremented by the deposited amount and the actor requesting the operation is notified with a `ReplyMessage` (lines 5–8). If a message of type `CreditMessage` is found, `balance` is decremented by the amount that is transferred to `recipient` (lines 9–13). Since the operation is meant to be atomic, the actor temporarily changes its behavior and waits for a `ReplyMessage` from `recipient` signalling that the transfer is complete, before notifying `sender` in turn (lines 14–17). A message of type `StopMessage` terminates the actor (line 18).

Note how the correct execution of this code depends on some key assumptions:

- `ReplyMessage` should be stored in the actor's mailbox only when the actor is involved in a transaction, or else the message would trigger the “catch all” clause that throws a “unsupported message” exception (lines 19–21).
- No debit or credit message should be in the actor's mailbox by the time it receives `StopMessage`, or else some critical operations affecting the balance would not be performed.

- Two distinct accounts should not try to simultaneously initiate a transaction with each other. If this were allowed, each account could consume the credit message found in its own mailbox and then deadlock waiting for a reply from the other account (lines 14–17).

Static analysis techniques that certify the validity of assumptions like these can be valuable for developers. For example, session types [29] have proved to be an effective formalism for the enforcement of communication protocols and have been applied to a variety of programming paradigms and languages [2], including those based on mailbox communications [41, 7, 19, 43]. However, session types are specifically designed to address point-to-point, ordered interactions over channels [27]. Retrofitting them to a substantially different communication model calls for some inevitable compromises on the network topologies that can be addressed and forces programmers to give up some of the flexibility offered by unordered message processing.

Another aspect that complicates the analysis of actor systems is that the pure actor model as it has been originally conceived [26, 1] does not accurately reflect the actual practice of actor programming. In the pure actor model, each actor owns a single mailbox and the only synchronization mechanism is message reception from such mailbox. However, it is a known fact that the implementation of complex coordination protocols in the pure actor model is challenging [54, 53, 33, 9]. These difficulties have led programmers to mix the actor model with different concurrency abstractions [31, 53], to extend actors with controlled forms of synchronization [54] and to consider actors with multiple/first-class mailboxes [23, 33, 9]. In fact, popular implementations of the actor model feature disguised instances of multiple/first-class mailbox usage, even if they are not explicitly presented as such: in Akka, the messages that an actor is unable to process immediately can be temporarily stashed into a different mailbox [23]; in Erlang, hot code swapping implies transferring at runtime the input capability on a mailbox from a piece of code to a different one [3].

In summary, there is still a considerable gap between the scope of available approaches used to analyze mailbox-based communicating systems and the array of features used in programming these systems. To help narrowing this gap, we make the following contributions:

- We introduce *mailbox types*, a new kind of behavioral types with a simple and intuitive semantics embodying the unordered nature of mailboxes. Mailbox types allow us to describe mailboxes subject to selective message processing as well as mailboxes concurrently accessed by several processes. Incidentally, mailbox types also provide precise information on the size of mailboxes that may lead to valuable code optimizations.
- We develop a mailbox type system for the *mailbox calculus*, a mild extension of the asynchronous π -calculus [51] featuring tagged messages, selective inputs and first-class mailboxes. The mailbox calculus allows us to address a broad range of systems with dynamic topology and varying number of processes possibly using a mixture of concurrency models (including multi-mailbox actors) and abstractions (such as locks and futures).
- We prove three main properties of well-typed processes: the absence of failures due to unexpected messages (*mailbox conformance*); the absence of pending activities and messages in irreducible processes (*deadlock freedom*); for a non-trivial class of processes, the guarantee that every message can be eventually consumed (*junk freedom*).
- We illustrate the expressiveness of mailbox types by presenting well-typed encodings of known concurrent objects (locks and futures) and actor benchmarks (atomic transactions and master-workers parallelism) and of binary sessions extended with forks and joins. In discussing these examples, we emphasize the impact of out-of-order message processing and of first-class mailboxes.

Structure of the paper. We start from the definition of the *mailbox calculus* and of the properties we expect from well-typed processes (Section 2). We introduce mailbox types (Section 3.1) and dependency graphs (Section 3.2) for tracking mailbox dependencies in processes that use more than one. Then, we present the typing rules (Section 3.3) and the soundness results of the type system (Section 3.4). In the latter part of the paper, we discuss a few more complex examples (Section 4), related work (Section 5) and ideas for further developments (Section 6).

2 The Mailbox Calculus

We assume given an infinite set of *variables* x, y , an infinite set of *mailbox names* a, b , a set of *tags* \mathbf{m} and a finite set of *process variables* \mathbf{X} . We let u, v range over variables and mailbox names without distinction. Throughout the paper we write \bar{e} for possibly empty sequences e_1, \dots, e_n of various entities. For example, \bar{u} stands for a sequence u_1, \dots, u_n of names and $\{\bar{u}\}$ for the corresponding set.

The syntax of the *mailbox calculus* is shown below:

Process $P, Q ::= \mathbf{done} \mid u!\mathbf{m}[\bar{v}] \mid G \mid P \mid Q \mid (\nu a)P \mid \mathbf{X}[\bar{u}]$
Guard $G, H ::= \mathbf{fail} \ u \mid \mathbf{free} \ u.P \mid u?\mathbf{m}(\bar{x}).P \mid G + H$

The term **done** represents the terminated process that performs no action. The term $u!\mathbf{m}[\bar{v}]$ represents a message stored in mailbox u . The message has tag \mathbf{m} and arguments \bar{v} . A guarded process G is a composition of actions offered on a mailbox. Actions will be described in a moment. We assume that all actions in the same guard refer to the same mailbox u . The term $P \mid Q$ represents the parallel composition of P and Q and $(\nu a)P$ represents a restricted mailbox a with scope P . The term $\mathbf{X}[\bar{u}]$ represents the invocation of the process named \mathbf{X} with parameters \bar{u} . For each process variable \mathbf{X} we assume that there is a corresponding *global process definition* of the form $\mathbf{X}(\bar{x}) \triangleq P$. The action **fail** u represents the process that fails with an error for having received an unexpected message from mailbox u . The action **free** $u.P$ represents the process that deletes the mailbox u if u is empty and then continues as P . The action $u?\mathbf{m}(\bar{x}).P$ represents the process that receives an \mathbf{m} -tagged message from mailbox u and then continues as P with \bar{x} replaced by the message's arguments. A compound guard $G + H$ offers all the actions offered by G and H . The notions of free and bound names of a process P are standard and respectively denoted by $\mathbf{fn}(P)$ and $\mathbf{bn}(P)$.

The operational semantics of the mailbox calculus is mostly conventional. We use the structural congruence relation \equiv defined below to rearrange equivalent processes:

$$\begin{array}{lll} \mathbf{fail} \ a + G \equiv G & G + H \equiv H + G & G + (H + H') \equiv (G + H) + H' \\ \mathbf{done} \mid P \equiv P & P \mid Q \equiv Q \mid P & P \mid (Q \mid R) \equiv (P \mid Q) \mid R \\ & (\nu a)(\nu b)P \equiv (\nu b)(\nu a)P & (\nu a)P \mid Q \equiv (\nu a)(P \mid Q) \quad \text{if } a \notin \mathbf{fn}(Q) \end{array}$$

Structural congruence captures the usual commutativity and associativity laws of guard and process composition, with **fail** and **done** acting as the respective units. Additionally, the order of mailbox restrictions is irrelevant and the scope of a mailbox may shrink or extend dynamically. The reduction relation \rightarrow is inductively defined by the rules

$$\begin{array}{ll} \text{[R-READ]} & a!\mathbf{m}[\bar{c}] \mid a?\mathbf{m}(\bar{x}).P + G \rightarrow P\{\bar{c}/\bar{x}\} \\ \text{[R-FREE]} & (\nu a)(\mathbf{free} \ a.P + G) \rightarrow P \\ \text{[R-DEF]} & \mathbf{X}[\bar{c}] \rightarrow P\{\bar{c}/\bar{x}\} \quad \text{if } \mathbf{X}(\bar{x}) \triangleq P \\ \text{[R-PAR]} & P \mid R \rightarrow Q \mid R \quad \text{if } P \rightarrow Q \\ \text{[R-NEW]} & (\nu a)P \rightarrow (\nu a)Q \quad \text{if } P \rightarrow Q \\ \text{[R-STRUCT]} & P \rightarrow Q \quad \text{if } P \equiv P' \rightarrow Q' \equiv Q \end{array}$$

where $P\{\bar{c}/\bar{x}\}$ denotes the usual capture-avoiding replacement of the variables \bar{x} with the mailbox names \bar{c} . Rule $_{[R-READ]}$ models the selective reception of an **m**-tagged message from mailbox a , which erases all the other actions of the guard. Rule $_{[R-FREE]}$ is triggered when the process is ready to delete the empty mailbox a and no more messages can be stored in a because there are no other processes in the scope of a . Rule $_{[R-DEF]}$ models a process invocation by replacing the process variable X with the corresponding definition. Finally, rules $_{[R-PAR]}$, $_{[R-NEW]}$ and $_{[R-STRUCT]}$ close reductions under parallel compositions, name restrictions and structural congruence. We write \rightarrow^* for the reflexive and transitive closure of \rightarrow , we write $P \rightarrow^* Q$ if not $P \rightarrow^* Q$ and $P \rightarrow$ if $P \rightarrow Q$ for all Q .

Hereafter, we will occasionally use numbers and conditionals in processes. These and other features can be either encoded or added to the calculus without difficulties.

► **Example 1 (lock)**. In this example we model a lock as a process that waits for messages from a *self* mailbox in which acquisition and release requests are stored. The lock is either free or busy. When in state free, the lock nondeterministically consumes an **acquire** message from *self*. This message indicates the willingness to acquire the lock by another process and carries a reference to a mailbox into which the lock stores a **reply** notification. When in state busy, the lock waits for a **release** message indicating that it is being released:

$$\begin{aligned} \text{FreeLock}(self) &\triangleq \text{free } self.\text{done} \\ &\quad + self?\text{acquire}(owner).\text{BusyLock}[self, owner] \\ &\quad + self?\text{release.fail } self \\ \text{BusyLock}(self, owner) &\triangleq owner!\text{reply}[self] \mid self?\text{release}.\text{FreeLock}[self] \end{aligned}$$

Note the presence of the **free** *self* guard in the definition of **FreeLock** and the lack thereof in **BusyLock**. In the former case, the lock manifests the possibility that no process is willing to acquire the lock, in which case it deletes the mailbox and terminates. In the latter case, the lock manifests its expectation to be eventually released. Also note that **FreeLock** fails if it receives a **release** message. In this way, the lock manifests the fact that it can be released only if it is currently owned by a process. A system where two users *alice* and *carol* compete for acquiring *lock* can be modeled as the process

$$(\nu lock)(\nu alice)(\nu carol)(\text{FreeLock}[lock] \mid \text{User}[alice, lock] \mid \text{User}[carol, lock]) \quad (1)$$

where

$$\text{User}(self, lock) \triangleq lock!\text{acquire}[self] \mid self?\text{reply}(l).(l!\text{release} \mid \text{free } self.\text{done})$$

Note that **User** uses the reference l – as opposed to *lock* – to release the acquired lock. As we will see in Section 3.3, this is due to the fact that it is this particular reference to the lock's mailbox – and not *lock* itself – that carries the capability to release the lock.

► **Example 2 (future variable)**. A future variable is a one-place buffer that stores the result of an asynchronous computation. The content of the future variable is *resolved* once and for all by the producer once the computation completes. After that, its content can be retrieved any number of times by the consumers. If a consumer attempts to retrieve the content of the future variable beforehand, the consumer suspends until the variable is resolved. We can model a future variable thus:

$$\begin{aligned} \text{Future}(self) &\triangleq self?\text{put}(x).\text{Present}[self, x] \\ \text{Present}(self, x) &\triangleq \text{free } self.\text{done} \\ &\quad + self?\text{get}(sender).(sender!\text{reply}[x] \mid \text{Present}[self, x]) \\ &\quad + self?\text{put.fail } self \end{aligned}$$

15:6 Mailbox Types for Unordered Interactions

The process `Future` represents an unresolved future variable, which waits for a `put` message from the producer. Once the variable has been resolved, it behaves as specified by `Present`, namely it satisfies an arbitrary number of `get` messages from consumers but it no longer accepts `put` messages.

► **Example 3** (bank account). Below we see the process definition corresponding to the actor shown in Listing 1. The structure of the term follows closely that of the Scala code:

$$\begin{aligned} \text{Account}(\text{self}, \text{balance}) &\triangleq \text{self?debit}(\text{amount}, \text{sender}). \\ &\quad (\text{sender!reply} \mid \text{Account}[\text{self}, \text{balance} + \text{amount}]) \\ &+ \text{self?credit}(\text{amount}, \text{recipient}, \text{sender}). \\ &\quad (\text{recipient!debit}[\text{amount}, \text{self}] \mid \\ &\quad \text{self?reply}.\text{sender!reply} \mid \text{Account}[\text{self}, \text{balance} + \text{amount}]) \\ &+ \text{self?stop.free self.done} \\ &+ \text{self?reply.fail self} \end{aligned}$$

The last term of the guarded process, which results in a failure, corresponds to the catch-all clause in Listing 1 and models the fact that a `reply` message is not expected to be found in the account’s mailbox unless the account is involved in a transaction. The `reply` message is received and handled appropriately in the `credit`-guarded term.

We can model a deadlock in the case two distinct bank accounts attempt to initiate a transaction with one another. Indeed, we have

$$\begin{aligned} \text{Account}[\text{alice}, 8] \mid \text{alice!credit}[2, \text{carol}, \text{bank}] \mid &\rightarrow^* \text{carol!debit}[2, \text{alice}] \mid \text{alice?reply}\dots \mid \\ \text{Account}[\text{carol}, 9] \mid \text{carol!credit}[5, \text{alice}, \text{bank}] &\rightarrow^* \text{alice!debit}[5, \text{carol}] \mid \text{carol?reply}\dots \end{aligned}$$

where both *alice* and *carol* ignore the incoming `debit` messages, whence the deadlock.

We now provide operational characterizations of the properties enforced by our typing discipline. We begin with mailbox conformance, namely the property that a process never fails because of unexpected messages. To this aim, we define a process context \mathcal{C} as a process in which there is a single occurrence of an *unguarded hole* $[\]$:

$$\mathcal{C} ::= [\] \mid \mathcal{C} \mid P \mid P \mid \mathcal{C} \mid (\nu a)\mathcal{C}$$

The hole is “unguarded” in the sense that it does not occur prefixed by an action. As usual, we write $\mathcal{C}[P]$ for the process obtained by replacing the hole in \mathcal{C} with P . Names may be captured by this replacement. A mailbox conformant process never reduces to a state in which the only action of a guard is `fail`:

► **Definition 4.** We say that P is *mailbox conformant* if $P \rightarrow^* \mathcal{C}[\text{fail } a]$ for all \mathcal{C} and a .

Looking at the placement of the `fail` u actions in earlier examples we can give the following interpretations of mailbox conformance: a lock is never released unless it has been acquired beforehand (Example 1); a future variable is never resolved twice (Example 2); an account will not be notified of a completed transaction (with a `reply` message) unless it is involved in an ongoing transaction (Example 3).

We express deadlock freedom as the property that all irreducible residuals of a process are (structurally equivalent to) the terminated process:

► **Definition 5.** We say that P is *deadlock free* if $P \rightarrow^* Q \rightarrow$ implies $Q \equiv \text{done}$.

According to Definition 5, if a deadlock-free process halts we have that: (1) there is no sub-process waiting for a message that is never produced; (2) every mailbox is empty. Clearly, this is not the case for the transaction between *alice* and *carol* in Example 3.

► **Example 6** (deadlock). Below is another example of deadlocking process using `Future` from Example 2, obtained by resolving a future variable with the value it does not contain yet:

$$(\nu f)(\nu c)(\text{Future}[f] \mid f!\text{get}[c] \mid c?\text{reply}(x).\text{free } c.f!\text{put}[x]) \quad (2)$$

Notice that attempting to retrieve the content of a future variable not knowing whether it has been resolved is legal. Indeed, `Future` does not fail if a `get` message is present in the future variable's mailbox before it is resolved. Thus, the deadlocked process above is mailbox conformant but also an instance of undesirable process that will be ruled out by our static analysis technique (cf. Example 21). We will need dependency graphs in addition to types to flag this process as ill typed.

A property stronger than deadlock freedom is fair termination. A fairly terminating process is a process whose residuals always have the possibility to terminate. Formally:

► **Definition 7.** We say that P is *fairly terminating* if $P \rightarrow^* Q$ implies $Q \rightarrow^* \text{done}$.

An interesting consequence of fair termination is that it implies *junk freedom* (also known as lock freedom [34, 44]) namely the property that every message can be eventually consumed. Our type system does not guarantee fair termination nor junk freedom in general, but it does so for a non-trivial sub-class of well-typed processes that we characterize later on.

3 A Mailbox Type System

In this section we detail the type system for the mailbox calculus. We start from the syntax and semantics of mailbox types (Section 3.1) and of dependency graphs (Section 3.2), the mechanism we use to track mailbox dependencies. Then we present the typing rules (Section 3.3) and the properties of well-typed processes (Section 3.4).

3.1 Mailbox Types

The syntax of mailbox types and patterns is shown below:

$$\begin{array}{l} \text{Mailbox Type } \tau, \sigma ::= ?E \mid !E \\ \text{Pattern } E, F ::= 0 \mid \mathbb{1} \mid \mathfrak{m}[\bar{\tau}] \mid E + F \mid E \cdot F \mid E^* \end{array} \quad (3)$$

Patterns are *commutative regular expressions* [11] describing the configurations of messages stored in a mailbox. An atom $\mathfrak{m}[\bar{\tau}]$ describes a mailbox containing a single message with tag \mathfrak{m} and arguments of type $\bar{\tau}$. We let M range over atoms and abbreviate $\mathfrak{m}[\bar{\tau}]$ with \mathfrak{m} when $\bar{\tau}$ is the empty sequence. Compound patterns are built using sum ($E + F$), product ($E \cdot F$) and exponential (E^*). The constants $\mathbb{1}$ and 0 respectively describe the empty and the unreliable mailbox. There is no configuration of messages stored in an unreliable mailbox, not even the empty one. We will use the 0 pattern for describing mailboxes from which an unexpected message has been received. Let us look at a few simple examples. The pattern $\mathbf{A} + \mathbf{B}$ describes a mailbox that contains either an \mathbf{A} message or a \mathbf{B} message, but not both, whereas the pattern $\mathbf{A} + \mathbb{1}$ describes a mailbox that either contains a \mathbf{A} message or is empty. The pattern $\mathbf{A} \cdot \mathbf{B}$ describes a mailbox that contains both an \mathbf{A} message and also a \mathbf{B} message. Note that \mathbf{A} and \mathbf{B} may be equal, in which case the mailbox contains *two* \mathbf{A} messages. Finally, the pattern \mathbf{A}^* describes a mailbox that contains an arbitrary number (possibly zero) of \mathbf{A} messages. We adopt the usual conventions on the priority of connectives, whereby $*$ binds stronger than \cdot which, in turn, binds stronger than $+$.

15:8 Mailbox Types for Unordered Interactions

A *mailbox type* consists of a *capability* (either $?$ or $!$) paired with a pattern. The capability specifies whether the pattern describes messages to be received from ($?$) or stored in ($!$) the mailbox. Here are some examples: A process using a mailbox of type $!A$ *must* store an A message into the mailbox, whereas a process using a mailbox of type $?A$ expects to receive an A message from the mailbox. A process using a mailbox of type $!(A + \mathbb{1})$ *may* store an A message into the mailbox, but is not obliged to do so. A process using a mailbox of type $!(A + B)$ decides whether to store an A message or a B message in the mailbox, whereas a process using a mailbox of type $?(A + B)$ must be ready to receive both kinds of messages. A process using a mailbox of type $?(A \cdot B)$ expects to receive both an A message and a B message and may decide in which order to do so. A process using a mailbox of type $!(A \cdot B)$ must store both A and B into the mailbox. A process using a mailbox of type $!A^*$ decides how many A messages to store in the mailbox, whereas a process using a mailbox of type $?A^*$ must be prepared to receive an arbitrary number of A messages.

To cope with possibly infinite types we interpret the productions in (3) coinductively and consider as types the regular trees [13] built using those productions. We require every infinite branch of a type tree to go through infinitely many atoms. This strengthened contractiveness condition allows us to define functions inductively on the structure of patterns, provided that these functions do not recur into argument types (*cf.* Definitions 8 and 14).

The semantics of a pattern is a *set of multisets of atoms*. Because patterns include types, the given semantics is parametric in the subtyping relation, which will be defined next:

► **Definition 8** (subpattern). The *configurations* of E are inductively defined by the following equations, where A and B range over multisets $\langle \overline{M} \rangle$ of atoms and \uplus denotes multiset union:

$$\begin{aligned} \llbracket \mathbb{0} \rrbracket &\stackrel{\text{def}}{=} \emptyset & \llbracket E + F \rrbracket &\stackrel{\text{def}}{=} \llbracket E \rrbracket \cup \llbracket F \rrbracket & \llbracket M \rrbracket &\stackrel{\text{def}}{=} \{ \langle M \rangle \} \\ \llbracket \mathbb{1} \rrbracket &\stackrel{\text{def}}{=} \{ \langle \rangle \} & \llbracket E \cdot F \rrbracket &\stackrel{\text{def}}{=} \{ A \uplus B \mid A \in \llbracket E \rrbracket, B \in \llbracket F \rrbracket \} & \llbracket E^* \rrbracket &\stackrel{\text{def}}{=} \llbracket \mathbb{1} \rrbracket \cup \llbracket E \rrbracket \cup \llbracket E \cdot E \rrbracket \cup \dots \end{aligned}$$

Given a preorder relation \mathcal{R} on types, we write $E \sqsubseteq_{\mathcal{R}} F$ if $\langle \mathfrak{m}_i[\overline{\tau}_i] \rangle_{i \in I} \in \llbracket E \rrbracket$ implies $\langle \mathfrak{m}_i[\overline{\sigma}_i] \rangle_{i \in I} \in \llbracket F \rrbracket$ and $\overline{\tau}_i \mathcal{R} \overline{\sigma}_i$ for every $i \in I$. We write $\simeq_{\mathcal{R}}$ for $\sqsubseteq_{\mathcal{R}} \cap \supseteq_{\mathcal{R}}$.

For example, $\llbracket A + B \rrbracket = \{ \langle A \rangle, \langle B \rangle \}$ and $\llbracket A \cdot B \rrbracket = \{ \langle A, B \rangle \}$. It is easy to see that $\sqsubseteq_{\mathcal{R}}$ is a pre-congruence with respect to all the connectives and that $\simeq_{\mathcal{R}}$ includes all the known laws of commutative Kleene algebra [11]: both $+$ and \cdot are commutative and associative, $+$ is idempotent and has unit $\mathbb{0}$, \cdot distributes over $+$, it has unit $\mathbb{1}$ and is absorbed by $\mathbb{0}$. Also observe that $\sqsubseteq_{\mathcal{R}}$ is related covariantly to \mathcal{R} , that is $\overline{\tau} \mathcal{R} \overline{\sigma}$ implies $\mathfrak{m}[\overline{\tau}] \sqsubseteq_{\mathcal{R}} \mathfrak{m}[\overline{\sigma}]$.

We now define subtyping. As types may be infinite, we resort to coinduction:

► **Definition 9** (subtyping). We say that \mathcal{R} is a *subtyping relation* if $\tau \mathcal{R} \sigma$ implies either

1. $\tau = ?E$ and $\sigma = ?F$ and $E \sqsubseteq_{\mathcal{R}} F$, or
2. $\tau = !E$ and $\sigma = !F$ and $F \sqsubseteq_{\mathcal{R}} E$.

We write \leq for the largest subtyping relation and say that τ is a *subtype* of σ (and σ a *supertype* of τ) if $\tau \leq \sigma$. We write \leq for $\leq \cap \geq$, \sqsubseteq for $\sqsubseteq \leq$ and \simeq for $\simeq \leq$.

Items 1 and 2 respectively correspond to the usual covariant and contravariant rules for channel types with input and output capabilities [47]. For example, $!(A + B) \leq !A$ because a mailbox of type $!(A + B)$ is more permissive than a mailbox of type $!A$. Dually, $?A \leq ?(A + B)$ because a mailbox of type $?A$ provides stronger guarantees than a mailbox of type $?(A + B)$. Note that $!(A \cdot B) \leq !(B \cdot A)$ and $?(A \cdot B) \leq ?(B \cdot A)$, to witness the fact that the order in which messages are stored in a mailbox is irrelevant.

Mailbox types whose patterns are in particular relations with the constants $\mathbb{0}$ and $\mathbb{1}$ will play special roles, so we introduce some corresponding terminology.

- **Definition 10** (type and name classification). We say that (a name whose type is) τ is:
- *relevant* if $\tau \not\leq !\mathbb{1}$ and *irrelevant* otherwise;
 - *reliable* if $\tau \not\leq ?\mathbb{0}$ and *unreliable* otherwise;
 - *usable* if $!\mathbb{0} \not\leq \tau$ and *unusable* otherwise.

A relevant name *must* be used, whereas an irrelevant name may be discarded because not storing any message in the mailbox it refers to is allowed by its type. All mailbox types with input capability are relevant. A reliable mailbox is one from which no unexpected message has been received. All names with output capability are reliable. A usable name *can* be used, in the sense that there exists a construct of the mailbox calculus that expects a name with that type. All mailbox types with input capability are usable, but $?(A \cdot \mathbb{0})$ is unreliable. Both $!A$ and $!(\mathbb{1} + A)$ are usable. The former type is also relevant because a process using a mailbox with this type must (eventually) store an A message in it. On the contrary, the latter type is irrelevant, since not using the mailbox is a legal way of using it.

Henceforth we assume that all types are usable and that all argument types are also reliable. That is, we ban all types like $!\mathbb{0}$ or $!(\mathbb{0} \cdot m)$ and all types like $?m[?0]$ or $!m[?0]$.

► **Example 11** (lock type). The mailbox used by the lock (Example 1) will have several different types, depending on the viewpoint we take (either the lock itself or one of its users) and on the state of the lock (whether it is free or busy). As we can see from the definition of `FreeLock`, a free lock waits for an `acquire` message which is supposed to carry a reference to another mailbox into which the capability to release the lock is stored. Since the lock is meant to have several concurrent users, it is not possible in general to predict the number of `acquire` messages in its mailbox. Therefore, the mailbox of a free lock has type

$$?acquire[!reply[!release]]^*$$

from the viewpoint of the lock itself. When the lock is busy, it expects to find one `release` message in its mailbox, but in general the mailbox will also contain `acquire` messages corresponding to pending acquisition requests. So, the mailbox of a busy lock has type

$$?(release \cdot acquire[!reply[!release]]^*)$$

indicating that the mailbox contains (or will eventually contain) a single `release` message along with arbitrarily many `acquire` messages.

Prospective owners of the lock may have references to the lock's mailbox with type $!acquire[!reply[!release]]$ or $!acquire[!reply[!release]]^*$ depending on whether they acquire the lock exactly once (just like *alice* and *carol* in Example 1) or several times. Other intermediate types are possible in the case of users that acquire the lock a bounded number of times. The current owner of the lock will have a reference to the lock's mailbox of type $!release$. This type is relevant, implying that the owner must eventually release the lock.

3.2 Dependency Graphs

We use *dependency graphs* for tracking dependencies between mailboxes. Intuitively, there is a dependency between u and v if either v is the argument of a message in mailbox u or v occurs in the continuation of a process waiting for a message from u . Dependency graphs have names as vertices and undirected edges. However, the usual representation of graphs does not account for the fact that mailbox names may be restricted and that the multiplicity of dependencies matters. Therefore, we define dependency graphs using the syntax below:

Dependency Graph $\varphi, \psi ::= \emptyset \mid \{u, v\} \mid \varphi \sqcup \psi \mid (\nu a)\varphi$

15:10 Mailbox Types for Unordered Interactions

$$\begin{array}{c}
\{u, v\} \xrightarrow{u-v} \emptyset \quad [\text{G-AXIOM}] \qquad \frac{\varphi \xrightarrow{u-v} \varphi'}{\varphi \sqcup \psi \xrightarrow{u-v} \varphi' \sqcup \psi} \quad [\text{G-LEFT}] \qquad \frac{\psi \xrightarrow{u-v} \psi'}{\varphi \sqcup \psi \xrightarrow{u-v} \varphi \sqcup \psi'} \quad [\text{G-RIGHT}] \\
\\
\frac{\varphi \xrightarrow{u-v} \psi \quad a \neq u, v}{(\nu a)\varphi \xrightarrow{u-v} (\nu a)\psi} \quad [\text{G-NEW}] \qquad \frac{\varphi \xrightarrow{u-w} \psi \quad \psi \xrightarrow{w-v} \varphi'}{\varphi \xrightarrow{u-v} \varphi'} \quad [\text{G-TRANS}]
\end{array}$$

■ **Figure 1** Labelled transitions of dependency graphs.

The term \emptyset represents the empty graph which has no vertices and no edges. The unordered pair $\{u, v\}$ represents the graph made of a single edge connecting the vertices u and v . The term $\varphi \sqcup \psi$ represents the union of φ and ψ whereas $(\nu a)\varphi$ represents the same graph as φ except that the vertex a is restricted. The usual notions of free and bound names apply to dependency graphs. We write $\text{fn}(\varphi)$ for the free names of φ .

To define the semantics of a dependency graph we use the labelled transition system of Table 1. A label $u - v$ represents a path connecting u with v . So, a relation $\varphi \xrightarrow{u-v} \varphi'$ means that u and v are connected in φ and φ' describes the residual edges of φ that have not been used for building the path between u and v . The paths of φ are built from the edges of φ (cf. [G-AXIOM]) connected by shared vertices (cf. [G-TRANS]). Restricted names cannot be observed in labels, but they may contribute in building paths in the graph (cf. [G-NEW]).

► **Definition 12** (graph acyclicity and entailment). Let $\text{dep}(\varphi) \stackrel{\text{def}}{=} \{(u, v) \mid \exists \varphi' : \varphi \xrightarrow{u-v} \varphi'\}$ be the *dependency relation* generated by φ . We say that φ is *acyclic* if $\text{dep}(\varphi)$ is irreflexive. We say that φ *entails* ψ , written $\varphi \Rightarrow \psi$, if $\text{dep}(\psi) \subseteq \text{dep}(\varphi)$.

Note that \sqcup is commutative, associative and has \emptyset as unit with respect to $\text{dep}(\cdot)$. These properties of dependency graphs are key to prove that typing is preserved by structural congruence on processes. Note also that \sqcup is *not* idempotent. Indeed, $\{u, v\} \sqcup \{u, v\}$ is cyclic whereas $\{u, v\}$ is not. The following example motivates the reason why the multiplicity of dependencies is important.

► **Example 13** (multiplicity of dependencies). Even though we have not presented the typing rules yet, we can use the above intuition on how dependencies are established to argue that the process below yields a cyclic dependency graph:

$$P \stackrel{\text{def}}{=} (\nu a)(\nu b)(a!A[b] \mid a!B[b] \mid a?A(x).a?B(y).\text{free } a.x!m[y]) \rightarrow^* (\nu b)b!m[b] \rightarrow$$

Observe that P stores two messages in the mailbox a , each containing a reference to the mailbox b . Thus, the same dependency $\{a, b\}$ arises twice, resulting in a cyclic dependency involving a and b . By reducing the process, we note that the two variables x and y , which were syntactically different in P , are unified into b in the reduct, which is deadlocked.

3.3 Typing Rules

We use *type environments* for tracking the type of free names occurring in processes. A type environment is a partial function from names to types written as $\bar{u} : \bar{\tau}$ or $u_1 : \tau_1, \dots, u_n : \tau_n$. We let Γ and Δ range over type environments, we write $\text{dom}(\Gamma)$ for the domain of Γ and Γ, Δ for the union of Γ and Δ when $\text{dom}(\Gamma) \cap \text{dom}(\Delta) = \emptyset$. We say that Γ is reliable if so are all the types in its range.

Typing rules for processes

$$\boxed{\Gamma \vdash P :: \varphi}$$

$$\frac{}{\emptyset \vdash \mathbf{done} :: \emptyset} \text{ [T-DONE]} \quad \frac{X : (\bar{x} : \bar{\tau}; \varphi)}{\bar{u} : \bar{\tau} \vdash X[\bar{u}] :: \varphi\{\bar{u}/\bar{x}\}} \text{ [T-DEF]} \quad \frac{\Gamma, a : ?\mathbb{1} \vdash P :: \varphi}{\Gamma \vdash (\nu a)P :: (\nu a)\varphi} \text{ [T-NEW]}$$

$$\frac{}{u : !\mathbf{m}[\bar{\tau}], \bar{v} : \bar{\tau} \vdash u!\mathbf{m}[\bar{v}] :: \{u, \{\bar{v}\}\}} \text{ [T-MSG]} \quad \frac{u : ?E, \Gamma \vdash G \quad \vDash E}{u : ?E, \Gamma \vdash G :: \{u, \mathbf{dom}(\Gamma)\}} \text{ [T-GUARD]}$$

$$\frac{\Gamma_i \vdash P_i :: \varphi_i \quad (i=1,2)}{\Gamma_1 \parallel \Gamma_2 \vdash P_1 \mid P_2 :: \varphi_1 \sqcup \varphi_2} \text{ [T-PAR]} \quad \frac{\Delta \vdash P :: \psi \quad \Gamma \leq \Delta \quad \varphi \Rightarrow \psi}{\Gamma \vdash P :: \varphi} \text{ [T-SUB]}$$

Typing rules for guards

$$\boxed{\Gamma \vdash G}$$

$$\frac{}{u : ?\emptyset, \Gamma \vdash \mathbf{fail} \ u} \text{ [T-FAIL]} \quad \frac{\Gamma \vdash P :: \varphi}{u : ?\mathbb{1}, \Gamma \vdash \mathbf{free} \ u.P} \text{ [T-FREE]}$$

$$\frac{u : ?E, \Gamma, \bar{x} : \bar{\tau} \vdash P :: \varphi}{u : ?(\mathbf{m}[\bar{\tau}] \cdot E), \Gamma \vdash u?\mathbf{m}(\bar{x}).P} \text{ [T-IN]} \quad \frac{u : ?E_i, \Gamma \vdash G_i \quad (i=1,2)}{u : ?(E_1 + E_2), \Gamma \vdash G_1 + G_2} \text{ [T-BRANCH]}$$

■ **Figure 2** Typing rules.

Judgments for processes have the form $\Gamma \vdash P :: \varphi$, meaning that P is well typed in Γ and yields the dependency graph φ . Judgments for guards have the form $\Gamma \vdash G$, meaning that G is well typed in Γ . We say that a judgment $\Gamma \vdash P :: \varphi$ is well formed if $\mathbf{fn}(\varphi) \subseteq \mathbf{dom}(\Gamma)$ and φ is acyclic. Each process typing rule has an implicit side condition requiring that its conclusion be well formed. For each global process definition $X(\bar{x}) \triangleq P$ we assume that there is a corresponding *global process declaration* of the form $X : (\bar{x} : \bar{\tau}; \varphi)$. We say that the definition is *consistent* with the corresponding declaration if $\bar{x} : \bar{\tau} \vdash P :: \varphi$, where we require the dependency graph yielded by P to be the same φ associated with the definition. Hereafter, all process definitions are assumed to be consistent. We now discuss the typing rules in detail, introducing auxiliary notions and notation as we go along.

Terminated process. According to the rule [T-DONE], the terminated process **done** is well typed in the empty type environment and yields no dependencies. This is motivated by the fact that **done** does not use any mailbox. Later on we will introduce a subsumption rule [T-SUB] that allows us to type **done** in any type environment with irrelevant names.

Message. Rule [T-MSG] establishes that a message $u!\mathbf{m}[\bar{v}]$ is well typed provided that the mailbox u allows the storing of an \mathbf{m} -tagged message with arguments of type $\bar{\tau}$ and the types of \bar{v} are indeed $\bar{\tau}$. The subsumption rule [T-SUB] will make it possible to use arguments whose type is a *subtype* of the expected ones. A message $u!\mathbf{m}[\bar{v}]$ establishes dependencies between the target mailbox u and all of the arguments \bar{v} . We write $\{u, \{v_1, \dots, v_n\}\}$ for the dependency graph $\{u, v_1\} \sqcup \dots \sqcup \{u, v_n\}$ and use \emptyset for the empty graph union.

Names with output capability are introduced by the \parallel operator that combines type environments and that will be defined later on, when discussing rule [T-PAR].

15:12 Mailbox Types for Unordered Interactions

Process invocation. The typing rule for a process invocation $X[\bar{u}]$ checks that there exists a global definition for X which expects exactly the given number and type of parameters. Again, rule $[\text{T-SUB}]$ will make it possible to use parameters whose types are subtypes of the expected ones. A process invocation yields the same dependencies as the corresponding process definition, with the appropriate substitutions applied.

Guards. Guards are used to match the content of a mailbox and retrieve messages from it. According to rule $[\text{T-FAIL}]$, the action **fail** u matches a mailbox u with type $?0$, indicating that an unexpected message has been found in the mailbox. The type environment may contain arbitrary associations, since the **fail** u action causes a runtime error. Rule $[\text{T-FREE}]$ states that the action **free** $u.P$ matches a mailbox u with type $?1$, indicating that the mailbox is empty. The continuation is well typed in the residual type environment Γ . An input action $u?m(\bar{x}).P$ matches a mailbox u with type $?m[\bar{\tau}] \cdot E$ that guarantees the presence of an m -tagged message possibly along with other messages as specified by E . The continuation P must be well typed in an environment where the mailbox has type $?E$, which describes the content of the mailbox after the m -tagged message has been removed. Associations for the received arguments \bar{x} are also added to the type environment. A compound guard $G_1 + G_2$ offers the actions offered by G_1 and G_2 and therefore matches a mailbox u with type $?(E_1 + E_2)$, where E_i is the pattern that describes the mailbox matched by G_i . Note that the residual type environment Γ is the same in both branches, indicating that the type of other mailboxes used by the guard cannot depend on that of u .

The judgments for guards do not yield any dependency graph. This is compensated by the rule $[\text{T-GUARD}]$, which we describe next.

Guarded processes. Rule $[\text{T-GUARD}]$ is used to type a guarded process G , which matches some mailbox u of type $?E$ and possibly retrieves messages from it. As we have seen while discussing guards, E is supposed to be a pattern of the form $E_1 + \dots + E_n$ where each E_i is either 0 , 1 or of the form $M \cdot F$. However, only the patterns E that are in *normal form* (defined below) are suitable to be used in this typing rule and the side condition $\models E$ checks that this is indeed the case. Before defining the notion of pattern normal form we motivate its need by means of a simple example.

Suppose that our aim is to type a process $u?A.P + u?B.Q$ that consumes either an **A** message or a **B** message from u , whichever of these two messages is matched first in u , and then continues as P or Q correspondingly. Suppose also that the type of u is $?E$ with $E \stackrel{\text{def}}{=} A \cdot C + B \cdot A$, which allows the rules for guards to successfully type check the process. As we have seen while discussing rule $[\text{T-IN}]$, P and Q must be typed in an environment where the type of u has been updated so as to reflect the fact that the consumed message is no longer in the mailbox. In this particular case, we might be tempted to infer that the type of u in P is $?C$ and that the type of u in Q is $?A$. Unfortunately, the type $?C$ does not accurately describe the content of the mailbox after **A** has been consumed because, according to E , the **A** message may be accompanied by *either* a **B** message *or* by a **C** message, whereas $?C$ only accounts for the second possibility. Thus, the appropriate pattern to be used for typing this process is $A \cdot (B + C) + B \cdot A$, where the fact that **B** may be found after consuming **A** is made explicit. This pattern and E are equivalent as they generate exactly the same set of valid configurations. Yet, $A \cdot (B + C) + B \cdot A$ is in normal form whereas E is not. In general the normal form is not unique. For example, also the patterns $B \cdot A + C \cdot A$ and $A \cdot (B + C)$ are in normal form and equivalent to E and can be used for typing processes that consume messages from u in different orders or with different priorities.

The first ingredient for defining the notion of pattern normal form is that of pattern residual E/M , which describes the content of a mailbox that initially contains a configuration of messages described by E and from which we remove a single message with type M :

► **Definition 14** (pattern residual). The *residual* of a pattern E with respect to an atom M , written E/M , is inductively defined by the following equations:

$$\begin{aligned} 0/M &= 1/M \stackrel{\text{def}}{=} 0 & \mathfrak{m}[\bar{\tau}]/\mathfrak{m}[\bar{\sigma}] &\stackrel{\text{def}}{=} 1 & \text{if } \bar{\tau} \leq \bar{\sigma} & (E + F)/M &\stackrel{\text{def}}{=} E/M + F/M \\ (E^*)/M &\stackrel{\text{def}}{=} E/M \cdot E^* & \mathfrak{m}[\bar{\tau}]/\mathfrak{m}'[\bar{\sigma}] &\stackrel{\text{def}}{=} 0 & \text{if } \mathfrak{m} \neq \mathfrak{m}' & (E \cdot F)/M &\stackrel{\text{def}}{=} E/M \cdot F + E \cdot F/M \end{aligned}$$

If we take the pattern E discussed earlier we have $E/A = 1 \cdot C + A \cdot 0 + 0 \cdot 1 + B \cdot 1 \simeq B + C$. The pattern residual operator is closely related to Brzozowski's derivative in a commutative Kleene algebra [4, 28]. Unlike Brzozowski's derivative, the pattern residual is a partial operator: $E/\mathfrak{m}[\bar{\sigma}]$ is defined provided that the $\bar{\sigma}$ are supertypes of all types $\bar{\tau}$ found in \mathfrak{m} -tagged atoms within E . This condition has a natural justification: when choosing the message to remove from a mailbox containing a configuration of messages described by E , only the tag \mathfrak{m} of the message – and not the type of its arguments – matters. Thus, $\bar{\sigma}$ faithfully describe the received arguments provided that they are supertypes of *all* argument types of *all* \mathfrak{m} -tagged message types in E . For example, assuming $\text{nat} \leq \text{int}$, we have that $(\mathfrak{m}[\text{int}] + \mathfrak{m}[\text{nat}])/\mathfrak{m}[\text{int}]$ is defined whereas $(\mathfrak{m}[\text{int}] + \mathfrak{m}[\text{nat}])/\mathfrak{m}[\text{nat}]$ is not.

We use the notion of pattern residual to define pattern normal forms:

► **Definition 15** (pattern normal form). We say that a pattern E is in *normal form*, written $\vDash E$, if $E \vDash E$ is derivable by the following axioms and rules:

$$\begin{array}{c} E \vDash 0 \quad E \vDash 1 \quad \frac{F \simeq E/M}{E \vDash M \cdot F} \quad \frac{E \vDash F_1 \quad E \vDash F_2}{E \vDash F_1 + F_2} \end{array}$$

Essentially, the judgment $\vDash E$ verifies that E is expressed as a sum of 0 , 1 and $M \cdot F$ terms where F is (equivalent to) the residual of E with respect to M .

A guarded process yields all the dependencies between the mailbox u being used and the names occurring free in the continuations, because the process will not be able to exercise the capabilities on these names until the message from u has been received.

Parallel composition. Rule $[_{\text{T-PAR}}]$ deals with parallel compositions of the form $P_1 \mid P_2$. This rule accounts for the fact that the same mailbox u may be used in both P_1 and P_2 according to different types. For example, P_1 might store an \mathbf{A} message into u and P_2 might store a \mathbf{B} message into u . In the type environment for the parallel composition as a whole we must be able to express with a single type the combined usages of u in P_1 and P_2 . This is accomplished by introducing an operator that combines types:

► **Definition 16** (type combination). We write $\tau \parallel \sigma$ for the *combination* of τ and σ , where \parallel is the partial symmetric operator defined as follows:

$$!E \parallel !F \stackrel{\text{def}}{=} !(E \cdot F) \quad !E \parallel ?(E \cdot F) \stackrel{\text{def}}{=} ?F \quad ?(E \cdot F) \parallel !E \stackrel{\text{def}}{=} ?F$$

Continuing the previous example, we have $!\mathbf{A} \parallel !\mathbf{B} = !(\mathbf{A} \cdot \mathbf{B})$ because storing one \mathbf{A} message and one \mathbf{B} message in u means storing an overall configuration of messages described by the pattern $\mathbf{A} \cdot \mathbf{B}$. When u is used for both input and output operations, the combined type of u describes the overall balance of the mailbox. For example, we have $!\mathbf{A} \parallel ?(\mathbf{A} \cdot \mathbf{B}) = ?\mathbf{B}$: if we combine a process that stores an \mathbf{A} message into u with another process that consumes both

15:14 Mailbox Types for Unordered Interactions

an A message and a B message from the same mailbox in some unspecified order, then we end up with a process that consumes a B message from u .

Notice that \parallel is a partial operator in that not all type combinations are defined. It might be tempting to relax \parallel in such a way that $!(A \cdot B) \parallel ?A = !B$, so as to represent the fact that the combination of two processes results in an excess of messages that must be consumed by some other process. However, this would mean allowing different processes to consume messages from the same mailbox, which is not safe in general (see Example 17). For the same reason, the combination of $?E$ and $?F$ is always undefined regardless of E and F . Operators akin to \parallel for the combination of channel types are commonly found in substructural type systems for the (linear) π -calculus [51, 44]. Unlike these systems, in our case the combination concerns also the content of a mailbox in addition to the capabilities for accessing it.

► **Example 17.** Suppose that we extend the type combination operator so that $?(E \cdot F) = ?E \parallel ?F$. To see why this extension would be dangerous, consider the process

$$(u!A[\mathbf{True}] \mid u?A(x).(system!print_bool[x] \mid free\ u.done)) \mid \\ (u!A[2] \mid u?A(y).(system!print_int[y] \mid free\ u.done))$$

Overall, this process stores into u a combination of messages that matches the pattern $A[\mathbf{bool}] \cdot A[\mathbf{int}]$ and retrieves from u the same combination of messages. Apparently, u is used in a balanced way. However, there is no guarantee that the $u!A[\mathbf{True}]$ message is received by the process at the top and that the $u!A[2]$ message is received by the process at the bottom. In fact, the converse may happen because only the tag of a message – not the type or value of its arguments – is used for matching messages in the mailbox calculus.

We now extend type combination to type environments in the expected way:

► **Definition 18** (type environment combination). We write $\Gamma \parallel \Delta$ for the *combination* of Γ and Δ , where \parallel is the partial operator inductively defined by the equations:

$$\Gamma \parallel \Delta \stackrel{\text{def}}{=} \Gamma, \Delta \quad \text{if } \text{dom}(\Gamma) \cap \text{dom}(\Delta) = \emptyset \quad (u : \tau, \Gamma) \parallel (u : \sigma, \Delta) \stackrel{\text{def}}{=} u : \tau \parallel \sigma, (\Gamma \parallel \Delta)$$

With this machinery in place, rule $[\text{T-PAR}]$ is straightforward to understand and the dependency graph of $P_1 \mid P_2$ is simply the union of the dependency graphs of P_1 and P_2 .

Mailbox restriction. Rule $[\text{T-NEW}]$ establishes that the process creating a new mailbox a with scope P is well typed provided that the type of a is $?1$. This means that every message stored in the mailbox a by (a sub-process of) P is also consumed by (a sub-process of) P . The dependency graph of the process is the same as that of P , except that a is restricted.

Subsumption. As we have anticipated earlier in a few occasions, the subsumption rule $[\text{T-SUB}]$ allows us to rewrite types in the type environment and to introduce associations for irrelevant names. The rule makes use of the following notion of subtyping for type environments:

► **Definition 19** (subtyping for type environments). We say that Γ is a *subtype environment* of Δ if $\Gamma \leq \Delta$, where \leq is the least preorder on type environments such that:

$$\frac{}{u : !1, \Gamma \leq \Gamma} \quad \frac{\tau \leq \sigma}{u : \tau, \Gamma \leq u : \sigma, \Gamma}$$

Intuitively, $\Gamma \leq \Delta$ means that Γ provides more capabilities than Δ . For example, $u : !(A + B), v : !1 \leq u : !A$ since a process that is well typed in the environment $u : !A$ stores

an **A** message into u , which is also a valid behavior in the environment $u : !(A + B), v : !\mathbb{1}$ where u has more capabilities (it is also possible to store a **B** message into u) and there is an irrelevant name v not used by the process.

Rule $[\text{T-SUB}]$ also allows us to replace the dependency graph yielded by P with another one that generates a superset of dependencies. In general, the dependency graph should be kept as small as possible to minimize the possibility of yielding mutual dependencies (see $[\text{T-PAR}]$). The replacement allowed by $[\text{T-SUB}]$ is handy for technical reasons, but not necessary. The point is that the residual of a process typically yields fewer dependencies than the process itself, so we use $[\text{T-SUB}]$ to enforce the invariance of dependency graphs across reductions.

► **Example 20.** We show the full typing derivation for **FreeLock** and **BusyLock** defined in Example 1. Our objective is to show the consistency of the global process declarations

$$\text{FreeLock} : (\text{self} : \tau; \emptyset) \quad \text{BusyLock} : (\text{self} : \tau, \text{owner} : \rho; \{\text{self}, \text{owner}\})$$

where $\tau \stackrel{\text{def}}{=} ?\text{acquire}[\rho]^*$ and $\rho \stackrel{\text{def}}{=} !\text{reply}[\text{!release}]$. In the derivation trees below we rename self as x and owner and y to reasonably fit the derivations within the page limits. We start from the body of **BusyLock**, which is simpler, and obtain

$$\frac{\frac{\frac{\frac{}{x : \tau \vdash \text{FreeLock}[x] :: \emptyset} [\text{T-DEF}]}{x : \sigma \vdash x? \text{release} . \text{FreeLock}[x]} [\text{T-IN}]}{x : \sigma \vdash x? \text{release} . \text{FreeLock}[x] :: \emptyset} [\text{T-GUARD}]}{x : ! \text{release}, y : \rho \vdash y! \text{reply}[x] :: \{y, x\}} [\text{T-MSG}]}{x : \tau, y : \rho \vdash y! \text{reply}[x] \mid x? \text{release} . \text{Lock}[x] :: \{y, x\} \sqcup \emptyset} [\text{T-PAR}]$$

where $\sigma \stackrel{\text{def}}{=} ?(\text{release} \cdot \text{acquire}[\rho]^*)$.

Concerning **FreeLock**, the key step is rewriting the pattern of τ in a normal form that matches the branching structure of the process. To this aim, we use the property $E^* \simeq \mathbb{1} + E \cdot E^*$ and the fact that \emptyset is absorbing for the product connective:

$$\frac{\frac{\frac{\frac{}{x : ?\emptyset \vdash \text{fail } x} [\text{T-FAIL}]}{x : ?\emptyset \vdash \text{fail } x :: \emptyset} [\text{T-GUARD}]}{\vdots} \frac{}{x : ?(\text{release} \cdot \emptyset) \vdash x? \text{release} . \text{fail } x} [\text{T-IN}]}{x : ?(\mathbb{1} + \text{acquire}[\rho] \cdot \text{acquire}[\rho]^* + \text{release} \cdot \emptyset) \vdash \dots + x? \text{release} . \text{fail } x} [\text{T-BRANCH}]}{x : ?(\mathbb{1} + \text{acquire}[\rho] \cdot \text{acquire}[\rho]^* + \text{release} \cdot \emptyset) \vdash \dots + x? \text{release} . \text{fail } x :: \emptyset} [\text{T-GUARD}]}{x : \tau \vdash \text{free } x . \text{done} + \dots + x? \text{release} . \text{fail } x :: \emptyset} [\text{T-SUB}]$$

The elided sub-derivation concerns the first two branches of **FreeLock** and is as follows:

$$\frac{\frac{\frac{}{\emptyset \vdash \text{done} :: \emptyset} [\text{T-DONE}]}{x : ?\mathbb{1} \vdash \text{free } x . \text{done}} [\text{T-FREE}]}{x : ?(\mathbb{1} + \text{acquire}[\rho] \cdot \text{acquire}[\rho]^*) \vdash \text{free } x . \text{done} + x? \text{acquire}(y) . \text{BusyLock}[x, y]} [\text{T-DEF}]$$

The process (1), combining an instance of the lock and the users *alice* and *carol*, is also well typed. As we will see at the end of Section 3.4, this implies that both *alice* and *carol* are able to acquire the lock, albeit in some unspecified order.

15:16 Mailbox Types for Unordered Interactions

► **Example 21.** In this example we show that the process (2) of Example 6 is ill typed. In order to do so, we assume the global process declaration

$$\text{Future} : (\text{self} : ?(\text{put}[\text{int}] \cdot \text{get}[\text{!reply}[\text{int}]]^*); \emptyset)$$

which can be shown to be consistent with the given definition for **Future**. In the derivation below we use the pattern $F \stackrel{\text{def}}{=} \text{put}[\text{int}] \cdot \text{get}[\rho]^*$ and the types $\tau \stackrel{\text{def}}{=} \text{!put}[\text{int}]$, $\sigma \stackrel{\text{def}}{=} ?(\text{reply}[\text{int}] \cdot \mathbb{1})$ and $\rho \stackrel{\text{def}}{=} \text{!reply}[\text{int}]$:

$$\frac{\frac{\frac{\frac{f : \tau, x : \text{int} \vdash f! \text{put}[x] :: \emptyset}{f : \tau, c : ?\mathbb{1}, x : \text{int} \vdash \text{free } c.f! \text{put}[x]}{f : \tau, c : ?\mathbb{1}, x : \text{int} \vdash \text{free } c.f! \text{put}[x] :: \{c, f\}}{f : \tau, c : \sigma \vdash c? \text{reply}(x). \text{free } c.f! \text{put}[x]}{f : \text{!get}[\rho], c : \rho \vdash f! \text{get}[c] :: \{f, c\} \quad f : \tau, c : \sigma \vdash c? \text{reply}(x). \text{free } c.f! \text{put}[x] :: \{c, f\}}{f : \text{!(get}[\rho] \cdot \text{put}[\text{int}]), c : ?\mathbb{1} \vdash f! \text{get}[c] \mid c? \text{reply}(x). \text{free } c.f! \text{put}[x] :: -}}{f : \text{!}F, c : ?\mathbb{1} \vdash f! \text{get}[c] \mid c? \text{reply}(x). \text{free } c.f! \text{put}[x] :: -}}{f : \text{!}F \vdash (\nu c)(f! \text{get}[c] \mid c? \text{reply}(x). \text{free } c.f! \text{put}[x]) :: -}} \text{[T-SUB]}$$

In attempting this derivation we have implicitly extended the typing rules so that names with type **int** do not contribute in generating any significant dependency. The critical point of the derivation is the application of [T-PAR], where we are composing two parallel processes that yield a circular dependency between c and f . In the process on the left hand side, the dependency $\{f, c\}$ arises because c is sent as a reference in a message targeted to f . In the process on the right hand side, the dependency $\{c, f\}$ arises because there are guards concerning the mailbox c that block an output operation on the mailbox f .

► **Example 22** (non-deterministic choice). The same input action can occur multiple times in the same guarded process. This feature can be used to encode in the mailbox calculus the non-deterministic choice between P_1 and P_2 as the process

$$(\nu a)(\text{free } a.P_1 + \text{free } a.P_2) \tag{4}$$

provided that $\Gamma \vdash P_i :: \varphi_i$ for $i = 1, 2$. That is, P_1 and P_2 must be well typed in the same type environment. Below is the typing derivation for (4)

$$\frac{\frac{\frac{\Gamma \vdash P_1 :: \varphi_1}{\Gamma, a : ?\mathbb{1} \vdash \text{free } a.P_1} \text{[T-FREE]} \quad \frac{\Gamma \vdash P_2 :: \varphi_2}{\Gamma, a : ?\mathbb{1} \vdash \text{free } a.P_2} \text{[T-FREE]}}{\Gamma, a : ?(\mathbb{1} + \mathbb{1}) \vdash \text{free } a.P_1 + \text{free } a.P_2} \text{[T-BRANCH]}}{\frac{\Gamma, a : ?(\mathbb{1} + \mathbb{1}) \vdash \text{free } a.P_1 + \text{free } a.P_2}{\Gamma, a : ?(\mathbb{1} + \mathbb{1}) \vdash \text{free } a.P_1 + \text{free } a.P_2 :: \varphi} \text{[T-GUARD]}}{\frac{\Gamma, a : ?\mathbb{1} \vdash \text{free } a.P_1 + \text{free } a.P_2 :: \varphi}{\Gamma, a : ?\mathbb{1} \vdash \text{free } a.P_1 + \text{free } a.P_2 :: \varphi} \text{[T-SUB]}}{\Gamma \vdash (\nu a)(\text{free } a.P_1 + \text{free } a.P_2) :: (\nu a)\varphi} \text{[T-NEW]}$$

where $\varphi \stackrel{\text{def}}{=} \{a, \text{dom}(\Gamma)\}$. The key step is the application of [T-SUB], which exploits the idempotency of $+$ (in patterns) to rewrite $\mathbb{1}$ as the equivalent pattern $\mathbb{1} + \mathbb{1}$.

3.4 Properties of well-typed processes

In this section we state the main properties enjoyed by well-typed processes. As usual, subject reduction is instrumental for all of the results that follow as it guarantees that typing is preserved by reductions:

► **Theorem 23.** *If Γ is reliable and $\Gamma \vdash P :: \varphi$ and $P \rightarrow Q$, then $\Gamma \vdash Q :: \varphi$.*

Interestingly, Theorem 23 seems to imply that the types of the mailboxes used by a process do not change. In sharp contrast, other popular behavioral typing disciplines (session types in particular), are characterized by a subject reduction result in which types reduce along with processes. Theorem 23 also seems to contradict the observations made earlier concerning the fact that the mailboxes used by a process may have different types (Example 11). The type preservation guarantee assured by Theorem 23 can be explained by recalling that the type environment Γ in a judgment $\Gamma \vdash P :: \varphi$ already takes into account the overall balance between the messages stored into and consumed from the mailbox used by P (see Definition 18). In light of this observation, Theorem 23 simply asserts that well-typed processes are steady state: they never produce more messages than those that are consumed, nor do they ever try to consume more messages than those that are produced.

A practically relevant consequence of Theorem 23 is that, by looking at the type $?E$ of the mailbox a used by a guarded process P (rule $[_T\text{-GUARD}]$), it is possible to determine *bounds* to the number of messages that can be found in the mailbox as P waits for a message to receive. In particular, if every configuration of E contains at most k \mathfrak{m} -tagged atoms, then at runtime a contains at most \mathfrak{m} -tagged messages. As a special case, a mailbox of type $?1$ is guaranteed to be empty and can be statically deallocated. Note that the bounds may change after P receives a message. For example, a free lock is guaranteed to have no **release** messages in its mailbox, and will have at most one when it is busy (see Example 20).

The main result concerns the soundness of the type system, guaranteeing that well-typed (closed) processes are both mailbox conformant and deadlock free (Definitions 4 and 5):

► **Theorem 24.** *If $\emptyset \vdash P :: \varphi$, then P is mailbox conformant and deadlock free.*

Fair termination and junk freedom are not enforced by our typing discipline in general. The usual counterexamples include processes that postpone indefinitely the use of a mailbox with a relevant type. For instance, the \mathfrak{m} message in the well-typed process $(\nu a)(a!_{\mathfrak{m}} \mid X[a])$ where $X(x) \triangleq X[x]$ is never consumed because a is never used for an input operation. Nevertheless, fair termination is guaranteed for the class of finitely unfolding processes:

► **Theorem 25.** *We say that P is finitely unfolding if all maximal reductions of P use $[_R\text{-DEF}]$ finitely many times. If $\emptyset \vdash P :: \varphi$ and P is finitely unfolding, then P is fairly terminating.*

The class of finitely unfolding processes obviously includes all finite processes (those not using process invocations) but also many recursive processes. For example, every process of the form $(\nu a)(a!_{\mathfrak{m}} \mid \dots \mid a!_{\mathfrak{m}} \mid X[a])$ where $X(x) \triangleq x?_{\mathfrak{m}}.X[x] + \mathbf{free} \ x.\mathbf{done}$ is closed, well typed and finitely unfolding regardless of the number of \mathfrak{m} messages stored in a , hence is fairly terminating and junk free by Theorem 25.

4 Examples

In this section we discuss a few more examples that illustrate the expressiveness of the mailbox calculus and of its type system. We consider a variant of the bank account shown in Listing 1 (Section 4.1), the case of master-workers parallelism (Section 4.2) and the encoding of binary sessions extended with forks and joins (Sections 4.3 and 4.4).

4.1 Actors using futures

Many Scala programs combine actors with futures [53]. As an example, Listing 2 shows an alternative version of the `Account` actor in Akka that differs from Listing 1 in the

15:18 Mailbox Types for Unordered Interactions

```

1 class Account(var balance: Double) extends AkkaActor[AnyRef] {
2   override def process(msg: AnyRef) {
3     msg match {
4       case dm: DebitMessage =>
5         balance += dm.amount
6         sender() ! ReplyMessage.ONLY
7       case cm: CreditMessage =>
8         balance -= cm.amount
9         val recipient = cm.recipient.asInstanceOf[ActorRef]
10        val future = ask(recipient, new DebitMessage(self, cm.amount))
11        Await.result(future, Duration.Inf)
12        sender() ! ReplyMessage.ONLY
13      case _: StopMessage => exit()
14      case message =>
15        val ex = new IllegalArgumentException("Unsupported message")
16        ex.printStackTrace(System.err)
17    }
18  }
19 }

```

■ **Listing 2** An Akka actor using futures from the Savina benchmark suite [32].

handling of `CreditMessages` (lines 10–11). The `future` variable created here is initialized asynchronously with the result of the debit operation invoked on `recipient`. To make sure that each transaction is atomic, the actor waits for the variable to be resolved (line 11) before notifying `sender` that the operation has been completed.

This version of `Account` is arguably simpler than the one in Listing 1, if only because the actor has a unique top-level behavior. One way of modeling this implementation of `Account` in the mailbox calculus is to use `Future`, discussed in Example 2. A simpler modeling stems from the observation that `future` in Listing 2 is used for a one-shot synchronization. A future variable with this property is akin to a mailbox from which the value of the resolved variable is retrieved exactly once. Following this approach we obtain the process below:

$$\begin{aligned}
 \text{Account}(self, balance) &\triangleq self?debit(amount, sender). \\
 &\quad sender!reply \mid \text{Account}[self, balance + amount] \\
 &+ self?credit(amount, recipient, sender). \\
 &\quad (\nu future) \left(\begin{array}{l} recipient!debit[amount, future] \mid \\ future?reply.free future. \\ (sender!reply \mid \text{Account}[self, balance - amount]) \end{array} \right) \\
 &+ self?stop.free self.done \\
 &+ self?reply.fail self
 \end{aligned}$$

Compared to the process in Example 3, here the notification from the *recipient* account is received from the mailbox *future*, which is created locally during the handling of the `credit` message. The rest of the process is the same as before. This definition of `Account` and the one in Example 3 can both be shown to be consistent with the declaration

$$\text{Account} : (self : ?(debit[int, \rho]^* \cdot credit[int, !debit[int, \rho], \rho]^* + stop), balance : int; \emptyset)$$

where $\rho \stackrel{\text{def}}{=} !reply$. In particular, the dependencies between *self* and *future* that originate in this version of `Account` are not observable from outside `Account` itself.

The use of multiple mailboxes and the interleaving of blocking operations on them may increase the likelihood of programming mistakes causing mismatched communications and/or deadlocks. However, these errors can be detected by a suitable typing discipline such the one proposed in this paper. Types can also be used to mitigate the runtime overhead resulting from the use of multiple mailboxes. Here, for example, the typing of *future* guarantees that this mailbox is used for receiving a *single* message and that *future* is empty by the time `free future` is performed. A clever compiler can take advantage of this information to statically optimize both the allocation and the deallocation of this mailbox.

4.2 Master-workers parallelism

In this example we model a *master* process that receives tasks to perform from a *client*. For each task, the master creates a pool of *workers* and assigns each worker a share of work. The master waits for all partial results from the workers before sending the final result back to the client and making itself available again. The number of workers may depend on some quantity possibly related to the task to be performed and that is known at runtime only.

Below we define three processes corresponding to the three states in which the master process can be, and we leave *Worker* unspecified:

$$\begin{aligned}
 \text{Available}(self) &\triangleq self?task(client).(vpool)\text{CreatePool}[self, pool, client] \\
 &\quad + \text{free } self.done \\
 \text{CreatePool}(self, pool, client) &\triangleq \text{if } more\ workers\ needed\ \text{then} \\
 &\quad (vworker)(worker!work[pool] \mid \text{Worker}[worker]) \mid \\
 &\quad \text{CreatePool}[self, pool, client] \\
 &\quad \text{else} \\
 &\quad \text{CollectResults}[self, pool, client] \\
 \text{CollectResults}(self, pool, client) &\triangleq pool?result.CollectResults[self, pool, client] \\
 &\quad + \text{free } pool.(client!result \mid \text{Available}[self])
 \end{aligned}$$

The “*if condition then P else Q*” form used here can be encoded in the mailbox calculus and is typed similarly to the non-deterministic choice of Example 22. These definitions can be shown to be consistent with the following declarations:

$$\begin{aligned}
 \text{Available} &: (self : ?task[!result]^*; \emptyset) \\
 \text{CreatePool, CollectResults} &: (self : ?task[!result]^*, pool : ?result^*, client : !result; \\
 &\quad \{pool, self\} \sqcup \{pool, client\})
 \end{aligned}$$

The usual implementation of this coordination pattern requires the programmer to keep track of the number of active workers using a counter that is decremented each time a partial result is collected [32]. When the counter reaches zero, the master knows that all the workers have finished their job and notifies the client. In the mailbox calculus, we can model the counter using a dedicated mailbox *pool* from which the partial results are collected: when *pool* becomes disposable, it means that no more active workers remain.

4.3 Encoding of binary sessions

Session types [27, 29] have become a popular formalism for the specification and enforcement of structured protocols through static analysis. A session is a private communication channel shared by processes that interact through one of its *endpoints*. Each endpoint is associated with a *session type* that specifies the type, direction and order of messages that are supposed

15:20 Mailbox Types for Unordered Interactions

to be exchanged through that endpoint. A typical syntax for session types in the case of *binary sessions* (those connecting exactly two peer processes) is shown below:

$$T, S ::= \mathbf{end} \mid ?[\tau].T \mid ![\tau].T \mid T \& S \mid T \oplus S$$

A session type $?[\tau].T$ describes an endpoint used for receiving a message of type τ and then according to T . Dually, a session type $![\tau].T$ describes an endpoint used for sending a message of type τ and then according to T . An external choice $T \& S$ describes an endpoint used for receiving a selection (either **left** or **right**) and then according to the corresponding continuation (either T or S). Dually, an internal choice $T \oplus S$ describes an endpoint used for making a selection and then according to the corresponding continuation. Communication safety and progress of a binary session are guaranteed by the fact that its two endpoints are linear resources typed by *dual* session types, where the dual of T is obtained by swapping inputs with outputs and internal with external choices.

In this example we encode sessions and session types using mailboxes and mailbox types. We encode a session as a non-uniform, concurrent object. The object is “concurrent” because it is accessed concurrently by the two peers of the session. It is “non-uniform” because its interface changes over time, as the session progresses. The object uses a mailbox $self$ and its behavior is defined by the equations for $\mathbf{Session}_T(self)$ shown below, where T is the session type according to which it must be used by one of the peers:

$$\begin{aligned} \mathbf{Session}_{\mathbf{end}}(self) &\triangleq \mathbf{free} \ self \ . \ \mathbf{done} \\ \mathbf{Session}_{?[\tau].T}(self) &\triangleq self \ ? \ \mathbf{send}(x, s) \ . \ self \ ? \ \mathbf{receive}(r) \ . \\ &\quad (s \ ! \ \mathbf{reply}[self] \mid r \ ! \ \mathbf{reply}[x, self] \mid \mathbf{Session}_T[self]) \\ \mathbf{Session}_{![\tau].T}(self) &\triangleq \mathbf{Session}_{?[\tau].T}[self] \\ \mathbf{Session}_{T \& S}(self) &\triangleq self \ ? \ \mathbf{left}(s) \ . \ self \ ? \ \mathbf{receive}(r) \ . \\ &\quad (s \ ! \ \mathbf{reply}[self] \mid r \ ! \ \mathbf{left}[self] \mid \mathbf{Session}_T[self]) \\ &\quad + \ self \ ? \ \mathbf{right}(s) \ . \ self \ ? \ \mathbf{receive}(r) \ . \\ &\quad (s \ ! \ \mathbf{reply}[self] \mid r \ ! \ \mathbf{right}[self] \mid \mathbf{Session}_S[self]) \\ \mathbf{Session}_{T \oplus S}(self) &\triangleq \mathbf{Session}_{T \& S}[self] \end{aligned}$$

To grasp the intuition behind the definition of $\mathbf{Session}_T(self)$, it helps to recall that each stage of a session corresponds to an interaction between the two peers, where one process plays the role of “sender” and its peer that of “receiver”. Both peers manifest their willingness to interact by storing a message into the session’s mailbox. The receiver always stores a **receive** message, while the sender stores either **send**, **left** or **right** according to T . All messages contain a reference to the mailbox owned by sender and receiver (respectively s and r) where they will be notified once the interaction is completed. A **send** message also carries actual payload x being exchanged. The role of $\mathbf{Session}_T(self)$ is simply to forward each message from the sender to the receiver. The notifications stored in s and r contain a reference to the session’s mailbox so that its type reflects the session’s updated interface corresponding to the rest of the conversation.

Interestingly, the encoding of a session with type T is undistinguishable from that of a session with the dual type \bar{T} . This is natural by recalling that each stage of a session corresponds to a single interaction between the two peers: the order in which they store the respective messages in the session’s mailbox is in general unpredictable but also unimportant, for both messages are necessary to complete each interaction.

As an example, suppose we want to model a system where Alice asks Carol to compute the sum of two numbers exchanged through a session s . Alice and Carol use the session according to the session types $T \stackrel{\text{def}}{=} ![int].![int].?[int].\mathbf{end}$ and $\bar{T} \stackrel{\text{def}}{=} ?[int].?[int].![int].\mathbf{end}$,

respectively. The system is modeled as the process

$$(\nu alice)(\nu carol)(\nu s)(Alice[alice, s] \mid Carol[carol, s] \mid Session_T[s]) \quad (5)$$

where Alice and Carol are defined as follows:

$$\begin{aligned} Alice(self, s) &\triangleq s!send[4, self] \mid self?reply(s). \\ &\quad (s!send[2, self] \mid self?reply(s). \\ &\quad (s!receive[self] \mid self?reply(x, s). \\ &\quad (system!print_int[x] \mid free self.done))) \\ Carol(self, s) &\triangleq s!receive[self] \mid self?reply(x, s). \\ &\quad (s!receive[self] \mid self?reply(y, s). \\ &\quad (s!send[x + y, self] \mid self?reply(s).free self.done)) \end{aligned}$$

The process (5) and the definitions of Alice and Carol are well typed. In general, $Session_T$ is consistent with the declaration $Session_T : ?(\mathcal{E}(T) \cdot \mathcal{E}(\bar{T})); \emptyset$ where $\mathcal{E}(T)$ is the pattern defined by the following equations:

$$\begin{aligned} \mathcal{E}(\mathbf{end}) &\stackrel{\text{def}}{=} \mathbb{1} \\ \mathcal{E}(?\tau.T) &\stackrel{\text{def}}{=} \mathbf{receive}[\mathbf{!reply}[\tau, \mathcal{E}(T)]] \\ \mathcal{E}(\mathbf{!}\tau.T) &\stackrel{\text{def}}{=} \mathbf{send}[\tau, \mathbf{!reply}[\mathcal{E}(T)]] \\ \mathcal{E}(T \& S) &\stackrel{\text{def}}{=} \mathbf{receive}[\mathbf{!(left}[\mathcal{E}(T)] + \mathbf{right}[\mathcal{E}(S)])] \\ \mathcal{E}(T \oplus S) &\stackrel{\text{def}}{=} \mathbf{left}[\mathbf{!reply}[\mathcal{E}(T)]] + \mathbf{right}[\mathbf{!reply}[\mathcal{E}(S)]] \end{aligned}$$

By interpreting both the syntax of T and the definition of $Session_T$ coinductively, it is easy to see that this encoding of binary sessions extends to internal and external choices with arbitrary labels and also to recursive session types. The usual regularity condition ensures that $Session_T$ is finitely representable. Finally, note that the notion of subtyping for encoded session types induced by Definition 9 coincides with the conventional one [21]. Thus, the mailbox type system subsumes a rich session type system where Theorem 24 corresponds to the well-known communication safety and progress properties of sessions.

4.4 Encoding of sessions with forks and joins

We have seen that it is possible to share the output capability on a mailbox among several processes. We can take advantage of this feature to extend session types with forks and joins:

$$T, S ::= \mathbf{end} \mid ?\tau.T \mid \mathbf{!}\tau.T \mid T \& S \mid T \oplus S \mid \mathfrak{A}_{i \in I} \mathbf{m}_i[\tau_i]; T \mid \otimes_{i \in I} \mathbf{m}_i[\tau_i]; T$$

The idea is that the session type $\otimes_{1 \leq i \leq n} \mathbf{m}_i[\tau_i]; T$ describes an endpoint that can be used for sending *all* of the \mathbf{m}_i messages, and then according to T . The difference between $\otimes_{1 \leq i \leq n} \mathbf{m}_i[\tau_i]; T$ and a session type of the form $\mathbf{!}[\tau_1] \dots \mathbf{!}[\tau_n].T$ is that the \mathbf{m}_i messages can be sent by independent processes (for example, by parallel workers) in whatever order instead of by a single sender. Dually, the session type $\mathfrak{A}_{1 \leq i \leq n} \mathbf{m}_i[\tau_i]; T$ describes an endpoint that can be used for collecting *all* of the \mathbf{m}_i messages, and then according to T . Forks and joins are dual to each other, just like simple outputs are dual to simple inputs. The tags \mathbf{m}_i need not be distinct, but equal tags must correspond to equal argument types.

The extension of $Session_T$ to forks and joins is shown below:

$$\begin{aligned} Session_{\otimes_{i \in I} \mathbf{m}_i[\tau_i]; T}(self) &\triangleq self?send(s).self?receive(r).Join_{\otimes_{i \in I} \mathbf{m}_i[\tau_i]; T}[self, s, r] \\ Session_{\mathfrak{A}_{i \in I} \mathbf{m}_i[\tau_i]; T}(self) &\triangleq Session_{\otimes_{i \in I} \mathbf{m}_i[\tau_i]; T}[self] \\ Join_{\otimes_{i \in I} \mathbf{m}_i[\tau_i]; T}(self, s, r) &\triangleq \begin{cases} s!reply[self] \mid r!reply[self] \mid Session_T[self] & \text{if } I = \emptyset \\ self?\mathbf{m}_i(x_i).(r!\mathbf{m}_i[x_i] \mid Join_{\otimes_{i \in I \setminus \{i\}} \mathbf{m}_i[\tau_i]; T}[self, s, r]) & \text{if } i \in I \end{cases} \end{aligned}$$

As in the case of simple interactions, sender and receiver manifest their willingness to interact by storing **send** and **receive** messages into the session's mailbox *self*. At that point, $\text{Join}_T[\text{self}, s, r]$ forwards all the m_i messages coming from the sender side to the receiver side, in some arbitrary order (case $i \in I$). When there are no more messages to forward (case $I = \emptyset$) both sender and receiver are notified with a **reply** message that carries a reference to the session's endpoint, with its type updated according to the rest of the continuation.

The encoding of session types extended to forks and joins follows easily:

$$\begin{aligned} \mathcal{E}(\otimes_{i \in I} m_i[\tau_i]; T) &\stackrel{\text{def}}{=} \text{send}[\text{reply}[\mathcal{E}(T)]] \cdot \prod_{i \in I} m_i[\tau_i] \\ \mathcal{E}(\mathfrak{A}_{i \in I} m_i[\tau_i]; T) &\stackrel{\text{def}}{=} \text{receive}[(\prod_{i \in I} m_i[\tau_i]) \cdot \text{reply}[\mathcal{E}(T)]] \end{aligned}$$

An alternative definition of Join_T that forwards messages as soon as they become available can be obtained by providing suitable input actions for each $i \in I$ instead of picking an arbitrary $i \in I$.

5 Related Work

Concurrent Objects. There are analogies between actors and concurrent objects. Both entities are equipped with a unique identifier through which they receive messages, they may interact with several concurrent clients and their behavior may vary over time, as the entity interacts with its clients. Therefore, static analysis techniques developed for concurrent objects may be applicable to actors (and vice versa). Relevant works exploring behavioral type systems for concurrent objects include those of Najim *et al.* [42], Ravara and Vasconcelos [50], and Puntigam *et al.* [48, 49]. As in the pure actor model, each object has a unique mailbox and the input capability on that mailbox cannot be transferred. The mailbox calculus does not have these restrictions. A notable variation is the model studied by Ravara and Vasconcelos [50], which accounts for *distributed objects*: there can be several copies of an object that react to messages targeted to the same mailbox. Another common trait of these works is that the type discipline focuses on sequences of method invocations and types contain (abstract) information on the internal state of objects and on state transitions. Indeed, types are either finite-state automata [42], or terms of a process algebra [50] or tokens annotated with state transitions [49]. In contrast, mailbox types focus on the content of a mailbox and sequencing is expressed in the type of explicit continuations. The properties enforced by the type systems in these works differ significantly. Some do not consider deadlock freedom [50, 48], others do not account for out-of-order message processing [48]. Details on the enforced properties also vary. For example, the notion of protocol conformance used by Ravara and Vasconcelos [50] allows sending to an object any message that can be handled *by some future state* of the object. In our setting, this would mean allowing to send a **release** message to a free lock if the lock is acquired later on, or allowing to send a **reply** message to an account if the account will later be involved in a transaction.

The most closely related work among those addressing concurrent objects is the one by Crafa and Padovani [15], who propose the use of the Objective Join Calculus as a model for non-uniform, concurrent objects and develop a type discipline that can be used for enforcing concurrent object protocols. While mailbox types have been directly inspired by their types of concurrent objects, there are two major differences with our work. First, in the Objective Join Calculus every object is associated with a single mailbox, just like in the pure actor model [26, 1], meaning that mailboxes are not first class. As a consequence, the types considered by Crafa and Padovani [15] all have an (implicit) output capability. Second, in the Objective Join Calculus input operations are defined atomically on molecules of messages,

whereas in the mailbox calculus messages are received one at a time. As a consequence, the type of a mailbox in the work of Crafa and Padovani [15] is invariant, whereas the same mailbox may have different types at different times in the mailbox calculus (Example 11).

Static analysis of actors. Srinivasan and Mycroft [52] define a type discipline for controlling the ownership of messages and ensuring actor isolation, but consider only uniformly typed mailboxes and do not address mailbox conformance or deadlock freedom.

Christakis and Sagonas [10] describe a static analysis technique whose aim is to ensure matching between send and receive operations in actors. The technique, which is described only informally and does not account for deadlocks, has been implemented in a tool called *dialyzer* and used for the analysis of Erlang programs.

Crafa [14] defines a behavioural type system for actors aimed at ensuring that the order of messages produced and consumed by an actor follows a prescribed protocol. Protocols are expressed as types and describe the behavior of actors rather than the content of the mailboxes they use. Deadlock freedom is not addressed.

Charousset *et al.* [8] describe the design and implementation of CAF, the C++ Actor Framework. Among the features of CAF is the use of *type-safe message passing interfaces* that makes it possible to statically detect a number of protocol violations by piggybacking on the C++ type system. There are close analogies between CAF's message passing interfaces and mailbox types with output capability: both are equipped with a subset semantics and report only those messages that can be stored into the mailbox through a mailbox reference with that type. Charousset *et al.* [8] point out that this feature fosters the decoupling of actors and enables incremental program recompilation.

Giachino *et al.* [22, 40] define a type system for the deadlock analysis of actors making use of implicit futures. Mailbox conformance and deadlocks due to communications are not taken into account.

He *et al.* [25] discuss a typed extension of Akka [24] ensuring that, in well-typed programs, messages sent to an actor are understood by the actor. The type system is not behavioral though, meaning that it is not possible to reason on which *configurations* of messages are legal. In particular, behavior upgrades are monotonic and actors can only increase the type of messages they understand. This is in sharp contrast with our typing discipline, which allows behavior upgrades with possibly unrelated mailbox types (Examples 1 and 2).

Fowler *et al.* [20] formalize channel-based and mailbox-based communicating systems, highlighting the differences between the two models and studying type-preserving encodings between them. Mailboxes in their work are uniformly typed, but the availability of union types make it possible to host heterogeneous values within the same mailbox. This however may lead to a loss of precision in typing. This phenomenon, called *type pollution* by He *et al.* [25] and Fowler *et al.* [20], is observable to some extent also in our typing discipline and can be mitigated by the use of multiple mailboxes (*cf.* Section 4.2). Finally, Fowler *et al.* [20] leave the extension of their investigation to behaviorally-typed language of actors as future work. Our typing discipline is a potential candidate for this investigation and addresses a more general setting thanks to the support for first-class mailboxes.

Sessions and actors. The encoding of binary sessions into actors discussed in Section 4.3 is new and has been inspired by the encoding of binary sessions into the linear π -calculus [35, 16], whereby each message is paired with a continuation. In our case, the continuation, instead of being a fresh (linear) channel, is either the mailbox of the peer or that of the session. This style of communication with explicit continuation passing is idiomatic in the actor model,

which is based on asynchronous communications. The encoding discussed in Section 4.3 can be generalized to multiparty sessions by defining Session_T as a *medium process* through which messages are exchanged between the parties of the session. This idea has been put forward by Caires and Pérez [5] to encode multiparty sessions using binary sessions.

Mostrous and Vasconcelos [41] study a session type system for enforcing ordered dyadic interactions in core Erlang. They use *references* for distinguishing messages pertaining to different sessions, making use of the advanced pattern matching capabilities of Erlang. Their type system guarantees a weaker form of mailbox conformance, whereby junk messages may be present at the end of a computation, and does not consider deadlock freedom. Compared to our encoding of binary sessions, their approach does not require a medium process representing the session itself.

Neykova and Yoshida [43] propose a framework based on multiparty session types for the specification and implementation of actor systems with guarantees on the order of interactions. This approach is applicable when designing an entire system and both the network topology and the communication protocol can be established in advance. Fowler [19] builds upon the work of Neykova and Yoshida to obtain a runtime protocol monitoring mechanism for Erlang. Charalambides *et al.* [7] extend the multiparty session approach with a protocol specification language that is parametric in the number of actors participating in the system. In contrast to these approaches based on multiparty/global session types, our approach ensures mailbox conformance and deadlock freedom of a system compositionally, as the system is assembled out of smaller components, and permits the modeling of systems with a dynamic network topology or with a varying number of interacting processes.

Linear logic. Shortly after its introduction, linear logic has been proposed as a specification language suitable for concurrency. Following this idea, Kobayashi and Yonezawa [37, 38] have studied formal models of concurrent objects and actors based on linear logic. More recently, a direct correspondence between propositions of linear logic and session types has been discovered [6, 55, 39]. There are several analogies between the mailbox type system and the proof system of linear logic. Mailbox types with output capability are akin to positive propositions, with $!0$ and $!1$ respectively playing the roles of 0 and 1 in linear logic and $!(E + F)$ and $!(E \cdot F)$ corresponding to \oplus and \otimes . Mailbox types with input capability are akin to negative propositions, with $?0$ and $?1$ corresponding to \top and \perp and $?(E + F)$ and $?(E \cdot F)$ corresponding to $\&$ and \wp . Rules $[\text{T-FAIL}]$, $[\text{T-FREE}]$ and $[\text{T-BRANCH}]$ have been directly inspired from the rules for \top , \perp and $\&$ in the classical sequent calculus for linear logic. Subtyping corresponds to inverse linear implication and its properties are consistent with those of the logic connectives according to the above interpretation.

Deadlock freedom. There is a vast literature on type systems ensuring deadlock freedom (or stronger properties) of communicating processes and/or concurrent objects. These are based on various mechanisms, including dependency relations between channels, sessions or objects [12, 36, 45], process types [30] and behavioral types [42, 34, 44]. Hüttel *et al.* [29] survey most of these techniques. Our approach is based on the idea of enforcing an acyclic network topology and has been inspired by the session type systems based on linear logic [6, 55, 39]. Interestingly, these works do not require any additional mechanism to ensure deadlock freedom. There are two reasons that call for dependency graphs in our setting. First, the rule $[\text{T-PAR}]$ is akin to a symmetric cut rule. Dependency graphs are necessary to detect mutual dependencies that may consequently arise (Example 21). Second, unlike session endpoints, mailbox references can be used non-linearly. Thus, the multiplicity of dependencies, and not just the presence or lack thereof, is relevant (Example 13).

Igarashi and Kobayashi [30] study a *generic* type system for the π -calculus that allows the enforcement of various safety properties, among which deadlock freedom. Unlike our approach, which is based on typing mailboxes, their approach associates types with processes. Process types collect information about *both* the messages exchanged over channels *as well as* the dependencies between them, potentially achieving better precision in the analysis. This also means, however, that the properties of a system are established by a global check on the type of the system as a whole, which may hinder compositional reasoning. Determining whether their type system subsumes our own is non-trivial and left for future work.

As a final consideration, it is easy to see from the typing rules and the structure of judgments that dependency graphs are completely orthogonal to the other components of the type system. This makes it possible to remove or replace them with more fine-grained mechanisms if desired/appropriate. For example, some of the aforementioned works [36, 45] are able to establish deadlock freedom of some cyclic network topologies. It might be interesting to see whether and how these may be applied in our setting.

6 Concluding Remarks

We have presented a mailbox type system for reasoning about processes that communicate through first-class, unordered mailboxes. The type system enforces mailbox conformance, deadlock freedom and, for a significant class of processes, junk freedom as well. In sharp contrast with session types, mailbox types embody the unordered nature of mailboxes and enable the description of mailboxes concurrently accessed by several processes, abstracting away from the state and behavior of the processes using these mailboxes. The fact that a mailbox may have different types during its lifetime is entirely encapsulated by the typing rules and not apparent from mailbox types themselves. The mailbox calculus subsumes the actor model and allows us to analyze systems with a dynamic network topology and a varying number of processes mixing different concurrency abstractions.

In the associated technical report [17] we informally discuss how to relax the syntax of guarded processes to accommodate actions referring to different mailboxes as well as actions representing timeouts. This extension makes the typing rules for guards more complex to formulate but enhances expressiveness and precision of typing. As a further extension, it is also possible to allow multiple processes to receive messages from the same mailbox.

Concerning further developments, the intriguing analogies between the mailbox type system and linear logic pointed out in Section 5 surely deserve a more thorough investigation. On the practical side, a primary goal is the application of the proposed typing discipline to real-world programs based on the actor model and extensions thereof. In this respect, one promising approach is the development of a tool for the analysis of Java bytecode along the lines of what has already been done for Kilim [52]. Meanwhile, we have derived an algorithmic version of the typing rules (Table 2) and developed a proof-of-concept tool that applies the proposed typing discipline to the mailbox calculus [46]. The fact that each occurrence of a name might be typed differently calls for a non-trivial amount of type inference as well. To this aim, we make use of *pattern variables* to denote unknown patterns, we *generate constraints* involving these variables from the structure of the process being analyzed, and finally we look for a *solution* of the obtained constraints. This latter phase requires solving systems of inequations in a commutative Kleene algebra, for which we appeal to a particular instance of Newtonian program analysis [18] first introduced by Hopkins and Kozen [28].

References

- 1 Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.
- 2 Davide Ancona, Viviana Bono, Mario Bravetti, Joana Campos, Giuseppe Castagna, Pierre-Malo Deniérou, Simon J. Gay, Nils Gesbert, Elena Giachino, Raymond Hu, Einar Broch Johnsen, Francisco Martins, Viviana Mascardi, Fabrizio Montesi, Rumyana Neykova, Nicholas Ng, Luca Padovani, Vasco T. Vasconcelos, and Nobuko Yoshida. Behavioral Types in Programming Languages. *Foundations and Trends in Programming Languages*, 3:95–230, 2016. doi:10.1561/25000000031.
- 3 Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2013.
- 4 Janusz A. Brzozowski. Derivatives of Regular Expressions. *Journal of ACM*, 11(4):481–494, 1964. doi:10.1145/321239.321249.
- 5 Luís Caires and Jorge A. Pérez. Multiparty session types within a canonical binary theory, and beyond. In *Proceedings of FORTE’16*, LNCS 9688, pages 74–95. Springer, 2016. doi:10.1007/978-3-319-39570-8_6.
- 6 Luís Caires and Frank Pfenning. Session Types as Intuitionistic Linear Propositions. In *Proceedings of CONCUR’10*, LNCS 6269, pages 222–236. Springer, 2010. doi:10.1007/978-3-642-15375-4_16.
- 7 Minas Charalambides, Peter Dinges, and Gul A. Agha. Parameterized, concurrent session types for asynchronous multi-actor interactions. *Science of Computer Programming*, 115-116:100–126, 2016. doi:10.1016/j.scico.2015.10.006.
- 8 Dominik Charousset, Raphael Hiesgen, and Thomas C. Schmidt. Revisiting actor programming in C++. *Computer Languages, Systems & Structures*, 45:105–131, 2016. doi:10.1016/j.cl.2016.01.002.
- 9 Arghya Chatterjee, Branko Gvoka, Bing Xue, Zoran Budimlic, Shams Imam, and Vivek Sarkar. A distributed selectors runtime system for java applications. In *Proceedings of PPPJ’16*, pages 3:1–3:11. ACM, 2016. doi:10.1145/2972206.2972215.
- 10 Maria Christakis and Konstantinos Sagonas. Detection of asynchronous message passing errors using static analysis. In *Proceedings of PADL’11*, LNCS 6539, pages 5–18. Springer, 2011. doi:10.1007/978-3-642-18378-2_3.
- 11 John Conway. *Regular Algebra and Finite Machines*. William Clowes & Sons Ltd, 1971.
- 12 Mario Coppo, Mariangiola Dezani-Ciancaglini, Nobuko Yoshida, and Luca Padovani. Global Progress for Dynamically Interleaved Multiparty Sessions. *Mathematical Structures in Computer Science*, 26:238–302, 2016. doi:10.1017/S0960129514000188.
- 13 Bruno Courcelle. Fundamental Properties of Infinite Trees. *Theoretical Computer Science*, 25:95–169, 1983. doi:10.1016/0304-3975(83)90059-2.
- 14 Silvia Crafa. Behavioural types for actor systems. Technical Report 1206.1687, arXiv, 2012. URL: <http://arxiv.org/abs/1206.1687>.
- 15 Silvia Crafa and Luca Padovani. The Chemical Approach to Typestate-Oriented Programming. *ACM Transactions on Programming Languages and Systems*, 39:13:1–13:45, 2017. doi:10.1145/3064849.
- 16 Ornela Dardha, Elena Giachino, and Davide Sangiorgi. Session types revisited. *Information and Computation*, 256:253–286, 2017. doi:10.1016/j.ic.2017.06.002.
- 17 Ugo de’Liguoro and Luca Padovani. Mailbox types for unordered interactions. *CoRR*, abs/1801.04167, 2018. arXiv:1801.04167.
- 18 Javier Esparza, Stefan Kiefer, and Michael Luttenberger. Newtonian program analysis. *Journal of the ACM*, 57(6):33:1–33:47, 2010. doi:10.1145/1857914.1857917.
- 19 Simon Fowler. An Erlang implementation of multiparty session actors. In *Proceedings of ICE’16*, EPTCS 223, pages 36–50, 2016. doi:10.4204/EPTCS.223.3.

- 20 Simon Fowler, Sam Lindley, and Philip Wadler. Mixing metaphors: Actors as channels and channels as actors. In *Proceedings of ECOOP'17*, LIPIcs 74, pages 11:1–11:28, 2017. doi:10.4230/LIPIcs.ECOOP.2017.11.
- 21 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.
- 22 Elena Giachino, Ludovic Henrio, Cosimo Laneve, and Vincenzo Mastandrea. Actors may synchronize, safely! In *Proceedings PPDP'16*, pages 118–131. ACM, 2016. doi:10.1145/2967973.2968599.
- 23 Philipp Haller. On the integration of the actor model in mainstream technologies: the scala perspective. In *Proceedings of AGERE! 2012*, pages 1–6. ACM, 2012. doi:10.1145/2414639.2414641.
- 24 Philipp Haller and Frank Sommers. *Actors in Scala - concurrent programming for the multi-core era*. Artima, 2011.
- 25 Jiansen He, Philip Wadler, and Philip Trinder. Typecasting actors: From akka to takka. In *Proceedings of the Fifth Annual Scala Workshop (SCALA'14)*, pages 23–33. ACM, 2014. doi:10.1145/2637647.2637651.
- 26 Carl Hewitt, Peter Bishop, and Richard Steiger. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *Proceedings of IJCAI'73*, pages 235–245. William Kaufmann, 1973.
- 27 Kohei Honda. Types for Dyadic Interaction. In *Proceedings of CONCUR'93*, volume LNCS 715, pages 509–523. Springer, 1993. doi:10.1007/3-540-57208-2_35.
- 28 Mark W. Hopkins and Dexter Kozen. Parikh's Theorem in Commutative Kleene Algebra. In *Proceedings of LICS'99*, pages 394–401. IEEE, 1999. doi:10.1109/LICS.1999.782634.
- 29 Hans Hüttel, Ivan Lanese, Vasco T. Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Deniérou, Dimitris Mostrous, Luca Padovani, António Ravara, Emilio Tuosto, Hugo Torres Vieira, and Gianluigi Zavattaro. Foundations of Session Types and Behavioural Contracts. *ACM Computing Surveys*, 49(1):3:1–3:36, 2016. doi:10.1145/2873052.
- 30 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.
- 31 Shams Mahmood Imam and Vivek Sarkar. Integrating task parallelism with actors. *SIGPLAN Notices*, 47(10):753–772, 2012. doi:10.1145/2398857.2384671.
- 32 Shams Mahmood Imam and Vivek Sarkar. Savina - an actor benchmark suite: Enabling empirical evaluation of actor libraries. In *Proceedings of AGERE! 2014*, pages 67–80. ACM, 2014. doi:10.1145/2687357.2687368.
- 33 Shams Mahmood Imam and Vivek Sarkar. Selectors: Actors with multiple guarded mailboxes. In *Proceedings of AGERE! 2014*, pages 1–14. ACM, 2014. doi:10.1145/2687357.2687360.
- 34 Naoki Kobayashi. A Type System for Lock-Free Processes. *Information and Computation*, 177(2):122–159, 2002. doi:10.1006/inco.2002.3171.
- 35 Naoki Kobayashi. Type systems for concurrent programs. Technical report, Tohoku University, 2007. Short version appeared in 10th Anniversary Colloquium of UNU/IIST, 2002. URL: <http://www.kb.ecei.tohoku.ac.jp/~koba/papers/tutorial-type-extended.pdf>.
- 36 Naoki Kobayashi and Cosimo Laneve. Deadlock analysis of unbounded process networks. *Information and Computation*, 252:48–70, 2017. doi:10.1016/j.ic.2016.03.004.
- 37 Naoki Kobayashi and Akinori Yonezawa. Type-theoretic foundations for concurrent object-oriented programming. In *Proceedings of OOPSLA'94*, pages 31–45. ACM, 1994. doi:10.1145/191080.191088.
- 38 Naoki Kobayashi and Akinori Yonezawa. Asynchronous communication model based on linear logic. *Formal Aspects of Computing*, 7(2):113–149, 1995. doi:10.1007/BF01211602.


- 39 Sam Lindley and J. Garrett Morris. A semantics for propositions as sessions. In *Proceedings of ESOP'15*, LNCS 9032, pages 560–584. Springer, 2015. doi:10.1007/978-3-662-46669-8_23.
- 40 Vincenzo Mastandrea. Deadlock analysis with behavioral types for actors. In *Proceedings of ICTCS'16*, volume 1720 of *CEUR Workshop Proceedings*, pages 257–262, 2016. URL: <http://ceur-ws.org/Vol-1720/short7.pdf>.
- 41 Dimitris Mostrous and Vasco T. Vasconcelos. Session typing for a featherweight Erlang. In *Proceedings of COORDINATION'11*, LNCS 6721, pages 95–109. Springer, 2011. doi:10.1007/978-3-642-21464-6_7.
- 42 Elie Najm, Abdelkrim Nimour, and Jean-Bernard Stefani. Guaranteeing liveness in an object calculus through behavioural typing. In *Proceedings of FORTE'99*, volume 156, pages 203–221. Kluwer, 1999.
- 43 Rumyana Neykova and Nobuko Yoshida. Multiparty session actors. *Logical Methods in Computer Science*, 13(1), 2017. doi:10.23638/LMCS-13(1:17)2017.
- 44 Luca Padovani. Deadlock and Lock Freedom in the Linear π -Calculus. In *Proceedings of CSL-LICS'14*, pages 72:1–72:10. ACM, 2014. doi:10.1145/2603088.2603116.
- 45 Luca Padovani. Deadlock-Free Typestate-Oriented Programming. *Programming Journal*, 2, 2018. doi:10.22152/programming-journal.org/2018/2/15.
- 46 Luca Padovani. MC², the Mailbox Calculus Checker, 2018. URL: <http://www.di.unito.it/~padovani/Software/MCC/index.html>.
- 47 Benjamin C. Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, 6(5):409–453, 1996.
- 48 Franz Puntigam. Strong types for coordinating active objects. *Concurrency and Computation: Practice and Experience*, 13(4):293–326, 2001. doi:10.1002/cpe.570.
- 49 Franz Puntigam and Christof Peter. Types for active objects with static deadlock prevention. *Fundamenta Informaticae*, 48(4):315–341, 2001. URL: <http://content.iospress.com/articles/fundamenta-informaticae/fi48-4-02>.
- 50 António Ravara and Vasco T. Vasconcelos. Typing non-uniform concurrent objects. In *Proceedings of CONCUR'00*, LNCS 1877, pages 474–488. Springer, 2000. doi:10.1007/3-540-44618-4_34.
- 51 Davide Sangiorgi and David Walker. *The Pi-Calculus - A theory of mobile processes*. Cambridge University Press, 2001.
- 52 Sriram Srinivasan and Alan Mycroft. Kilim: Isolation-typed actors for java. In *Proceedings of ECOOP'08*, LNCS 5142, pages 104–128. Springer, 2008. doi:10.1007/978-3-540-70592-5_6.
- 53 Samira Tasharofi, Peter Dinges, and Ralph E. Johnson. Why do scala developers mix the actor model with other concurrency models? In *Proceedings of ECOOP'13*, LNCS 7920, pages 302–326. Springer, 2013. doi:10.1007/978-3-642-39038-8_13.
- 54 Carlos A. Varela and Gul Agha. Programming dynamically reconfigurable open systems with SALSA. *SIGPLAN Notices*, 36(12):20–34, 2001. doi:10.1145/583960.583964.
- 55 Philip Wadler. Propositions as sessions. *Journal of Functional Programming*, 24(2-3):384–418, 2014. doi:10.1017/S095679681400001X.

Accelerating Dynamically-Typed Languages on Heterogeneous Platforms Using Guards Optimization

Mohaned Qunaibit

University of California, Irvine

m.qunaibit@uci.edu

 <https://orcid.org/0000-0001-6759-7890>

Stefan Brunthaler

National Cyber Defense Research Institute CODE, Munich, and SBA Research

brunthaler@unibw.de

Yeoul Na

University of California, Irvine

yeouln@uci.edu

Stijn Volckaert

University of California, Irvine

stijnv@uci.edu

Michael Franz

University of California, Irvine

franz@uci.edu

Abstract

Scientific applications are ideal candidates for the “heterogeneous computing” paradigm, in which parts of a computation are “offloaded” to available accelerator hardware such as GPUs. However, when such applications are written in dynamic languages such as Python or R, as they increasingly are, things become less straightforward. The same flexibility that makes these languages so appealing to programmers also significantly complicates the problem of automatically and transparently partitioning a program’s execution between a CPU and available accelerator hardware without having to rely on programmer annotations.

A common way of handling the features of dynamic languages is by introducing *speculation* in conjunction with *guards* to ascertain the validity of assumptions made in the speculative computation. Unfortunately, a single guard violation during the execution of “offloaded” code may result in a huge performance penalty and necessitate the complete re-execution of the offloaded computation. In the case of dynamic languages, this problem is compounded by the fact that a full compiler analysis is not always possible ahead of time.

This paper presents MEGAGUARDS, a new approach for speculatively executing dynamic languages on heterogeneous platforms in a fully automatic and transparent manner. Our method translates each target loop into a single *static region* devoid of any dynamic type features. The dynamic parts are instead handled by a construct that we call a *mega guard* which checks all the speculative assumptions ahead of its corresponding static region. Notably, the advantage of MEGAGUARDS is not limited to heterogeneous computing; because it removes guards from compute-intensive loops, the approach also improves sequential performance.

We have implemented MEGAGUARDS along with an automatic loop parallelization backend in ZipPy, a Python Virtual Machine. The results of a careful and detailed evaluation reveal very significant speedups of an order of magnitude on average with a maximum speedup of up to two orders of magnitudes when compared to the original ZipPy performance as a baseline. These results demonstrate the potential for applying heterogeneous computing to dynamic languages.



© Mohaned Qunaibit, Stefan Brunthaler, Yeoul Na, Stijn Volckaert, and Michael Franz; licensed under Creative Commons License CC-BY

32nd European Conference on Object-Oriented Programming (ECOOP 2018).

Editor: Todd Millstein; Article No. 16; pp. 16:1–16:29

Leibniz International Proceedings in Informatics



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

2012 ACM Subject Classification Software and its engineering → Interpreters, Software and its engineering → Just-in-time compilers, Software and its engineering → Dynamic compilers

Keywords and phrases Type Specialization, Guards Optimization, Automatic Heterogeneous Computing, Automatic Parallelism

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2018.16

Funding This material is based upon work partially supported by the Defense Advanced Research Projects Agency (DARPA) under contracts FA8750-15-C-0124 and FA8750-15-C-0085, by the National Science Foundation under awards CNS-1513837 and CNS-1619211, and by the Office of Naval Research under award N00014-17-1-2782. The competence center SBA Research (SBA-K1) is funded within the framework of COMET – Competence Centers for Excellent Technologies by BMVIT, BMDW, and the federal state of Vienna, managed by the FFG. We also gratefully acknowledge a gift from Oracle Corporation. Mohaned Qunaibit was supported by MOHE’s Graduate Studies Scholarship. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the Defense Advanced Research Projects Agency (DARPA), its Contracting Agents, the National Science Foundation, the Office for Naval Research, or any other agency of the U.S. Government.

1 Motivation

Heterogeneous computing, in which the execution of a program is shared between a CPU and other hardware such as a GPU or dedicated accelerator chips, is gaining in importance. By judiciously “offloading” some of the computations to available acceleration hardware, performance can in many cases be raised far beyond single threaded CPU capabilities.

Unfortunately, writing programs for heterogeneous computing is difficult. Some researchers are focusing on “transparent” approaches, in which computations are distributed to hardware accelerators fully automatically, while others are concentrating on a more manual approach in which programmers guide this process explicitly through specific programming language constructs or compiler-directed annotations. As Hager et al. noted in 2015 [26], for the long term it is still an open question

[...] whether accelerators will be automatically invoked by compilers and runtime systems, [...] or be explicitly managed by application programmers.

Looking back on decades of successful research on automated compiler optimizations, we can state that “transparent” approaches that perform optimizations without programmer intervention and that can automatically adapt to changes in available accelerator hardware are clearly preferable to manual approaches that might require program re-writing each time that the hardware is changed. Accordingly, much of the overall research on heterogeneous computing has focused on this automation aspect. A closer look at this prior work, however, reveals that most research on automating heterogeneous computing has centered on *statically-typed* programming languages.

When looking at existing research on heterogeneous computing for *dynamic* programming languages such as Python, we find that the emphases are reversed: most of the work in the dynamic languages domain focuses on explicit manual management of accelerators through programmer-directed addition of source code annotations and/or the use of idiosyncratic libraries. Getting to know these annotations and libraries is a time-consuming obstacle that may prevent programmers from re-writing their code to benefit from heterogeneous

programming. In addition, customizing code to adhere to one specific library vs another naturally inhibits a program’s portability. Moreover, these libraries and annotations often force programmers to abandon the flexibility afforded by dynamic typing.

Automating heterogeneous computing is challenging for dynamic languages because the dynamic types of objects may change at any time during program execution. Consider, for example, an operation that changes from an integer addition to a string concatenation as a result of a type change in an underlying operand. In a dynamic compilation environment, such code will probably first be optimized to an integer addition. When then the type change is captured by a runtime type check, i.e., via a *guard*, the existing optimization is invalidated and the execution falls back to the interpreter or a less optimized version of the code. Eventually, it may then be optimized again for the new type.

Now consider what happens when such mis-speculation happens during the execution of a piece of code that has been offloaded to a hardware acceleration device; we will call such pieces of code “kernels” in the remainder of this paper. In case of a mis-speculation, the existing kernel may become invalid. But because the kernel was executing on a device external to the CPU, the performance penalties may be much higher than merely dropping back into an interpreter or a lower level of optimization. In the worst case, it may not even be possible to salvage the results computed so far, so that re-execution of the whole kernel will be required.

In general, transparent offloading may require complex static analyses such as points-to analysis to check dependencies across loop iterations. The additional code required to handle mis-speculations complicates the program to analyze further, making adoption of such static analysis techniques to dynamic compilation very difficult.

To overcome these challenges, we propose MEGAGUARDS¹, which removes the obstacles in dynamic languages that prevent compute-intensive loops to be transparently offloaded to GPUs or other acceleration devices. MEGAGUARDS translates a loop as a *static region* in which type changes or type mis-speculations do not exist. The key insight and novelty of MEGAGUARDS is how it guarantees that the offloaded code does not encounter type changes or type mis-speculations. To this end, MEGAGUARDS conducts a type stability analysis for loops to see if all the guards can be safely moved outside of the loops. If so, MEGAGUARDS removes all the guards from the loop and constructs a single guard, i.e., a “*mega guard*,” which checks all the speculative assumptions ahead of the loop. This way the loop itself can be seen as the static region. MEGAGUARDS then offloads among the stabilized loops if it can prove that the loop does not have any cross-iterational dependencies. The advantage of MEGAGUARDS, however, is not limited to enabling offloading. Since it removes guards from the loop, MEGAGUARDS also improves performance on a single threaded CPU.

We implemented MEGAGUARDS in ZipPy [61], a modern Python 3 implementation targeting the Java Virtual Machine (JVM). ZipPy relies on the Truffle framework to optimize interpreted programs on the CPU [58]. To determine parallelizable loops, we perform a bounds check optimization and conduct dependence analysis by leveraging the polyhedral model [25]. MEGAGUARDS dynamically translates parallel loops into OpenCL code, which it then executes on the fastest acceleration device available on the target system. MEGAGUARDS significantly improves the performance of data-parallel applications over sequential execution of Python. Even if we cannot parallelize a loop, we still translate it to a guard-less AST, which also improves the sequential performance.

¹ Our software will be publicly available at <https://github.com/seuresystemslab/zippy-megaguards>

In summary, the contribution of this paper are as follows:

- We introduce a novel technique, MEGAGUARDS, that eliminates type speculation inside of loops to efficiently offload speculative code to kernels (Section 3.3.1). Eliminating speculation inside of loops also improves sequential performance (Section 3.5).
- We describe the design and implementation of MEGAGUARDS, a Python-based system that transparently offloads data-parallel loops to an acceleration device, such as a GPU, without requiring code rewriting or annotations from the programmer (Section 3).
- We report results of a careful and detailed evaluation (Section 4). Specifically, our experiments indicate that MEGAGUARDS offers:
 - **Performance:** Our measurements show that MEGAGUARDS (i) performs within $2.82\times$ of the average performance of handwritten, native OpenCL C/C++ implementations on the GPU, and (ii) yields substantial speedups when compared with existing Python implementations (with average speedups exceeding $84\times$).
 - **Implementation Efficiency:** By way of optimizing pure Python code in an automatic and transparent manner, MEGAGUARDS removes the necessity for the labor-intensive task of manually rewriting code. To quantify these gains in implementation efficiency, we measured reductions in (i) lines of code, and (ii) McCabe’s cyclomatic complexity. Our results indicate average reductions by about three quarters in both dimensions.

2 Background

2.1 Heterogeneous Programming Frameworks

Programming in a heterogeneous computing environment is highly challenging because heterogeneous programming frameworks (e.g., CUDA [42] and OpenCL [48]) have steep learning curves and requiring knowledge of the inner workings of the GPU.

To alleviate this problem for **statically-typed languages**, researchers have proposed transformations that map existing parallel paradigms for the CPU to run on the GPU [44, 24, 56]. Others proposed libraries and lambda expressions [22, 28, 46, 47, 36, 7] to automatically generate GPU code. Some techniques automatically parallelize sequential loops and run them on GPUs [35, 3]. New languages such as Lime [18, 2] implicitly perform parallel computations on GPUs.

Dynamically-typed languages have fewer options to simplify GPU programming and must typically resort to external APIs for generating OpenCL or CUDA code. Python programmers, for example, can use libraries such as Numba to design kernel code targeting CUDA. In-depth knowledge of the GPU’s architecture and manual data management remains necessary to use these libraries.

2.2 Interpreters and Virtual Machines

The fact that variable types can change at any moment in dynamically-typed languages hinders ahead-of-time optimization. The rate at which variable types change in practice is, however, usually minimal [16, 57]. This observation has inspired various specialization approaches that minimize the interpreter’s type-checking overhead [9, 8, 59, 55, 1, 63]. In our work, we leverage specialized types to eliminate all type-checking in the generated OpenCL code.

Truffle [58], the self-optimizing runtime system we use in MEGAGUARDS, performs specialization via automatic node rewriting on an abstract syntax tree (AST). Truffle speculatively replaces generic AST nodes, which are capable of operating on variables of any

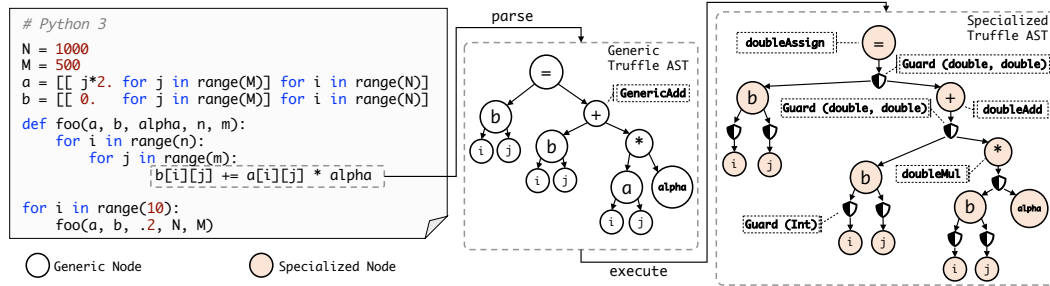


Figure 1 Node specialization in Truffle.

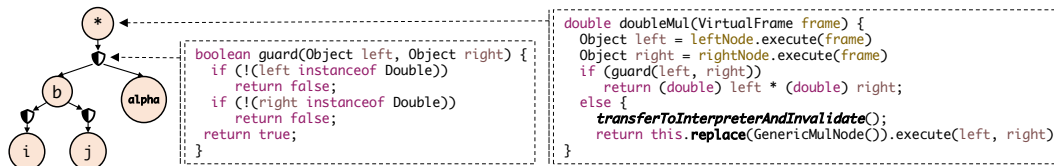


Figure 2 Handling mis-speculation using type guards.

data type, with nodes that are specialized for a specific data type (Figure 1). This speculation approach facilitates just-in-time compilation of Truffle’s hosted languages, which include ZipPy, FastR, and TruffleRuby. When a Truffle AST reaches a stable state, the Truffle framework invokes the Graal just-in-time compiler [19, 60] to further optimize the Truffle AST through *partial evaluation* and to compile the AST into highly optimized machine code.

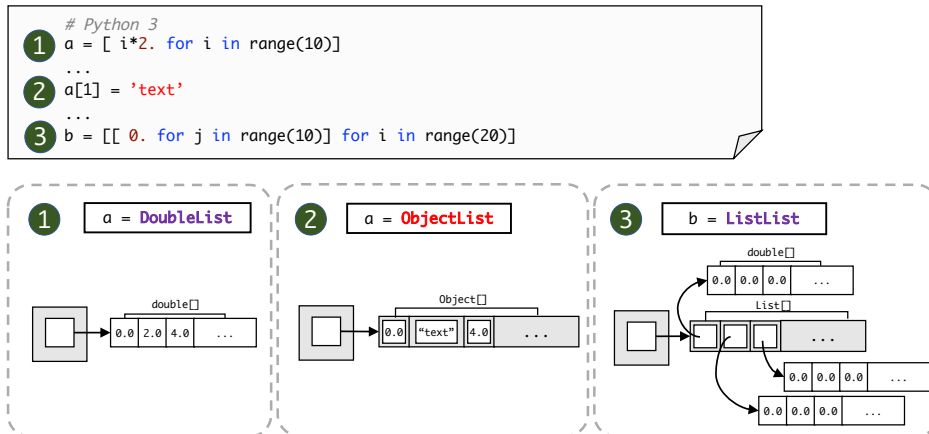
To preserve the correctness of the program execution, Truffle must be able to handle mis-speculation. Figure 2 shows how Truffle embeds type guards into the specialized AST. Guards verify that the specialized input types for a node match the expected types, and trigger deoptimization if they detect a mismatch. When a node’s return type mismatches the specialized data type, an exception is thrown and Truffle also proceeds to deoptimize the node. During deoptimization, Truffle discards the specialized node and replaces it by a generic node.

ZipPy, the Python 3 VM we use in MEGAGUARDS, is built on top of Truffle. ZipPy’s type system specializes objects based on their content. In Figure 3, we see several examples of type specialization in ZipPy. At ①, the program creates a list `a` containing items of the same type. ZipPy internally specializes this list to be of type `DoubleList`. At ②, one of the list items is replaced by a value of a different type. Here, ZipPy generalizes the list to be of type `ObjectList`. At ③, the program creates a multi-dimensional list `b`. In this case, ZipPy specializes the nested lists to be of type `DoubleList`. ZipPy stores variables values in a virtual frame corresponding to the context that variables have been created in. This virtual frame is usually referred to as the *context frame*. Each variable in the context frame maintains its specialized type. This object layout design assists Truffle specialization process and minimizes node type generalization (i.e., deoptimization).

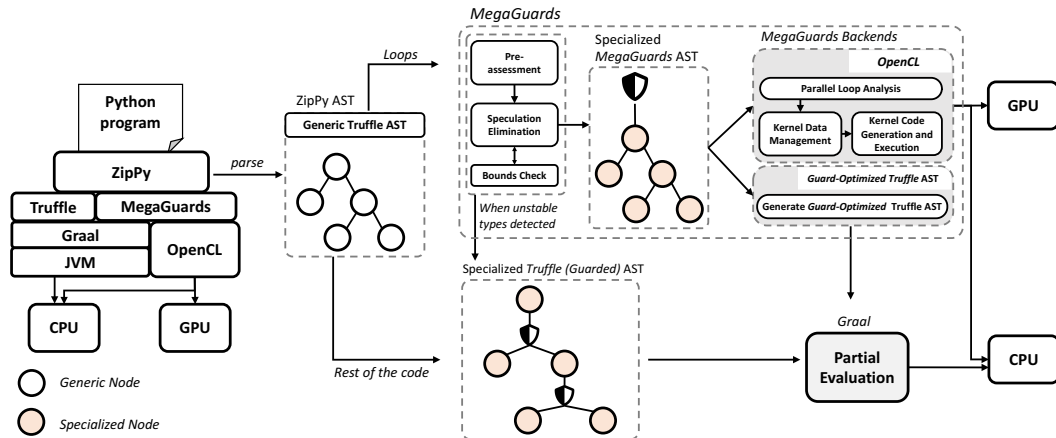
3 The MegaGuards System

3.1 Overview

Figure 4 shows how MEGAGUARDS fits into the ZipPy ecosystem. Conceptually, MEGAGUARDS works as follows. First, whenever the interpreter executes a loop with an identifiable index expression, such as the `for i in range(n)` statement in Figure 1, MEGAGUARDS



■ Figure 3 ZipPy type specialization.



■ Figure 4 ZipPy+MEGAGUARDS System Overview.

determines if the loop is a potential candidate for offloading to an accelerator device (Section 3.2). If the loop is a suitable candidate, MEGAGUARDS analyzes if the loop can be stabilized using our type stability analysis. If so, MEGAGUARDS eliminates all type checks from the loop and creates a *mega guard* which checks all the speculative assumptions outside the loop (Section 3.3). MEGAGUARDS then performs a bounds check optimization analysis and marks operations that require run-time checks (Section 3.3.3). After that, MEGAGUARDS performs a dependence analysis to see if the loop iterations are independent of each other and thus can be safely offloaded (Section 3.4.1). MEGAGUARDS then optimizes the AST of the parallelizable loop and translates it into OpenCL kernel code (Section 3.4.2). Finally, MEGAGUARDS compiles the OpenCL kernel and adaptively selects the best acceleration device to offload (Section 3.4.5). If MEGAGUARDS finds that a loop is not a candidate for offloading, MEGAGUARDS will force ZipPy to execute that loop on top of Graal, a dynamic compiler. If the loop is proven to be type stable, however, MEGAGUARDS will still perform the mega guard optimization.

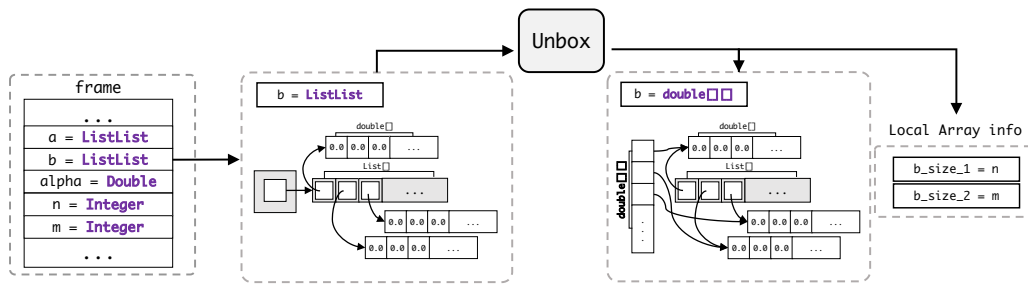


Figure 5 MEGAGUARDS unboxing process.

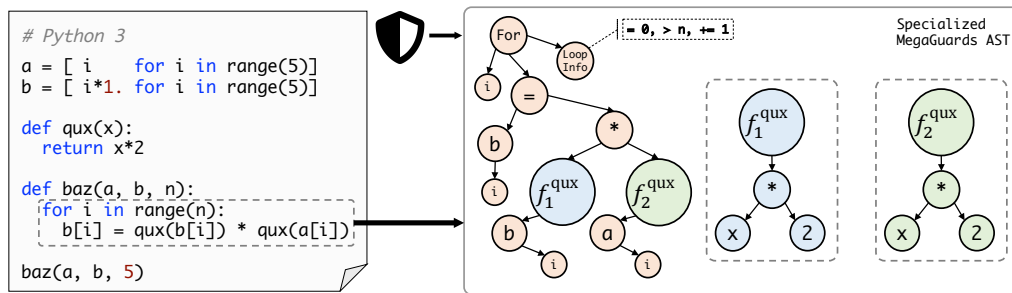


Figure 6 MEGAGUARDS specialized AST with inter-procedural invocations.

3.2 Lightweight Pre-assessment

MEGAGUARDS begins its analysis when the interpreter reaches a loop with an identifiable index expression that has an explicit number of iterations. MEGAGUARDS considers the loop a suitable candidate for offloading if its step sizes are constant.

For suitable candidate loops, MEGAGUARDS traverses the AST sub-tree constituting the loop to ensure that all the instructions in the loop are supported by the OpenCL framework.

3.3 Guards Optimization

Truffle uses type guards and exceptions to handle mis-speculations, as shown in Section 2.2. MEGAGUARDS hoists type, bounds, and overflow checks out of a loop to translate the loop into a *static region*. This way these checks are performed *before* that loop is executed. To this end, MEGAGUARDS performs type stability analysis for each AST node, identifies all the input data to be type-guarded, and generates specialized, strongly-typed ASTs. Moreover, MEGAGUARDS analyzes array subscripts and arithmetic operations in affine expressions to optimize bounds and overflow checks. The nodes in a specialized AST do not contain type checks but may contain bounds and arithmetic overflow checks that MEGAGUARDS is unable to optimize (see Section 3.3.3).

3.3.1 Type Stability Analysis

MEGAGUARDS now assesses the type stability of the loop. We say a loop is type-stable if we can deduce a single data type for each node and can guarantee all potential type changes can only result from outside the loop, not from the inside. MEGAGUARDS performs this type stability analysis before executing or profiling the loop but it leverages type feedback information of live-in variables available in the context frame maintained by ZipPy (see

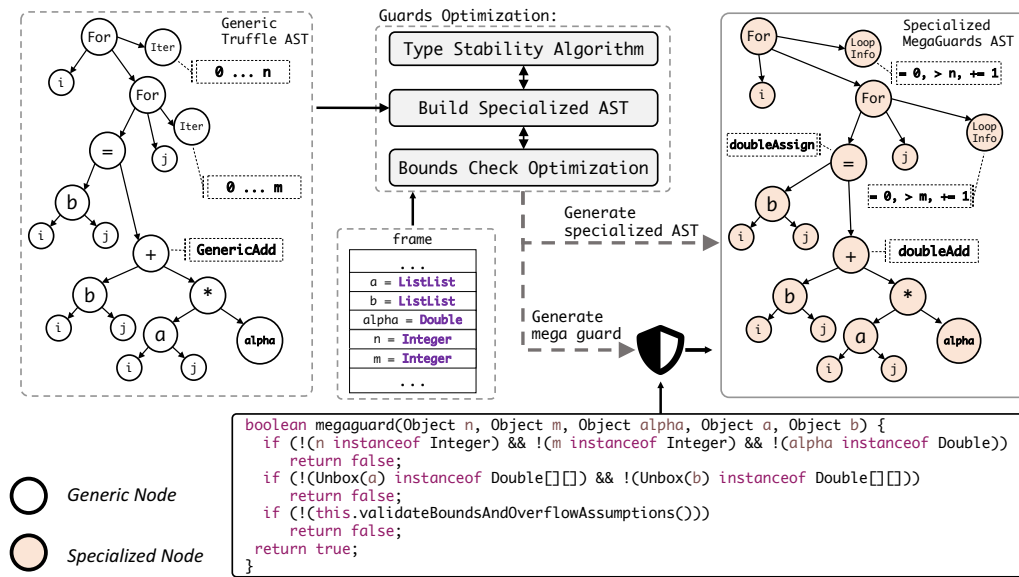
ALGORITHM 1: Type Stability Analysis Algorithm.

```

Function DominantType(left, right)
  Result: Return the strongest data type (e.g., (double, long) → double)
  if left == None then return right;
  else if right == None then return left;
  else if left > right then return left;
  else return right;
end

Function NodeVisitor(node) /* Depth-First tree traversal */
  Result: Return data type of the tree
  op ← node.getOp()
  if op == AssignmentNode then
    leftDataType ← NodeVisitor(node.getLeftChild())
    rightDataType ← NodeVisitor(node.getRightChild())
    if leftDataType == rightDataType then
      | return leftDataType
    else /* Possible data type change */
      | Exit MegaGuards and transfer to interpreter
    end
  else if op == IfElseAssignmentNode then
    /* e.g.  $\alpha = (1 \text{ if } \beta > 0 \text{ else } 2)$  */
    leftDataType ← NodeVisitor(node.getLeftChild())
    thenDataType ← NodeVisitor(node.getThenChild())
    elseDataType ← NodeVisitor(node.getElseChild())
    if thenDataType == elseDataType and leftDataType == thenDataType then
      | return leftDataType
    else /* Possible data type change */
      | Exit MegaGuards and transfer to interpreter
    end
  else
    if node is User-Defined Function Call then
      returnDataType ← None
      Enter new Scope
      foreach argument in node.getArguments() do
        /* assign argument data types to the function parameter */
        parameter ← NodeVisitor(argument)
      end
      returnDataType ← NodeVisitor(node.getFunctionRoot())
      /* assert all return sites have the same data type */
      Exit Scope
      return returnDataType
    else /* Other nodes, e.g., binary arithmetic, return, etc. */
      currentDataType ← None
      foreach child in node.getChildren() do
        childDataType ← NodeVisitor(child)
        currentDataType ← DominantType(childDataType, currentDataType)
      end
      return currentDataType
    end
  end
end

```



■ **Figure 7** MEGAGUARDS specialized AST build process.

Section 2.2). In figure 5, MEGAGUARDS runs an unboxing pass on variables of generic boxed types (e.g., object lists) to augment the context frame with more precise information. If MEGAGUARDS finds multiple types within in the same boxed data structure (e.g., a list that stores both strings and integers), it will mark that structure as type-unstable in the context frame.

After unboxing, MEGAGUARDS runs Algorithm 1 on each AST node in a loop body to infer the type of each node, and to verify the type stability of each statement. The main method in the algorithm, `NodeVisitor`, traverses the loop's AST statements in depth-first order, propagating the data types from the augmented context frame through each operation. For assignment operations, represented by `AssignmentNode` nodes in the AST, our algorithm consults the context frame to check if the source (`rightDataType`) and target (`leftDataType`) data types are the same. If so, the assignment operation itself is given that data type. If not, the assignment node is considered type-unstable, and MEGAGUARDS will force the entire loop to be executed by the interpreter. Similarly, MEGAGUARDS tags `OrElseAssignmentNode` nodes with the data type of its child nodes unless any of its child nodes have different data types, or if any child node is marked as type-unstable. In both of these cases, MEGAGUARDS forces the interpreter to execute the loop instead. MEGAGUARDS does, however, re-evaluate loops it fails to offload should they ever be executed again. For operations such as binary arithmetic operations and function calls, the algorithm tags the operation with the dominant type of the operation's child nodes using the `DominantType` method. If our algorithm determines that all operations in the AST are type-stable, it will return the inferred data types for each node.

Interprocedural Analysis Support

To support interprocedural type stability analysis, MEGAGUARDS does function cloning; it creates new variants of functions called within loops and specializes each variant based on its argument types. Figure 6 shows an example of a loop with two function calls.

MEGAGUARDS runs Algorithm 1 on the loop in function `baz`. While traversing the loop's AST, MEGAGUARDS identifies a user-defined function call to `qux` with one argument, `b[i]`. MEGAGUARDS creates a specialized version of this function using Algorithm 1. Since `b[i]` is of type `double`, the algorithm can determine that this specialized version of function `qux` returns a value of type `double`. MEGAGUARDS reports this return type back to the call site in the loop and continues the loop traversal. MEGAGUARDS then identifies another call to function `qux` with argument `a[i]` of type `int`. Since MEGAGUARDS has only created a variant of `qux` specialized for arguments of type `double`, MEGAGUARDS creates another variant here specialized for argument type `int`.

3.3.2 MegaGuards-Specialized AST

MEGAGUARDS translates the original ASTs for type-stable loops into specialized, strongly-typed ASTs based on the type information inferred during our type stability analysis. Figure 7 shows an example of such a translation. The figure shows how MEGAGUARDS converts the generic `For` node in the ZipPy AST into a specialized `For` node, which has a `LoopInfo` child node. The `LoopInfo` node stores the loop expression, loop bounds, and step size. The information in the `LoopInfo` node is later used for the bounds check optimization (Section 3.3.3), dependence analysis (Section 3.4.1) and kernel code generation (Section 3.4.2).

Each node in an MEGAGUARDS-specialized AST operates on a specific data type. `doubleAssign`, for example, can only assign a double floating-point value to a variable. The nodes in the ZipPy AST, on the other hand, are generic and can handle any data type. These ZipPy AST nodes contain type checks and conditional branches. The MEGAGUARDS-specialized nodes do not.

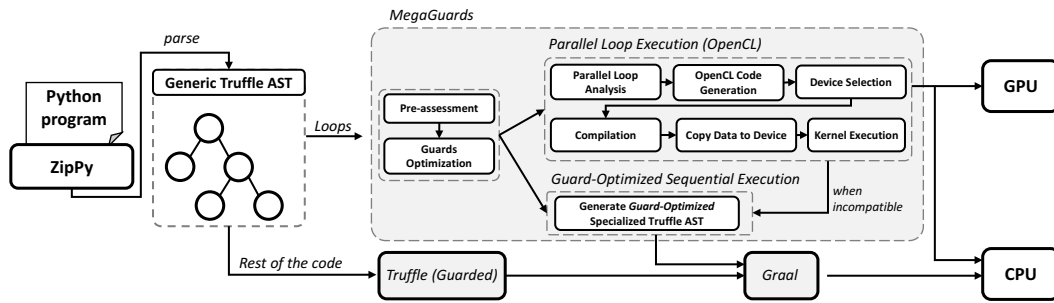
MEGAGUARDS supports translation of AST operations that operate on generic boxed data types. The assign operation in the ZipPy AST, for example, writes to `b[i][j]`. Variable `b` is of generic type `ListList` according to the original context frame generated by ZipPy, but during type stability analysis, MEGAGUARDS augments the context frame with more precise type information by unboxing `b` into a primitive data structure of type `double[][]`. Based on feedback from this unboxing pass, MEGAGUARDS establishes that the `=` operation in question must be translated into a write operation of type `double` (i.e., a `doubleAssign` node).

The `GenericAdd` operation has two input types, representing the left and right sides of the add operation. Our type stability analysis recursively finds the dominant type for this operation. Since both sides are of type `double`, MEGAGUARDS can translate this node into a `doubleAdd`.

The MEGAGUARDS-specialized AST is considered to be type-stable and, thus, does not contain any traditional type guards. Once it is translated to OpenCL code or a guard-less specialized Truffle AST, the loop will only need to handle bounds checks and arithmetic overflows, resulting in code with significantly fewer conditional branches.

3.3.3 Bounds Check Optimization

Dynamically-typed languages must perform a bounds check for every array access. MEGAGUARDS optimizes this bounds check for arrays whose subscripts are *affine expressions*. The form of an affine expression is $\alpha x + \beta$ where x is a loop induction variable, and α and β are loop-invariant values. Array subscripts in this form allow us to safely determine the upper and lower bounds for all array accesses before executing the loop. As with guards, MEGAGUARDS removes bounds checks from the loop body and inserts only checks for the upper and lower bounds ahead of the loop.



■ **Figure 9** MEGAGUARDS detailed internal analysis and offloading.

in the MEGAGUARDS-specialized AST. If MEGAGUARDS detects a mismatch, it invalidates the specialized AST and rebuilds it based on the effective types. The mega guard also performs bounds and overflow checks hoisted out of the loop.

3.4 Parallel Analysis and Execution

After creating a specialized AST, MEGAGUARDS tests if the loop is eligible to be a OpenCL kernel as shown in Figure 9. To guarantee the independence of loop iterations, MEGAGUARDS performs cross-iteration dependence analysis by leveraging a polyhedral model (Section 3.4.1).

3.4.1 Dependence Analysis

MEGAGUARDS runs a dependence analysis to verify that no flow (i.e., read after write), anti (i.e., write after read), or output (i.e., write after write) dependencies exist between the different iterations of the loop. MEGAGUARDS does not offload any loops having such cross-iteration dependencies. MEGAGUARDS performs polyhedral dependence analysis using the Integer Set Library (ISL) and the Polyhedral Extraction Tool (PET) [53, 52], which provides a polyhedral compilation API.

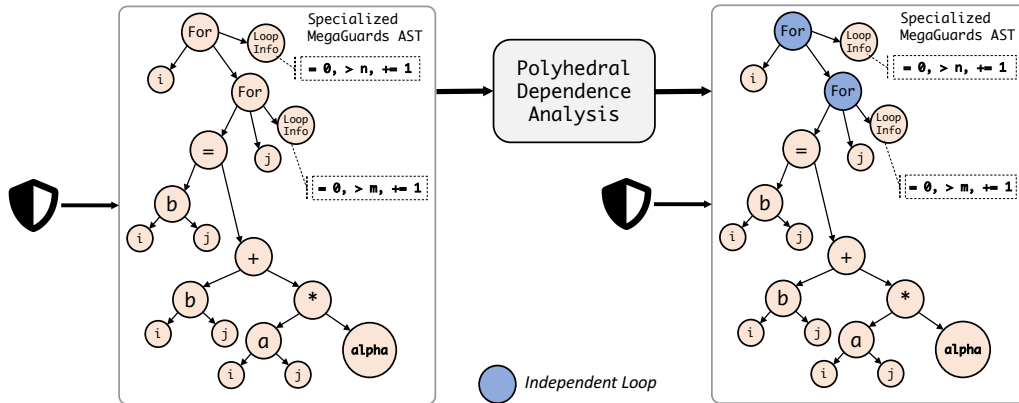
On top of the dependence analysis, we also perform alias analysis to ensure that references between different data structures are completely separate. This alias analysis pass is necessary since the polyhedral analysis incorrectly treats aliases as references to separate memory locations, and might, consequently, fail to identify certain loop dependencies. Our alias analysis scans through data structure references to verify that each data structure does indeed point to a separate memory location. If we do detect aliases within the same loop, then we refrain from offloading that loop.

Figure 10 shows how we feed the MEGAGUARDS-specialized AST, generated from the code in Figure 1, to the polyhedral dependence analysis. MEGAGUARDS is able to verify that no cross-iteration dependencies exist in either of the loops and can therefore safely optimize both loops.

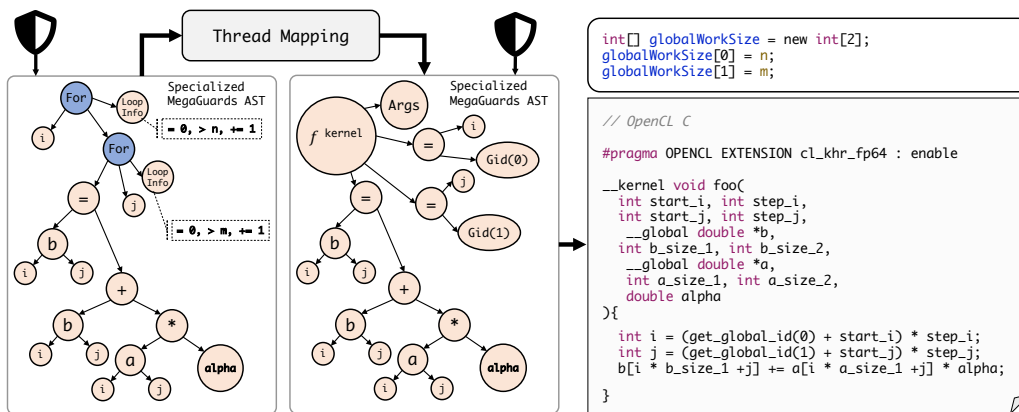
MEGAGUARDS supports scalar privatization for temporary scalar variables that are not referenced outside the loop [10]. This eliminates loop-carried output dependencies resulting from the temporary scalar variables and, as a result, increases the number of offloading candidates.

3.4.2 Kernel Code Generation

Once it has fully analyzed a loop, MEGAGUARDS adds the necessary run-time bounds checks to the loop’s AST (see Section 3.3.3) and then translates the AST into OpenCL code. MEGAGUARDS then compiles the code into a binary kernel and stores this kernel in a cache. Keeping this cache allows us to skip analysis and code generation and compilation.



■ **Figure 10** MEGAGUARDS polyhedral dependence analysis of a MEGAGUARDS-specialized AST.



■ **Figure 11** MEGAGUARDS thread mapping and code generation.

3.4.3 Thread Mapping

MEGAGUARDS leverages OpenCL’s multi-dimensional thread range capability, called NDRange, to maximize the thread-level parallelism (TLP) for kernels with nested loops. Where possible, MEGAGUARDS attempts to parallelize entire nested loops. Our thread mapping scheme is compatible with existing concurrency schemes [30, 31].

MEGAGUARDS’s thread mapping follows an *outer-loop-first* policy to maximize the parallelized region and, at the same time, minimize the number of kernel invocations. MEGAGUARDS currently only supports thread mapping of perfectly nested loops. We leave support for imperfectly nested loops as future work.

NDRange allows us to specify the number of threads we want to create on each computing device. We map each thread to an N-dimensional index space. As the latest version of OpenCL supports up to three dimensions, MEGAGUARDS can map nested loops with up to three nesting levels to SIMT threads.

Figure 11 illustrates MEGAGUARDS’s thread mapping pass. MEGAGUARDS takes the list of independent loops produced by our dependence analysis as input (see Section 3.4.1), and searches for a perfectly nested form of loops starting from the outer-most independent loop. We repeat this process until we get a maximum of 3-D ranges. MEGAGUARDS’s thread mapping follows the *outer-loop-first* policy in order to maximize the parallelized region and, at the same time, minimize the number of kernel invocations.

MEGAGUARDS converts `for` loops into an OpenCL kernel based on the SIMT programming model by rewriting the specialized AST into a kernel AST, as shown in Figure 11. In this step, an iteration vector of nested `for` loops is mapped to a unique thread ID given to each SIMT thread. For example, an iteration vector of a 2-level nested loop, (i, j) , is mapped to a unique thread ID represented as a 2-D array value which can be accessed by the `getGlobalId(dim)` node. Then, the AST of a loop body is mapped to a kernel body and the `For` nodes are removed. In this example, $n \times m$ threads are created according to the iteration space range of the nested `for` loops, (n, m) . Instead of iterating loops with induction variables, the kernel body will be concurrently executed by the SIMT threads with their unique IDs.

3.4.4 Kernel Data Management

Before the execution of an offloaded loop can start, we need to make sure that all the data the loop accesses is present on the OpenCL device. This means that MEGAGUARDS might have to copy data structures from the main memory to the OpenCL device.

To avoid redundant copy operations, MEGAGUARDS manages a cache of data that is present on each OpenCL device. MEGAGUARDS does not copy any data that is already present on the device, unless the data is marked as invalid in the cache. This kernel data management (KDM) optimization allows kernels to share common data. MEGAGUARDS automatically inserts the code that marks cache entries as invalid during the unboxing pass, when the associated data is modified.

MEGAGUARDS also optimizes `map` operations that write their results to a list they never read from. Instead of copying an empty result list before we offload a map operation, MEGAGUARDS simply allocates that list on the device but does not initialize it. MEGAGUARDS only copies the list from the OpenCL device to the main memory when the offloaded kernel finishes its execution.

3.4.5 Kernel Execution and Device Selection

MEGAGUARDS proceeds to the kernel execution stage as soon as the interpreter reports the loop offset. For non-zero loop offsets, we only offload the remaining iterations of the loop.

MEGAGUARDS can execute kernels on a specific acceleration device or select the best device for each kernel *adaptively*. With adaptive device selection enabled, we compile kernels for each available acceleration device and cache the compiled kernels, one for each device. Then, we pick an accelerator to execute the kernel on and we store the total run-time of the loop. We configured MEGAGUARDS to always try a CPU device when a loop executes for the first time. After multiple kernel invocations, sufficient performance data will be available to select the fastest device for that kernel. A device is selected if it is faster than the others, and if the kernel has executed at least once on every accelerator. This strategy can cause the program to miss out on performance benefits for a few runs, but it quickly pays off when the selection converges. In case of a tie, MEGAGUARDS selects the GPU as the best execution device.

3.4.6 Execution of A Cached Kernel

Future executions of a kernel can use the cached kernel code if the following conditions are met:

- *the mega guard check passes:* The mega guard check reads the loop's context frame, unboxes all variables in the frame, and compares their types with the cached copy of the kernel's augmented context frame (see Section 3.3).
- *the loop body does not contain aliases:* We conservatively perform alias analysis to make sure that the program has not introduced new aliases since we performed the original translation of the loop.
- *the hoisted bounds and overflow checks are still valid:* We re-run part of the bounds and integer overflow check optimization pass to ensure that no new bounds checks or overflow checks are required.

If all three conditions hold, we offload the cached copy of the kernel code to the acceleration device. If not, we re-run the complete analysis and generate a new, specialized kernel.

3.5 Guards-optimized Sequential Execution

MEGAGUARDS translates the MEGAGUARDS-specialized AST to a guards-optimized Truffle AST if any of the parallel loop analysis fails (see Section 3.4). The resulting Truffle AST will not have any type checks but may still have bounds and overflow checks that MEGAGUARDS is unable to optimize (see Section 3.3.3). In order to preserve the integrity of data, MEGAGUARDS backs up the modifiable data structures and restores them if a bounds violation or an overflow occurs. MEGAGUARDS then executes this guards-optimized Truffle AST directly on top of the Truffle/Graal stack.

If the sequential Truffle AST contains any nested loop that can be parallelized, MEGAGUARDS offloads the nested loop(s) and optimizes the data transfers within the sequential execution scope.

3.6 Implementation Capabilities and Limitations

Recursion

MEGAGUARDS constructs a call graph of function calls in the loops to verify that no recursion exists in any of the loops that can potentially be offloaded. If MEGAGUARDS does detect recursion in an offloading candidate, then it will execute the loop using the guards-optimized sequential execution instead.

Built-in Functions

MEGAGUARDS supports reduction throughout Python's built-in `reduce` function and the embarrassingly parallel `map` operator. MEGAGUARDS specializes `map`'s and `reduce`'s `apply` functions based on the lists that are passed iteratively to the function.

MEGAGUARDS also supports many of Python's built-in math functions (e.g., `max`, `sqrt`, `cos`, ...). MEGAGUARDS translates calls to such functions into calls to their respective counterparts in the OpenCL framework, and then specializes the translated calls based on the call arguments.

Non-local Control Constructs

Currently, MEGAGUARDS does not support language features that cause a non-local control flow, such as exceptions and generator expressions, i.e., `suspend/resume`. Our lightweight pre-assessment (see Section 3.2) checks if such a non-local control construct exists in the loop and if so, it falls back before proceeding to optimize guards.

Loop Transformations

MEGAGUARDS supports scalar variable loop privatization to increase the number of parallelizable loops. Other loop transformation techniques such as array variable loop privatization [51], loop splitting and loop peeling [20] could further enhance the parallelism if applied to MEGAGUARDS. Loop peeling, for example, splits any first or last few problematic iterations from the loop such that the remaining iterations are no longer dependent on each other. As a future work, MEGAGUARDS can incorporate such loop transformations and parallelize the transformed loops that become free of a loop-carried dependence.

4 Evaluation

4.1 Experimental Setup

We ran our benchmarks on the following system:

CPU: Intel Core i7-6700K @ 4 GHz Quad-Core CPU with Hyper-Threading representing 8 compute units (CU). 64GB of RAM. Turbo Boost disabled.

GPU: NVIDIA GeForce GTX 1080 Ti with 11GB of RAM and 3584 Stream Processors.

OS: Ubuntu x86_64 16.04.2 LTS using Linux kernel 4.4.0-122. GNU GCC 5.5.0, Oracle labsjdk1.8.0_151-jvnci-0.39, GraalVM v0.30 and AMD APP SDK v3.0.136.

We compared the performance of MEGAGUARDS with:

Python Systems:

- **CPython** version 3.5.2: The standard Python 3 interpreter.
- **PyPy 3** version 5.10.0 [6]: Python 3 implementation, uses a meta-tracing JIT compiler to compile Python code into machine code for CPU.
- **ZipPy** (github revision ff6d067) [49]: Python 3 implementation targeting Graal, uses the Truffle framework to JIT-compile specialized AST nodes into x86 machine code.

Heterogeneous Computing Frameworks:

- **OpenCL C/C++ (CPU)** Intel driver ver. 1.2.0.25
- **OpenCL C/C++ (GPU)** NVIDIA driver ver. 390.59

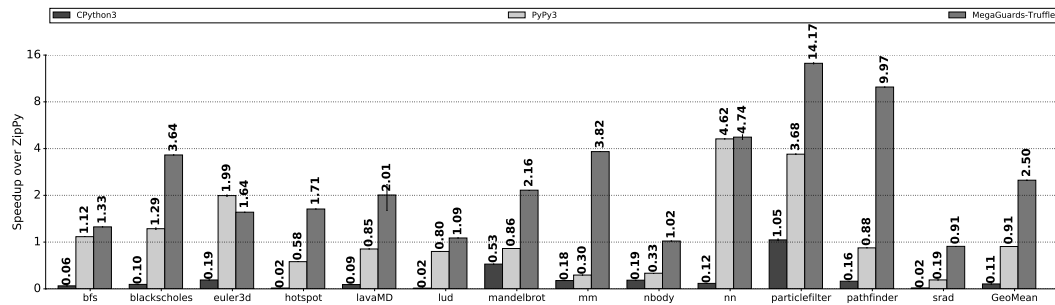
We ran each benchmark three times on each system and calculated the geometric mean of the execution times. We measured the execution times including data transfers from the CPU memory to the accelerator device memory and vice versa.

We ran the pure Python implementations of each benchmark to measure the MEGAGUARDS, ZipPy, PyPy and CPython performance. To properly measure peak performance, we warmed up the benchmarks to allow ZipPy and PyPy to just-in-time compile the Python code.

We compared four different backend/device selection configurations for MEGAGUARDS:

- **MegaGuards-Truffle:** running ZipPy sequentially on the CPU with our guards optimization enabled.
- **MegaGuards-CPU:** offloading to CPU OpenCL devices only.
- **MegaGuards-GPU:** offloading to GPU OpenCL devices only.
- **MegaGuards-Adaptive:** using our adaptive device selection we discussed in Section 3.4.5.

We carefully chose program inputs that are representative of large, real-world data sets and simulations.



■ **Figure 12** Sequential execution speedup of MEGAGUARDS-Truffle compared to CPython, PyPy, ZipPy normalized to ZipPy on a \log_2 scale.

■ **Table 1** Sequential execution time (in seconds) for MEGAGUARDS-Truffle, CPython, PyPy and ZipPy.

	bfs	blackscholes	euler3d	hotspot	lavaMD	lud	mandelbrot	mm	nbody	nn	particlefilter	pathfinder	srad
ZipPy	16.913	139.482	40.366	7.205	50.329	148.981	98.069	1018.838	498.037	16.914	61.751	4.78	4.668
MG-Truffle	12.759	38.326	24.617	4.219	25.037	136.91	45.418	266.377	487.023	3.567	4.359	0.48	5.135
PyPy3	15.154	108.412	20.236	12.389	59.004	186.196	113.609	3452.785	1489.154	3.662	16.769	5.454	24.412
CPython	264.321	1462.925	212.529	418.922	546.66	9616.348	185.193	5734.737	2688.715	144.596	58.968	29.221	244.785

Benchmark Selection

We ported a set of benchmarks from the Rodinia benchmark suite [13, 14] to pure Python, using only Python built-in data types². We complemented this extensive set of benchmarks with the ones from the Numba Benchmark Suite [15] and the NVIDIA OpenCL SDK [43].

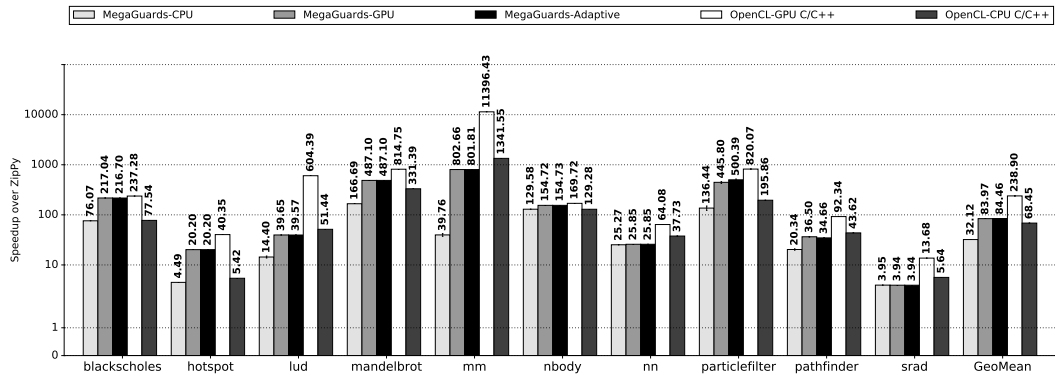
The selected benchmark programs (listed in Table 4) have two implementations: (i) pure Python, and (ii) a native hand-optimized version for OpenCL C/C++. We excluded bfs, euler3d and lavaMD benchmarks from the table as the existing polyhedral analysis could not disprove dependence (see Section 3.4.1) and thus the benchmarks only ran using MEGAGUARDS-Truffle backend.

4.2 Effect of Guards Optimization

In Figure 12, we show the performance impact of our guards optimization by measuring the sequential performance of ZipPy with the guards optimization enabled (MEGAGUARDS-Truffle). The performance is normalized to the baseline ZipPy on a logarithmic scale. The last set of bars represents the geometric mean performance of each system. Standard errors are also marked in the figure. Table 1 shows the execution time for each benchmark (in seconds). We measured the performance with the largest data sizes the Rodinia benchmark suite provides.

Our guards optimization improves the sequential Python performance by up to $62.3\times$, $12.7\times$, and $14.1\times$ compared to CPython, PyPy and ZipPy. On average, we achieve a performance improvement of $22.72\times$, $2.74\times$ and $2.50\times$ over CPython, PyPy and ZipPy. `particlefilter` shows the most substantial performance improvement ($14.17\times$ over ZipPy) because our guards optimization removes most of its overflow checks (see Section 3.3.3).

² Will be publicly available at <https://github.com/seuresystemslab/megaguards-benchmarks>



■ **Figure 13** Parallel execution speedup of MEGAGUARDS compared to OpenCL C/C++ (CPU and GPU) normalized to ZipPy on a \log_{10} scale.

■ **Table 2** Parallel execution time (in seconds) for MEGAGUARDS, OpenCL C/C++ (CPU and GPU), and sequential ZipPy.

	blackscholes	hotspot	lud	mandelbrot	mm	nbody	nn	particlefilter	pathfinder	srad
ZipPy	139.482	7.205	148.981	98.069	1018.838	498.037	16.914	61.751	4.78	4.668
MG	0.644	0.357	3.765	0.201	1.271	3.219	0.654	0.123	0.138	1.186
MG-GPU	0.643	0.357	3.757	0.201	1.269	3.219	0.654	0.139	0.131	1.185
MG-CPU	1.834	1.606	10.346	0.588	25.626	3.843	0.669	0.453	0.235	1.181
OpenCL-GPU	0.588	0.179	0.246	0.12	0.089	2.934	0.264	0.075	0.052	0.341
OpenCL-CPU	1.799	1.329	2.896	0.296	0.759	3.852	0.448	0.315	0.11	0.828

4.3 Parallel Execution Performance and Complexity Analysis

4.3.1 Characteristics of Kernels

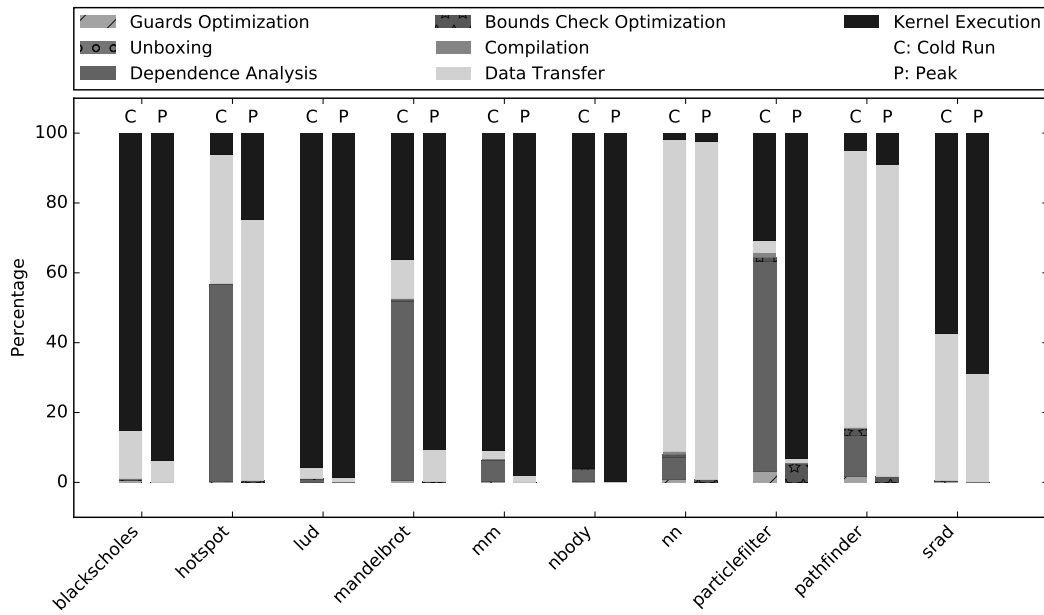
Table 4 shows the following characteristics for each benchmark:

- **Loops:** the number of executed and the number of offloaded loops. Nested loops are counted separately.
- **Kernels:** the number of generated kernels and the number of kernel invocations for a single run.
- **Thread Count:** the total number of parallel executions of the kernel(s) body for a single run.
- **MegaGuards-Adaptive:** the final acceleration device selection on the generated kernels using our adaptive selection technique (see Section 3.4.5).
- **LOC:** the lines-of-code counts for the Python and OpenCL C/C++ implementations of the benchmark's source code.
- **McCabe Cyclomatic Complexity:** the Cyclomatic Complexity [38] of the Python and OpenCL C/C++ implementations of the benchmark's source code.

Our analyses of the benchmarks' source code, i.e., LOC and McCabe Cyclomatic Complexity, show that the plain Python implementations of the benchmarks are significantly less complex than the OpenCL implementations.

4.3.2 Parallel Execution Performance

Figure 13 shows MEGAGUARDS's speedups normalized to ZipPy on a logarithmic scale. The last set of bars represents the geometric mean performance of each system. We measured the performance with the largest data sizes the Rodinia benchmark suite provides. We



■ **Figure 14** Breakdown of MEGAGUARDS passes for parallel execution.

■ **Table 3** Time (in milliseconds) for each pass of the parallel execution.

	blackscholes		hotspot		lud		mandelbrot		mm		nbody		nn		particlefilter		pathfinder		srad	
	Cold	Peak	Cold	Peak	Cold	Peak	Cold	Peak	Cold	Peak	Cold	Peak	Cold	Peak	Cold	Peak	Cold	Peak	Cold	Peak
Guards Optimization	4	0	4	0	2	0	3	0	2	0	4	0	2	0	9	0	3	0	5	0
Unboxing	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Dependence Analysis	0	0	873	0	21	0	266	0	89	0	125	0	14	0	174	0	21	0	0	0
Bounds Check Optimization	1	0	2	2	2	1	2	1	2	0	2	2	2	1	3	4	3	2	1	1
Compilation	2	0	2	0	2	0	2	0	3	0	2	0	2	0	4	0	1	0	1	0
Data Transfer	95	38	573	305	80	31	58	34	36	23	2	1	204	122	10	1	141	108	591	362
Kernel Execution	592	563	95	101	2422	2353	187	342	1321	1221	3238	3247	4	3	89	68	9	11	803	808

also marked standard errors in the graph but the errors are too small to be seen except in the MEGAGUARDS-CPU run of `particlefilter`. Table 2 shows the execution time for each benchmark complemented with the sequential execution time of ZipPy.

MEGAGUARDS shows substantial speedups compared to other systems using pure Python benchmarks. For this set of benchmarks, our system performed up to $802\times$ faster than ZipPy and $84\times$ on average. MEGAGUARDS approaches the performance of native hand-optimized OpenCL C/C++ code (CPU and GPU), being only $2.82\times$ slower on average, without requiring extensive knowledge on heterogeneous computing frameworks. Note that dynamic languages are typically one or two orders of magnitudes slower than C/C++.

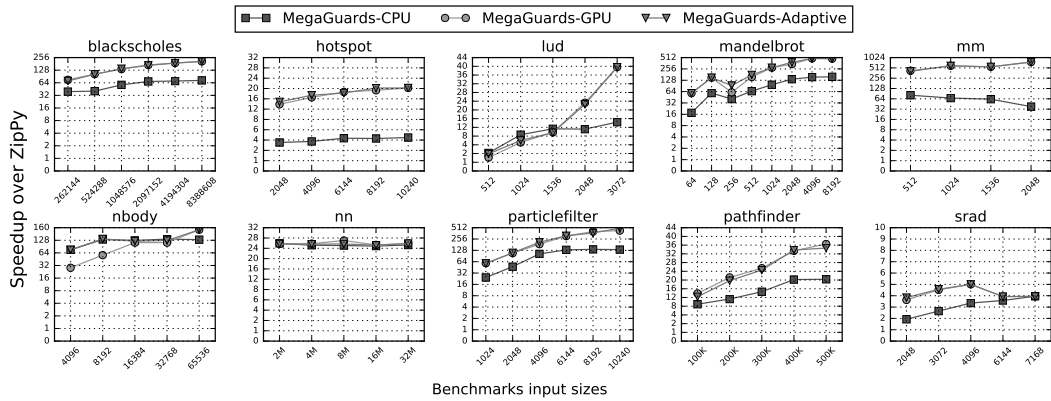
Figure 14 shows the cost of each analysis pass in MEGAGUARDS during a cold run (*Cold Run*), and when utilizing pre-evaluated (i.e., cached) kernels (*Peak*). Table 3 shows the execution time of each analysis pass for each benchmark (in milliseconds). Noticeably in Figure 14, our guards optimization and bounds checking stages account for limited overhead due to their inexpensive computations.

In the black-scholes and nbody benchmarks, MEGAGUARDS approaches the performance

■ **Table 4** Benchmark characteristics for parallel execution.

benchmark	Loops		Kernels		Thread Count	MegaGuards-Adaptive		LOC		McCabe Cyclomatic Complexity	
	Total	Offloaded	Gen.	Exec.		CPU	GPU	Python	OpenCL	Python	OpenCL
blackscholes	2	1	1	100	8.4×10^8	0	1	51	203	13	41
hotspot	12	2	1	10	1.0×10^9	0	1	146	288	32	99
lud	3	2	2	1024	1.0×10^7	0	2	55	391	44	104
mandelbrot	3	3	1	1	6.7×10^7	0	1	42	183	16	51
mm	3	3	2	2	8.4×10^6	0	2	29	230	15	49
nbody	2	2	1	1	6.6×10^4	0	1	34	192	10	44
nn	3	1	1	1	1.6×10^7	0	1	82	456	24	67
pathfinder	2	1	1	101	3.0×10^7	0	1	63	258	22	92
particlefilter	36	16	10	94	1.2×10^6	0	10	255	719	101	205
srad	9	4	2	4	2.1×10^8	0	2	89	477	27	136
Median	3	2	1	7	3.0×10^7	0	1	59	273	23	79.5

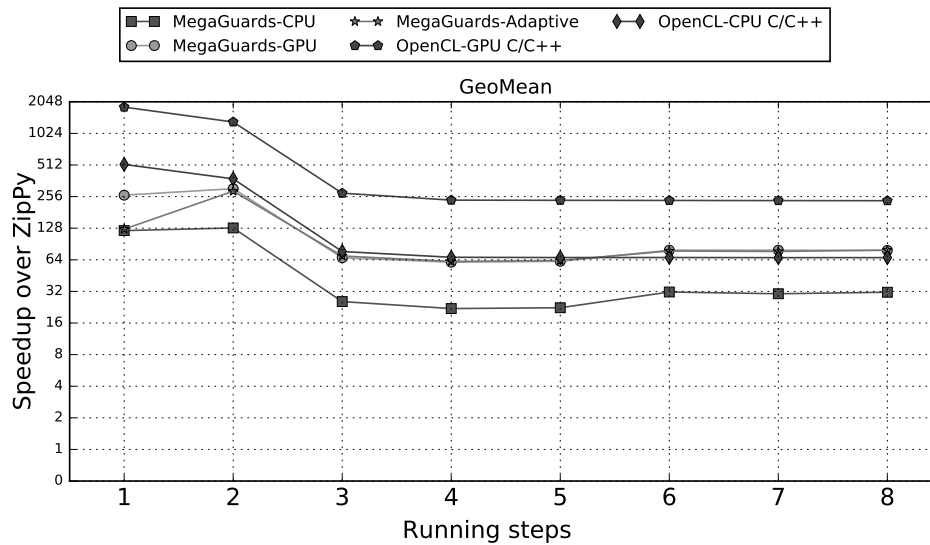
Offloaded: Number of offloaded loops. Gen.: Number of generated kernels. Exec.: Number of kernels' executions.



■ **Figure 15** Peak performance of MEGAGUARDS with different data sizes for parallel execution.

of the OpenCL C/C++ implementations and is able to reduce the number of bounds and overflow checks significantly (see Section 3.3.3). MEGAGUARDS outperformed ZipPy by $217\times$ and $154\times$.

The mm and mandelbrot benchmarks had minimal data transfer rates, and have 2-level nested loops that MEGAGUARDS assigned to a 2-dimensional thread range (see Section 3.4.3). This resulted in large speedups, especially when executing on GPU acceleration devices. In the mm benchmark, MEGAGUARDS created two specialized kernels for the same loop, one for double floating point-typed variables and one for long integer-typed variables. NVIDIA's optimized OpenCL implementation of mm outperformed MEGAGUARDS by $14.2\times$. The reason is that this hand-optimized OpenCL implementation aggressively exploits data locality between local threads. Plus, unlike Python, the OpenCL code does not include safety checks for detecting out-of-bounds array accesses and arithmetic overflows because in a static language like OpenCL writing a safe code is user's responsibility. MEGAGUARDS bounds and overflow checks enforce the safety of the kernel operations and guarantee the integrity of the result. Nevertheless, the high performance of the mm benchmark on MEGAGUARDS demonstrates the flexibility of our system to adapt to type changes at run time without degrading performance.



■ **Figure 16** Mean speedup of warming up steps of MEGAGUARDS normalized runs for ZipPy.

The situation in `particlefilter`, `srad`, `hotspot` and `lud` is similar. MEGAGUARDS generated ten, two, one and two specialized kernels for these benchmarks respectively. Most kernels were assigned to a 1-dimensional thread range in `particlefilter` and `lud`, and to 2-dimensional thread ranges in `srad` and `hotspot`. In the OpenCL-GPU implementation, the core computation of `particlefilter`, `srad`, `hotspot` and `lud` features cooperative local threads that share data through a local cache. As a result, OpenCL-GPU outperformed MEGAGUARDS using a GPU by 1.6 \times , 3.4 \times , 2 \times and 15.2 \times , respectively.

Overall, we observed that acceleration-compatible loops experienced speedups by up to an order of magnitude under MEGAGUARDS.

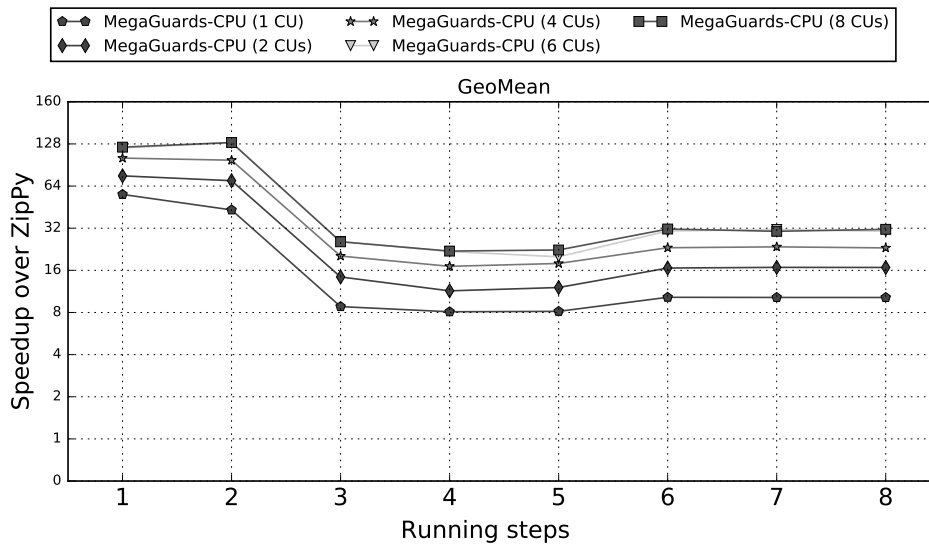
4.3.3 Peak Performance with Various Input Sizes

To show the scalability of MEGAGUARDS, we measured the peak performance with different data sizes. The results are normalized to the sequential ZipPy implementation. As shown in Figure 15, MEGAGUARDS yields performance gains relative to the size of the inputs. MEGAGUARDS-Adaptive follows the best device performance curves with the varying input sizes and relieves the user from manually setting a specific accelerator device.

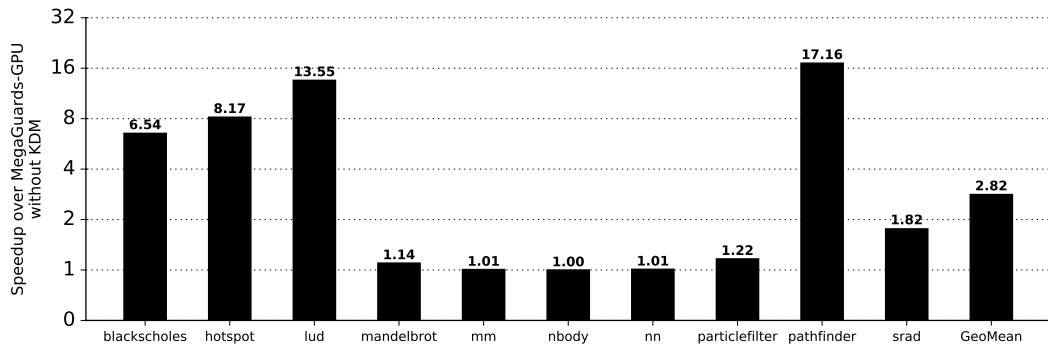
4.3.4 Performance of MegaGuards on Each Run Step

So far, we only measured the peak performance of ZipPy. We gave ZipPy's underlying Truffle/Graal stack and PyPy a couple of warm up runs to specialize, optimize and compile every hot path in ZipPy's AST into x86 machine code. MEGAGUARDS, by contrast, specializes, optimizes and compiles the program's hottest paths (i.e., loops) immediately. MEGAGUARDS therefore brings even greater performance benefits to end users who do not warm up the program interpreter.

These performance benefits are illustrated in Figure 16. In this figure, we see the mean performance of each run of our benchmarks. In the first run, ZipPy is executing the benchmarks in the interpreter. Through the second to the sixth runs, the JIT compiler



■ **Figure 17** Mean speedup of warming up steps of MEGAGUARDS-CPU with various CU counts normalized runs for ZipPy.



■ **Figure 18** Effect of KDM optimization on MEGAGUARDS.

specializes and optimizes the code. After that, ZipPy reaches a steady state. By contrast, MEGAGUARDS specializes the loop from the first run, thanks to our type stability analysis *before* executing a loop. This leads to large performance benefits instantly. During the run steps, our MEGAGUARDS-Adaptive execution mode explores acceleration devices and benchmarks their performance at run time until it settles on the best device.

4.3.5 Scalability

Figure 17 shows how MEGAGUARDS scales with the number of CUs. We scaled the number of available CUs for the CPU OpenCL device and measured the mean performance of each run of our benchmarks. MEGAGUARDS shows performance gains as we increase the number of CUs.

We see the down curve in the middle of the performance graph. This is because the CPU is shared among other processes such as the Graal compiler’s background analysis. This also explains MEGAGUARDS-CPU underperforms compared to OpenCL-CPU C/C++ in Figure 13.

4.3.6 Effect of Kernel Data Management

In Figure 18, we show the effect of our kernel data management optimization on peak performance. This optimization yields an average performance gain of $2.82\times$. The KDM optimization provides the larger performance gains for the benchmarks with the higher data transfer rates.

5 Related Work

5.1 Type Inference and Guards Optimization

Several guards optimizations [4, 32, 17] have been proposed to reduce the number of type checks during executions of a dynamically-typed languages. Bebenita et al. proposed a profile-based guards optimization that significantly reduces the number of type checks on hot execution paths [4]. Dot et al. proposed a HW/SW-based profiling mechanism to reduce the number of guards [17]. The disadvantage of this approach is that the profiling itself incurs some overhead, and that guards optimization only applies to code paths that execute during profiling. MEGAGUARDS, by contrast, would also optimize those code paths. Kedlaya et al. proposed to optimize guards using type inference and type feedback [32]. The type inference phase has similarities to our approach, but does enforce guards around global variables and function calls. MEGAGUARDS, on the other hand, propagates global variable types to the generated *mega guard* and creates internal specialized variants of functions based on their argument types. Thus, the MEGAGUARDS-specialized AST contains guards-less regions with static typing properties and optimized bounds and overflow checks.

5.2 Heterogeneous Programming in Dynamic Languages

Although their popularity resulted from ease of use and high productivity, dynamically-typed languages are less attractive in terms of performance given their traditional lack of support for parallelism. PyCUDA and PyOpenCL facilitate native GPU programming in Python to enable accelerations on parallel hardware for dynamically-typed languages [33]. Harnessing the full potential of GPU with these platforms, however, requires low-level understanding of heterogeneous programming models and faces a steep learning curve.

Several programming models have been proposed to ease the exploitation of parallel hardware for dynamically-typed languages. Numba [34] and unPython [23] proposed an annotation-based solution to perform vectorized operations on both CPU and GPU. Theano provides a set of pre-compiled vector operations for GPUs [5]. Copperhead is a Python programming model that performs GPU parallelization using aggregate operations, called skeletons, such as `map` and `reduce`, that are implicitly parallel [11]. Chakravarty et al. also proposed a similar skeleton-based aggregate operations to dynamically generate GPU code for Haskell programs [12]. Jibaja et al. proposed a language extension for JavaScript to support SIMD vector instructions [29]. This requires users to explicitly specify data types of program operations. River Trail [27] proposed a programming model for the implicitly parallel operations targeting OpenCL acceleration devices. To get the full performance benefits, however, these approaches require code annotations and the use of parallel libraries and special type definitions such as parallel arrays and NumPy's type system. The overarching goal of this paper is, however, to automatically and transparently exploit heterogeneous parallel hardware without code rewriting and knowledge of underlying system architecture.

5.3 Auto-Parallelization for Dynamic Languages

There are previous approaches to automatically vectorize or parallelize vector computations in dynamically-typed languages [50, 41, 54, 45]. Plangger and Krall introduced a vectorization technique that uses SIMD vector instructions on the CPU [45]. The technique is employed in PyPy. Plangger and Krall also applied loop unrolling and array bounds check optimizations to enhance the vectorization. This solution relies on NumPy’s type system, however, and only works for loops whose structure matches one of the patterns supported by the tool. Similarly, Riposte [50] and pqR [41] exploit parallelism on the CPU vector operations for the R language. Wang et al. vectorize the `Apply` class of operations targeting multi-core CPUs and GPUs [54].

Fumero et al. offload the `Apply` to GPU using the collected type information profile on Graal’s partial evaluation [21]. Their approach bears some similarities with ours in how they check input data types before offloading and how they handle mis-speculations occurred while executing kernels. However, Fumero et al. rely on a language’s parallel semantic, i.e., `Apply` and still requires that developers deal with the effects of arithmetic overflows themselves. On the other hand, MEGAGUARDS is not confined to certain types of operations, such as vector computations, or specific forms of operations, such as operations in the `Apply` class, which is what prior work does. Moreover, MEGAGUARDS ensures the integrity of the computed results with its in place bounds and overflow checks.

Ma et al. proposed pR that automatically parallelizes loops and independent tasks from unmodified R code [37]. pR identifies all parallelizable elements including loop iterations and methods and dispatches them to multiple CPU cores. pR does not incorporate a JIT compiler but it conducts dependence analysis and parallelization at the interpretation level. With this feature, pR does not require complicated pointer and type analysis for parallelization because the language itself does not have pointers and types. However, this approach hardly benefits from SIMD execution on the GPU because the interpreter has complex control flows and frequently accesses shared VM state, which must generally be avoided during GPU execution. MEGAGUARDS’s parallelization is part of the JIT compilation process, and the generated code does not contain complex control flow instructions or accesses to shared VM state.

Thread-level speculation (TLS) based approaches facilitate automatic parallelization for dynamically-typed languages at the JIT compilation level [39, 40]. Mehrara et al. proposes a system that dynamically parallelizes data-parallel loops in JavaScript by handling runtime type changes based on TLS [39]. In this mechanism, all live-in data is saved before speculative execution and when there is a speculation failure (e.g., a type change), the program restarts from the checkpoint with the saved data. Na et al. leverages the property of idempotence to recompute the mis-speculated loops without side effect [40], instead of checkpoint-and-recovery which may require handling of large amount of data. However, both of these TLS-based approaches do not remove speculations in multi-threaded execution. This makes it hard to move to GPUs because the mis-speculation penalty of the kernel execution is significant either with checkpointing or recomputing, due to the excessive data transfer and kernel invocation overhead. Furthermore, the mis-speculation handling code may result in complex control-flows, which should be avoided in the kernel execution. In this paper, we separate speculations from the offloaded kernel code based on our type feedback and analysis. This feature makes MEGAGUARDS effectively accelerate dynamically-typed languages on GPUs.

Generally, none of these techniques apply to multiple languages since the previous approaches are either based on single language platforms [37, 39] or rely on language-specific primitives [54, 21]. MEGAGUARDS, by contrast, parallelizes *for loops*, which are a near-

universal language construct, and preserves portability of the programs. Since MEGAGUARDS is based on Truffle, a multi-language platform, our approach therefore generalizes to other Truffle languages, such as R, JavaScript, and Ruby.

6 Conclusions

We have presented the design and implementation of MEGAGUARDS, a new system that automatically and transparently optimizes Python programs by offloading compute-intensive kernels to accelerator devices. The key component in our system is an *a priori* type stability analysis step that overcomes the speculative limitations of type feedback by analyzing potential, future type changes. Only after this analysis ensures that no unexpected type changes *can* occur, we continue to optimize the code for execution on acceleration devices; if a dependence is detected, we mark the code for sequential execution.

Because our system is built on top of the ZipPy Python 3 virtual machine, which itself builds on the Truffle framework, MEGAGUARDS generalizes to all other Truffle languages, such as TruffleRuby and FastR. All the major Truffle languages—JavaScript, R, and Ruby—can, therefore, directly benefit from the presented techniques and their implementations and enjoy “free” and significant speedups.

Although MEGAGUARDS is just a first step, our results indicate that this research direction holds tremendous potential for further investigation. Presently, we are interested in extending MEGAGUARDS in multiple ways. First, we plan on improving the adaptive device selection to be able to select the best device for generated kernels ahead-of-time. Second, we plan to apply our *a priori* type stability analysis on a complete program to produce a precompiled and optimized executable form. Moreover, our plan is to add support for heterogeneous computing to generators [62]. Finally, support for collaborative CPU and GPU parallelism is also an interesting research direction.

References

- 1 Keith Adams, Jason Evans, Bertrand Maher, Guilherme Ottoni, Andrew Paroski, Brett Simmers, Edwin Smith, and Owen Yamauchi. The hiphop virtual machine. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA '14*, pages 777–790, New York, NY, USA, 2014. ACM.
- 2 Joshua Auerbach, David F. Bacon, Ioana Burcea, Perry Cheng, Stephen J. Fink, Rodric Rabbah, and Sunil Shukla. A compiler and runtime for heterogeneous computing. In *Proceedings of the 49th Annual Design Automation Conference, DAC '12*, pages 271–276. ACM, 2012.
- 3 Muthu Manikandan Baskaran, J. Ramanujam, and P. Sadayappan. Automatic c-to-cuda code generation for affine programs. In *Proceedings of the International Conference on Compiler Construction, CC'10*, pages 244–263. Springer-Verlag, 2010.
- 4 Michael Bebenita, Florian Brandner, Manuel Fahndrich, Francesco Logozzo, Wolfram Schulte, Nikolai Tillmann, and Herman Venter. Spur: A trace-based jit compiler for cil. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '10*, pages 708–725, New York, NY, USA, 2010. ACM.
- 5 James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. Theano: a CPU and GPU math expression compiler. In *Proceedings of the Python for Scientific Computing Conference (SciPy)*, 2010.

- 6 Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. Pypy, 2017. URL: <http://pypy.org/>.
- 7 Kevin J. Brown, HyoukJoong Lee, Tiark Rompf, Arvind K. Sajeeth, Christopher De Sa, Christopher Aberger, and Kunle Olukotun. Have abstraction and eat performance, too: Optimized heterogeneous computing with parallel patterns. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, CGO 2016, pages 194–205. ACM, 2016.
- 8 Stefan Brunthaler. Efficient interpretation using quickening. In *Proceedings of the 6th Symposium on Dynamic Languages*, DLS '10, pages 1–14. ACM, 2010.
- 9 Stefan Brunthaler. Inline caching meets quickening. In *Proceedings of the 24th European Conference on Object-oriented Programming*, ECOOP'10, pages 429–451. Springer-Verlag, 2010.
- 10 Michael Burke, Ron Cytron, Jeanne Ferrante, and Wilson Hsieh. Automatic generation of nested, fork-join parallelism. *The Journal of Supercomputing*, 3(2):71–88, 1989.
- 11 Bryan Catanzaro, Michael Garland, and Kurt Keutzer. Copperhead: Compiling an embedded data parallel language. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPOPP '11, pages 47–56. ACM, 2011.
- 12 Manuel M.T. Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. Accelerating haskell array codes with multicore gpus. In *Proceedings of the Sixth Workshop on Declarative Aspects of Multicore Programming*, DAMP '11, pages 3–14. ACM, 2011.
- 13 Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization*, IISWC '09, pages 44–54. IEEE Computer Society, 2009.
- 14 Shuai Che, Jeremy W. Sheaffer, Michael Boyer, Lukasz G. Szafaryn, Liang Wang, and Kevin Skadron. A characterization of the rodinia benchmark suite with comparison to contemporary cmp workloads. In *Proceedings of the IEEE International Symposium on Workload Characterization*, IISWC '10, pages 1–11. IEEE Computer Society, 2010.
- 15 Continuum Analytics. Numba benchmark suite, 2017. URL: <https://github.com/numba/numba-benchmark>.
- 16 L Peter Deutsch and Allan M Schiffman. Efficient implementation of the smalltalk-80 system. In *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 297–302. ACM, 1984.
- 17 G. Dot, A. Martinez, and A. Gonzalez. Removing checks in dynamically typed languages through efficient profiling. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 257–268, Feb 2017.
- 18 Christophe Dubach, Perry Cheng, Rodric Rabbah, David F. Bacon, and Stephen J. Fink. Compiling a high-level language for gpus: (via language support for architectures and compilers). In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 1–12. ACM, 2012.
- 19 Gilles Duboscq, Thomas Würthinger, Lukas Stadler, Christian Wimmer, Doug Simon, and Hanspeter Mössenböck. An intermediate representation for speculative optimizations in a dynamic compiler. In *Proceedings of the 7th ACM Workshop on Virtual Machines and Intermediate Languages*, VMIL '13, pages 1–10, New York, NY, USA, 2013. ACM. doi: 10.1145/2542142.2542143.
- 20 Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349, 1987.

- 21 Juan Fumero, Michel Steuwer, Lukas Stadler, and Christophe Dubach. Just-in-time gpu compilation for interpreted languages with partial evaluation. In *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE '17*, pages 60–73, New York, NY, USA, 2017. ACM.
- 22 Juan José Fumero, Toomas Remmelg, Michel Steuwer, and Christophe Dubach. Runtime code generation and data management for heterogeneous computing in java. In *Proceedings of the Principles and Practices of Programming on The Java Platform, PPPJ '15*, pages 16–26. ACM, 2015.
- 23 Rahul Garg and José Nelson Amaral. Compiling python to a hybrid execution environment. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units, GPGPU-3*, pages 19–30. ACM, 2010.
- 24 Tobias Grosser, Albert Cohen, Justin Holewinski, P. Sadayappan, and Sven Verdoolaege. Hybrid hexagonal/classical tiling for gpus. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO '14*, pages 66:66–66:75. ACM, 2014.
- 25 Tobias Grosser, Armin Groesslinger, and Christian Lengauer. Polly—performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters*, 22(04):1250010, 2012.
- 26 Gregory D. Hager, Mark D. Hill, and Katherine Yelick. Opportunities and challenges for next generation computing, 2015.
- 27 Stephan Herhut, Richard L. Hudson, Tatiana Shpeisman, and Jaswanth Sreeram. River trail: A path to parallelism in javascript. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '13*, pages 729–744, New York, NY, USA, 2013. ACM.
- 28 Kazuaki Ishizaki, Akihiro Hayashi, Gita Koblents, and Vivek Sarkar. Compiling and optimizing java 8 programs for gpu execution. In *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation, PACT '15*, pages 419–431. IEEE Computer Society, 2015.
- 29 Ivan Jibaja, Peter Jensen, Ningxin Hu, Mohammad R. Haghghat, John McCutchan, Dan Gohman, Stephen M. Blackburn, and Kathryn S. McKinley. Vector parallelism in javascript: Language and compiler support for simd. In *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation, PACT '15*, pages 407–418, Washington, DC, USA, 2015. IEEE Computer Society.
- 30 Onur Kayiran, Adwait Jog, Mahmut Taylan Kandemir, and Chita Ranjan Das. Neither more nor less: optimizing thread-level parallelism for gpgpus. In *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*, pages 157–166. IEEE Press, 2013.
- 31 Onur Kayiran, Nachiappan Chidambaram Nachiappan, Adwait Jog, Rachata Ausavarungnirun, Mahmut T Kandemir, Gabriel H Loh, Onur Mutlu, and Chita R Das. Managing gpu concurrency in heterogeneous architectures. In *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*, pages 114–126. IEEE, 2014.
- 32 Madhukar N. Kedlaya, Jared Roesch, Behnam Robotmili, Mehrdad Reshadi, and Ben Hardekopf. Improved type specialization for dynamic scripting languages. In *Proceedings of the 9th Symposium on Dynamic Languages, DLS '13*, pages 37–48, New York, NY, USA, 2013. ACM.
- 33 Andreas Klöckner, Nicolas Pinto, Yunsup Lee, Bryan Catanzaro, Paul Ivanov, and Ahmed Fasih. Pycuda and pyopencl: A scripting-based approach to gpu run-time code generation. *Parallel Computing*, 38(3):157–174, 2012.

- 34 Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: A llvm-based python jit compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC, LLVM '15*, pages 7:1–7:6, New York, NY, USA, 2015. ACM.
- 35 Alan Leung, Ondřej Lhoták, and Ghulam Lashari. Automatic parallelization for graphics processing units. In *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java, PPPJ '09*, pages 91–100. ACM, 2009.
- 36 Thibaut Lutz and Vinod Grover. Lambdajit: A dynamic compiler for heterogeneous optimizations of stl algorithms. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Functional High-performance Computing, FHPC '14*, pages 99–108. ACM, 2014.
- 37 X. Ma, J. Li, and N. F. Samatova. Automatic parallelization of scripting languages: Toward transparent desktop parallel computing. In *21th International Parallel and Distributed Processing Symposium (IPDPS 2007), Proceedings, 26-30 March 2007, Long Beach, California, USA*, pages 1–6. IEEE, March 2007. doi:10.1109/IPDPS.2007.370488.
- 38 T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, Dec 1976.
- 39 Mojtaba Mehrara, Po-Chun Hsu, Mehrzad Samadi, and Scott Mahlke. Dynamic parallelization of javascript applications using an ultra-lightweight speculation mechanism. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture, HPCA '11*, pages 87–98. IEEE Computer Society, 2011.
- 40 Yeoul Na, Seon Wook Kim, and Youngsun Han. Javascript parallelizing compiler for exploiting parallelism from data-parallel html5 applications. *ACM Trans. Archit. Code Optim.*, 12(4):64:1–64:25, 2016.
- 41 Radford M. Neal. pqr, 2016. URL: <http://www.pqr-project.org/>.
- 42 NVIDIA Corporation. Cuda, 2017. URL: <https://developer.nvidia.com/cuda-zone>.
- 43 NVIDIA Corporation. Nvidia opencl sdk code samples, 2017. URL: <https://developer.nvidia.com/opencl>.
- 44 Michael F. P. O’Boyle, Zheng Wang, and Dominik Grewe. Portable mapping of data parallel programs to opencl for heterogeneous systems. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), CGO '13*, pages 1–10. IEEE Computer Society, 2013.
- 45 Richard Plangger and Andreas Krall. Vectorization in pypy’s tracing just-in-time compiler. In *Proceedings of the 19th International Workshop on Software and Compilers for Embedded Systems, SCOPES '16*, pages 67–76, New York, NY, USA, 2016. ACM. doi:10.1145/2906363.2906384.
- 46 Philip C. Pratt-Szeliga, James W. Fawcett, and Roy D. Welch. Rootbeer: Seamlessly using gpus from java. In *Proceedings of the 2012 IEEE 14th International Conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems, HPCCC '12*, pages 375–380. IEEE Computer Society, 2012.
- 47 Michel Steuwer, Christian Fensch, Sam Lindley, and Christophe Dubach. Generating performance portable code using rewrite rules: From high-level functional expressions to high-performance opencl code. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015*, pages 205–217. ACM, 2015.
- 48 John E. Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *IEEE Des. Test*, 12(3):66–73, 2010.
- 49 Secure Systems and Software Laboratory. Zippy, 2015. URL: <https://github.com/securesystemslab/zippy>.
- 50 Justin Talbot, Zachary DeVito, and Pat Hanrahan. Riposte: A trace-driven compiler and parallel vm for vector code in r. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques, PACT '12*, pages 43–52. ACM, 2012.


- 51 Peng Tu and David Padua. Automatic array privatization. In *Compiler optimizations for scalable parallel systems*, pages 247–281. Springer, 2001.
- 52 Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. Polyhedral parallel code generation for cuda. *ACM Trans. Archit. Code Optim.*, pages 54:1–54:23, 2013.
- 53 Sven Verdoolaege and Tobias Grosser. Polyhedral extraction tool. In *In Second International Workshop on Polyhedral Compilation Techniques (IMPACT '12)*, 2012.
- 54 Haichuan Wang, David Padua, and Peng Wu. Vectorization of apply to reduce interpretation overhead of r. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015*, pages 400–415. ACM, 2015.
- 55 Haichuan Wang, Peng Wu, and David Padua. Optimizing r vm: Allocation removal and path length reduction via interpreter-level specialization. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO '14*, pages 295:295–295:305. ACM, 2014.
- 56 Zheng Wang, Daniel Powell, Björn Franke, and Michael O’Boyle. *Exploitation of GPUs for the Parallelisation of Probably Parallel Legacy Code*, pages 154–173. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.
- 57 Thomas Würthinger, Christian Wimmer, Christian Humer, Andreas Woss, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon, and Matthias Grimmer. Practical partial evaluation for high-performance dynamic language runtimes. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017*, pages 662–676. ACM, 2017.
- 58 Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. One vm to rule them all. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software, Onward! 2013*, pages 187–204. ACM, 2013.
- 59 Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. Self-optimizing AST interpreters. In *Proceedings of the 8th symposium on Dynamic languages - DLS '12*, page 73. ACM Press, 2012.
- 60 Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. Self-optimizing ast interpreters. In *Proceedings of the 8th Symposium on Dynamic Languages, DLS '12*, pages 73–82, New York, NY, USA, 2012. ACM. doi:10.1145/2384577.2384587.
- 61 Wei Zhang. *Efficient Hosted Interpreter for Dynamic Languages*. PhD thesis, University of California Irvine, 2015.
- 62 Wei Zhang, Per Larsen, Stefan Brunthaler, and Michael Franz. Accelerating iterators in optimizing ast interpreters. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA '14*, pages 727–743. ACM, 2014.
- 63 Haiping Zhao, Iain Proctor, Minghui Yang, Xin Qi, Mark Williams, Qi Gao, Guilherme Ottoni, Andrew Paroski, Scott MacVicar, Jason Evans, and Stephen Tu. The hiphop compiler for php. *SIGPLAN Not.*, pages 575–586, 2012.

CROCHET: Checkpoint and Rollback via Lightweight Heap Traversal on Stock JVMs

Jonathan Bell

George Mason University, Fairfax, VA, USA


bellj@gmu.edu

 <https://orcid.org/0000-0002-1187-9298>

Luís Pina

George Mason University, Fairfax, VA, USA

lpina2@gmu.edu

 <https://orcid.org/0000-0003-4585-5259>

Abstract

Checkpoint/rollback (CR) mechanisms create snapshots of the state of a running application, allowing it to later be restored to that checkpointed snapshot. Support for checkpoint/rollback enables many program analyses and software engineering techniques, including test generation, fault tolerance, and speculative execution.

Fully automatic CR support is built into some modern operating systems. However, such systems perform checkpoints at the coarse granularity of whole pages of virtual memory, which imposes relatively high overhead to incrementally capture the changing state of a process, and makes it difficult for applications to checkpoint only some logical portions of their state. CR systems implemented at the application level and with a finer granularity typically require complex developer support to identify: (1) where checkpoints can take place, and (2) which program state needs to be copied. A popular compromise is to implement CR support in managed runtime environments, e.g. the Java Virtual Machine (JVM), but this typically requires specialized, non-standard runtime environments, limiting portability and adoption of this approach.

In this paper, we present a novel approach for *Checkpoint Rollback via lightweight HEap Traversal* (CROCHET), which enables fully automatic fine-grained lightweight checkpoints within unmodified commodity JVMs (specifically Oracle's HotSpot and OpenJDK). Leveraging key insights about the internal design common to modern JVMs, CROCHET works entirely through bytecode rewriting and standard debug APIs, utilizing special proxy objects to perform a lazy heap traversal that starts at the root references and traverses the heap as objects are accessed, copying or restoring state as needed and removing each proxy immediately after it is used. We evaluated CROCHET on the DaCapo benchmark suite, finding it to have very low runtime overhead in steady state (ranging from no overhead to 1.29x slowdown), and that it often outperforms a state-of-the-art system-level checkpoint tool when creating large checkpoints.

2012 ACM Subject Classification Software and its engineering → Frameworks

Keywords and phrases Checkpoint rollback, runtime systems, dynamic analysis

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2018.17

Supplement Material Code available at <https://github.com/gmu-swe/crochet>

Acknowledgements We would like to thank the anonymous reviewers for their feedback. Riley Spahn and Michael Hicks provided many helpful comments on this document as well.



© Jonathan Bell and Luís Pina;

licensed under Creative Commons License CC-BY

32nd European Conference on Object-Oriented Programming (ECOOP 2018).

Editor: Todd Millstein; Article No. 17; pp. 17:1–17:31

Leibniz International Proceedings in Informatics



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

1 Introduction

Checkpoint/rollback (CR) tools capture the state of an application and store it in some serialized form, allowing the application to later resume execution by returning to that same state. CR tools have been employed to support many tasks, including fault tolerance [50, 46], input generation and testing [53, 49], and process migration [44, 48, 38, 16, 26, 9]. For instance, fault-tolerance tools can checkpoint at critical system decision points, allowing for automated recovery in the event of an otherwise unrecoverable failure. As another example, an input fuzzer can run a program unimpeded until the program reaches an interesting function f , and then perturb f 's input, using checkpoint and rollback to re-execute f many times soundly (i.e. while holding all other state constant). Similarly, most tools that perform code synthesis or automated program repair [31, 28, 40, 45] benefit greatly from speculative execution – testing whether the generated code meets the correct post-conditions, and, if not, resetting the program state to generate a more suitable replacement.

Typically, these CR tools rely on support from the operating system (OS), such as POSIX `fork` and various memory management functions. To perform a checkpoint, a CR tool write-protects all pages that the application uses. When the application modifies its memory, the OS notifies the CR tool, which copies the application's data as needed. Alternatively, an application can perform checkpoints by forking and resuming execution on the child process. Due to the copy-on-write nature of the `fork` system call, the parent process holds a checkpoint of the heap. Later, to rollback, the child terminates, effectively discarding its changes to the program state; and the parent forks again, resuming execution on a new child with the program state at the time of the original checkpoint (i.e. `fork`).

Although both of these approaches impose no overhead in the steady state (when no checkpoint is performed), they are inefficient when checkpointing many sparsely populated pages [18]. That is, even though an application may overwrite only 4KB in total, such a CR tool may need to copy up to 16MB if the application overwrites a single byte on 4,000 different pages. Furthermore, mapping the state of an OS-level checkpoint back to the JVM (e.g., for comparing two different executions) is a complex problem in itself [12]. Rather than rely on OS support for lightweight checkpoints, we examine the case of checkpointing in managed language runtime environments, specifically, the Java Virtual Machine (JVM).

Prior work in JVM checkpointing required a specialized, custom JVM [18, 26, 9], or developer support [55]. Our goal is to provide efficient, fine-grained, and incremental checkpoint support within the JVM, using only commercial, stock, off-the-shelf, state-of-the-art JVMs (e.g., Oracle HotSpot and OpenJDK). Guided by key insights into the JVM Just-In-Time (JIT) compiler behavior and the typical object memory layout, we present CROCHET: Checkpoint ROLLbaCk with lightweight HEap Traversal for the JVM. CROCHET is a system for in-JVM checkpoint and rollback, providing copy-on-access semantics for *individual variables* (on the heap and stack) that imposes *very low steady-state overhead* and requires no modifications to the JVM. CROCHET allows developers to checkpoint either: (1) the state reachable from all current heap roots (i.e. static fields, stack pointers, even objects held by the garbage collector for finalization), or (2) an object graph encapsulated by a few well identified roots (e.g., a list encapsulated by its head as root). CROCHET also can manipulate active stack frames, allowing it both to checkpoint values on the stack and to resume execution from a rollback (restoring the entire stack, creating and destroying frames as necessary). Moreover, CROCHET is thread-safe, and fully automatic. It allows developers to checkpoint or rollback dynamically at *any time* in execution, without requiring any advance annotations or restrictions on what data to include in that checkpoint.

At its core, CROCHET uses a novel *lazy heap traversal* algorithm that provides a general-purpose page-fault-like mechanism within the JVM, generating traps at the granularity of individual objects which can be enabled/disabled dynamically to checkpoint very large object graphs *in parallel* with the program’s execution and *without* pausing all threads while the checkpoint takes place. We demonstrate that CROCHET shows negligible runtime overhead in a steady state on both Oracle’s HotSpot JVM and OpenJDK, and reasonable performance to checkpoint and rollback an application.

We describe the design and implementation of a prototype of CROCHET that does not require any Java-specific features, and is thus directly applicable to any JVM-based language. Through bytecode rewriting, CROCHET leverages a novel deployment of automated *proxy types*, allowing it to checkpoint and rollback individual objects very efficiently and with very little steady state overhead (ranging from no overhead to 1.29x slowdown on the DaCapo benchmark suite [7]). By paying this marginal steady state overhead, CROCHET can create fine-grained checkpoints very efficiently (average overhead of 1.49x to checkpoint each benchmark state), often outperforming a state-of-the-art process-level checkpoint system (CRIU [16], average overhead of 2.25x to perform the same checkpoint).

In summary, the main contributions of this paper are:

- A general purpose approach to modify the runtime behavior of live objects in a JVM with very low overhead. This approach could be used to enable general ‘run-once’ dynamic analyses that are enabled infrequently and impose very low overhead when disabled.
- A general and efficient approach to checkpoint and restore the heap and stack state of a running application in a JVM.
- A detailed description and an extensive evaluation of our open-source implementation of this technique, CROCHET.
- Several case studies on possible applications that benefit directly from our approach.

2 Design

We set out to design CROCHET with several key goals in mind:

Goal 1 Require no modifications to the JVM itself;

Goal 2 Provide very low runtime overhead when a checkpoint/rollback is not in progress, and only a minimal slowdown when doing so;

Goal 3 Provide efficient checkpoints (i.e. copy only the data needed);

Goal 4 Allow developers to request a checkpoint or rollback at any arbitrary time.

With CROCHET, developers decide (dynamically) to checkpoint all heap and stack structures, only heap roots, or only a specified set of objects using the following high level interface (respectively): `checkpointAllRoots()`, `checkpointHeapRoots()`, and `checkpoint(Object... objects)`; and matching `rollback` functions. We expect that checkpoints and rollbacks will all occur within a single, continuously running JVM. We could imagine CROCHET being extended to asynchronously flush its checkpoints to disk, allowing rollbacks to occur in a separate process. We do not intend to directly support checkpointing state outside of the JVM (e.g. files and network connections), since if such behavior were desired, we could easily integrate existing systems (e.g., versioning filesystems).

CROCHET’s design is heavily influenced by our primary self-imposed constraint (Goal 1): it must operate entirely within the bounds of the API exposed by the JVM, without requiring any modifications to the JVM itself. Before presenting CROCHET, we first present three strawman approaches for implementing checkpoint and rollback within the JVM that fail to reach all of these goals. The simplest approach – **Strawman 1** – is to pause execution of

all threads immediately upon call to `checkpoint`, collect all variables, copy them, and then resume execution. Upon rollback, pause all threads again, and replace all variables with the previously collected copies. This simple approach trivially satisfies Goals 1, 2 and 4: from the time that checkpoint is called, all writes are subject to being replaced by their original values. However, this approach is inefficient, copying every single variable in the JVM, including those that may not ever be changed. Moreover, Strawman 1 pauses all threads in the JVM for the duration of an arbitrarily large checkpoint, which clearly defeats Goal 3.

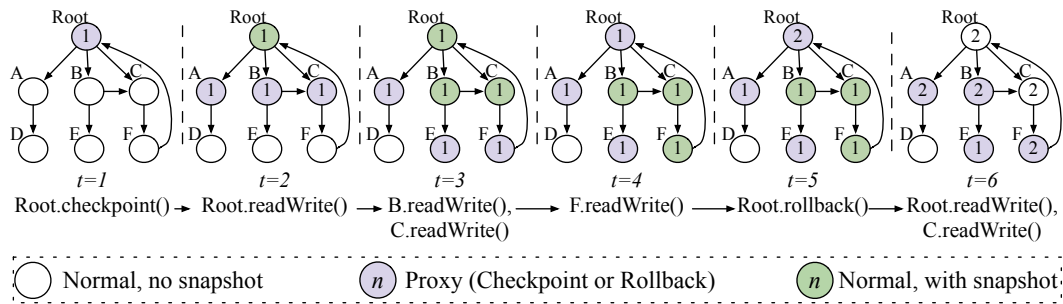
Strawman 2 provides a lazier approach: Guard all data accesses (i.e. field reads/writes, array reads/writes, local variable reads/writes), checking at the time-of-access if the variable needs to be saved or restored, and then doing so. The lazy Strawman 2 likely requires far less storage, as it copies only the minimum set of variables that change after the checkpoint. Similarly, Strawman 2 can be implemented with per-object locking, allowing other threads to make progress while they are not touching the same objects being checkpointed. However, Strawman 2 introduces prohibitive runtime: before any read or write, Strawman 2 needs to check if a checkpoint or rollback is taking place, and, if so, if the variable being accessed should be copied. Intercepting all field accesses this way can introduce up to 50% overhead on steady-state [42]. Therefore, Strawman 2 defeats Goal 2 by imposing a constant performance overhead, even when no checkpoint or rollback is occurring.

The copy-on-access semantics of Strawman 2 are similar to those that an Operating System (OS) uses when executing the `fork` system call: After `fork` is called, the child process shares its memory pages with the parent process (albeit with copy-on-write semantics). If the child attempts to write to any of those pages, a page fault is trapped and the page is copied and mapped to the child. For an OS, page access checks already occur as part of the memory address translation process, regardless of whether `fork` has been called or not. **Strawman 3** improves on **Strawman 2** by using OS-level `fork` support to provide inexpensive CR. However, `fork` does not duplicate all parent threads (only the forking thread is alive in the child); and all kernel state about a process, which results in some state being shared between parent and child (e.g., `epoll` descriptors).¹ Furthermore, mapping OS-level page fault handlers to variables in the JVM is not trivial. Dealing with these two issues would surely require modifications to the JVM, and likely to the OS, thus defeating Goal 1 (some JVM migration techniques do exactly this [9]). Moreover, if the objects that need to be saved populate many pages sparsely, this approach copies much more data than strictly necessary [18], thus defeating Goal 3.

CROCHET leverages an observation that the JVM already performs various checks before accessing data, and that we can exploit these checks. When performing dynamic dispatch for methods and fields overridden by several different classes, the JVM must decide which concrete implementation of the method/field to choose. For instance, the JVM selects different methods on the same callsite for method `toString`, depending on the type of the receiver object (e.g., `Integer` versus `LinkedList` versus `Object`). Even if the JVM can prove that a call site is monomorphic, it relies on profiling data to predict the likely receiver type. Further, monomorphic call sites can become polymorphic due to class loading. Hence, when the JVM optimizes a (non-static) call site or field access, rather than directly linking a specific method to call or field to access, it maintains instead a small lookup table, to point from class types to the specific code to be invoked.²

¹ <https://lkm1.org/lkm1/2007/10/27/25>

² <https://wiki.openjdk.java.net/display/HotSpot/PerformanceTechniques> provides a nice summary of JVM method optimization techniques



■ **Figure 1** High-level operation of CROCHET, showing the lazy traversal and propagation algorithm in six steps, as objects are manipulated after a checkpoint and a rollback.

Importantly, this means that the JVM already issues checks at every field and method access – and CROCHET exploits these checks³. CROCHET adds an empty method, `onReadWrite`, to each class and instruments all field accesses to call that method first. This empty `onReadWrite` method is then inlined by the JVM, effectively optimizing away any invocation overhead. This results in negligible steady-state overhead (often no overhead on DaCapo, ranging up to at worst, 1.13x steady-state slowdown). Later, when performing checkpoints, CROCHET generates *proxy classes* that extend the original classes and override the empty `onReadWrite` method with specific behavior (copying the object and propagating the traversal). CROCHET can then turn an object into a proxy simply by changing its type from its normal class C to the corresponding proxy class C_p (we explain the surprisingly simple mechanism to change the type of an object in §4.2).

Rather than change every object into a proxy object when checkpoint or rollback are invoked (which would require pausing all threads in order to do soundly), CROCHET performs a *lazy heap traversal*, as shown in Figure 1. During this traversal, every object is in one of three states: *Normal*, *Checkpoint* or *Rollback*. To start a traversal, CROCHET first transforms all heap roots into their proxy types by calling `Checkpoint` (transforming to *Checkpoint*) or `Rollback` (transforming to *Rollback*) on each root. Proxied objects have a special behavior when they are read or written (as shown in the right half of Figure 1): first, the object checkpoints (or rolls back) itself, and then it transforms all objects directly reachable by it into their corresponding proxy type.

Figure 1 shows how this lazy traversal propagates in a heap with one root object (*root*) and six other objects ($A - F$). A checkpoint takes place at instant $t = 1$, followed by manipulating (i.e. reading and/or writing) the *root* at $t = 2$, then objects B and C at $t = 3$, and F at $t = 4$. A rollback takes place at $t = 5$, followed by manipulating the *root* and C at $t = 6$. The background denotes the state of each object: white for regular objects without a snapshot present, purple for proxies, and green for objects with a snapshot present. Besides its state, each object O keeps a version counter with the number of the most recent traversal that reached O . Note that after each checkpoint/rollback there is always at least one proxy between the *root* and each object not yet reached by the traversal. This is in fact one of three invariants that are key for CROCHET’s correctness, as we elaborate in §3.

CROCHET uses a wait-free traversal algorithm to propagate checkpoint proxies, allowing it to safely and efficiently perform its operations in multi-threaded code; rollback operations are blocking, but due to the locality of Java objects, often experience little to no contention

³ Notable exceptions are private or static field/method accesses.

(§6). Further, since these proxy objects remove themselves from the object graph, most of the steady-state overhead that such proxies would otherwise introduce is eliminated, often resulting in zero runtime overhead in the absence of checkpoint or rollback operations (§5.2).

3 Lazy Heap Traversal

We now describe the algorithm that CROCHET uses to lazily checkpoint and rollback the heap. CROCHET provides simple, flat-nested checkpoints: When performing two checkpoints in a row, CROCHET discards the object graph copy of the first checkpoint and keeps the copy of the second checkpoint. Hence, when an application calls checkpoint several times, and then eventually calls rollback, such a rollback restores values captured by the last checkpoint only. We leave supporting arbitrarily nested checkpoints to future work. While CROCHET requires pausing all application threads to collect all root references, the remainder of the algorithm is mostly non-blocking.

3.1 Invariants

Throughout the entirety of program execution, CROCHET maintains the following invariants:

1. **Identity:** Version numbers are not reused between different checkpoints and rollbacks;
2. **Total order:** New checkpoints and rollbacks introduce higher version numbers;
3. **Continuity:** For every object O , either (1) O has been reached by the current traversal, or (2) there is at least one proxy object (i.e. object with status CHECKPOINT or ROLLBACK) with the highest version on every path that reaches O .

Invariant 1 (**Identity**) identifies each checkpoint and rollback uniquely. Invariant 2 (**Total order**) provides a simple total order of all the checkpoints and rollbacks that is easy to check when propagating proxies throughout the object graph. Finally, Invariant 3 (**Continuity**) ensures that proxies will mediate every first interaction with objects during a checkpoint or rollback. CROCHET maintains these invariants even in the presence of multi-threading. The continuity invariant is especially important in multi-threading, as it ensures that two threads which might be racing to checkpoint or rollback the same object will be racing to perform *the exact same operation*. An important consequence of these invariants is that a proxy object can never access objects with a version higher than itself.

Tracking the status of heap traversal. To support its lazy heap traversal, CROCHET needs to track three facts about each object: (1) its **version** (the current version of the object), (2) its **snapshot** (a copy of the object, if it has been checkpointed), and (3) its **status** (representing its status in the traversal: either CHECKPOINT or ROLLBACK, or NONE to indicate the object is not proxied). The checkpoint and rollback algorithms are implemented by automatically generated methods for each class: `onReadWrite`, `onCheckpoint`, `onRollback`, `copyFieldsTo` and `copyFieldsFrom`. CROCHET rewrites all field accesses to first call `onReadWrite`. `onReadWrite` is used to trigger checkpointing or rolling back (by calling `onCheckpoint` or `onRollback` respectively). `copyFieldsTo` and `copyFieldsFrom` are utility methods that allow CROCHET to copy the fields of each object; `propagateCheckpoint` and `propagateRollback` are the key methods used to advance the frontier of a heap traversal that we will now describe.

These methods behave differently depending on the status of the object that they are invoked on. Throughout this paper, we use dynamic dispatch notation in our examples. For

instance: Invoking `onReadWrite` on object `obj` results in method `CHECKPOINT.onReadWrite` being called when `obj.status` is `CHECKPOINT`. We make wide use of the `CompareAndSwap` (CAS) primitive, with the notation $CAS(f, v_o, v_n)$, which atomically updates field f to value v_n if f 's current value is v_o . CAS operations do not succeed if another thread updated field f from the previously observed value v_o . Finally, we omit similar methods for the sake of brevity; a complete reference containing all of the methods appears in Appendix A.

3.2 Algorithm for checkpointing

Figure 1 provides a high-level outline of how CROCHET's lazy traversal works, we will refer again to it in order to describe the checkpoint algorithm in detail. For simplicity, we first present the algorithm assuming a single-threaded execution (in which all compare-and-swap operations succeed); we will later provide a thorough argument about thread-safety in §3.4. To checkpoint the heap, the applications calls method `onCheckpoint` on all root references (§4.5 describes how those roots are found): object `root` in our example, which is in the `NORMAL` state at this point. Listing 1 shows the pseudo-code for method `NORMAL.onCheckpoint`. This is a fast operation that simply turns objects into proxies by changing their state (line 8), thus deferring the snapshot until it is needed.

In terms of the three invariants described above: CROCHET automatically manages version counters, based on how many times checkpoint or rollback operations have been started (on a global count), making Invariant 1 (Identity) and 2 (Total order) easy to enforce. Line 7 updates the version before updating the state, which does not violate Invariant 3, as objects only become proxies with the correct version. The opposite order would create a window during which a proxy would have a version lower than the highest, thus violating Invariant 3.

Once all root references are turned into proxies this way, Invariant 3 is established and the program can resume execution ($t = 1$). Next, the program manipulates the `root` object ($t = 2$), which triggers the invocation of method `CHECKPOINT.onReadWrite`. This method creates a snapshot of the object (lines 19–22); then propagates the checkpoint to all fields (line 26), effectively pushing the frontier of proxies one level forward in the object graph; and, finally, makes the object not a proxy (line 28). The last step does not violate Invariant 3 because all objects referred to by fields are now proxies themselves.

The program keeps executing, propagating proxies as it manipulates more objects. In our example (Figure 1) the program manipulates objects `B` and `C` in that order at $t = 3$; and object `F` at $t = 4$. Note that manipulating object `B` leads to invoking method `CHECKPOINT.onCheckpoint` on object `C`, which already is a proxy. This method is a fast operation that simply updates the version number when needed. In this case, it simply exits on line 33. However, consider if the example instead issued another checkpoint at $t = 2$, and then manipulated the `root` object. In that case, CROCHET would propagate new proxies to outdated proxies (object `A – C`), simply updating their version on line 35. Finally, note that manipulating object `F` turns the `root` back into a proxy at $t = 4$, as the check on line 4 fails. This behavior is correct but inefficient; we present an optimization to avoid this case in §3.5.

3.3 Algorithm for rolling-back

Performing a rollback is the dual of performing a checkpoint. At time $t = 5$, the program uses CROCHET to rollback to the earlier checkpoint. Similarly to the checkpoint, CROCHET starts the rollback by calling method `NORMAL.onRollback` on the `root` reference; which is the same as method `NORMAL.onCheckpoint`, but replaces `CHECKPOINT` with `ROLLBACK`. Again, Invariant 3 is established once all root references are turned into proxies.

```

1 NORMAL.onCheckpoint(int version){
2   int curV = this.version;
3   if (curV==version &&
4       this.status==CHECKPOINT)
5     return;
6
7   CAS(this.version, curV, version);
8   CAS(this.status, NONE, CHECKPOINT);
9 }
10
11 CHECKPOINT.onReadWrite() {
12   int curV = this.version;
13
14   Object snap = this.snapshot;
15   if (snap==NULL ||
16       snap.version<curV) {
17     // Allocates empty object
18     // Without running constructor
19     Object newSnap = ...
20     this.copyFieldsTo(newSnap);
21     newSnap.version = curV;
22     CAS(this.snapshot, snap, newSnap);
23   }
24
25   for (Field f in this)
26     f.onCheckpoint(curV);
27
28   CAS(this.status, CHECKPOINT,
29        NORMAL);
30 }
31 CHECKPOINT.onCheckpoint(int vers) {
32   int curV = this.version;
33   if (curV == vers) return;
34
35   CAS(this.version, curV, vers);
36 }
37 ROLLBACK.onReadWrite() {
38   int curV = this.version;
39
40   Object snap = this.snapshot;
41   if (snap != NULL &&
42       snap.version<curV) {
43     synchronized (snap) {
44       snap = this.snapshot;
45       if (snap != NULL &&
46           snap.version<curV) {
47         this.copyFieldsFrom(snap);
48         snap.version = curV;
49       }
50     }
51   }
52
53   for (Field f in this)
54     f.onRollback(curV);
55
56   CAS(this.status, ROLLBACK, NORMAL
57        );
58 }
59 ROLLBACK.onCheckpoint(int vers) {
60   int curV = this.version;
61   if (curV==vers &&
62       this.status==CHECKPOINT)
63     return;
64
65   this.onReadWrite();
66
67   CAS(this.version, curV, vers);
68   CAS(this.status, ROLLBACK,
69        CHECKPOINT);
70 }

```

■ **Listing 1** Pseudo-code for checkpoint algorithm.

■ **Listing 2** Pseudo-code for rollback algorithm.

After this invocation, the program manipulates the *root* object at $t = 6$, thus invoking method `ROLLBACK.onReadWrite`. At this time, CROCHET reverts the state of the *root* object to the snapshot saved at $t = 1$ (lines 40 – 51). Then, CROCHET propagates proxies one level into the object graph (lines 53–54). Finally, the algorithm makes the *root* object not a proxy (line 56), which, as before, does not violate Invariant 3. Continuing the example, the program then manipulates object *C*, thus causing it to be rolled back to the saved version and propagating the proxies one more level in.

To understand the need for line 65, consider the case of the program issuing another checkpoint, at $t = 6$; and manipulating the *root* object. This results in CROCHET invoking method `ROLLBACK.onCheckpoint` for proxied object *B*, which must be rolled back to the snapshot taken at $t = 3$. Line 65 thus performs the needed rollback; the rest of the method is similar to method `NORMAL.onCheckpoint`.

Some methods are very similar and are thus omitted: method `NORMAL.onRollback` is similar to `NORMAL.onCheckpoint` as explained above, method `ROLLBACK.onRollback` is similar to `CHECKPOINT.onCheckpoint`, and method `CHECKPOINT.onRollback` is the same as method `NORMAL.onRollback`. A complete reference containing all of these methods appears in Appendix A.

3.4 Thread safety

Before scanning for root references, CROCHET pauses all threads to call checkpoint/rollback on each root, and then resumes all threads. Thereafter, multiple threads may race to perform the same checkpoint or rollback of the same (non-root) object, but it is impossible for different checkpoints and rollbacks to race with each other, or for the underlying program to race with the checkpoint/rollback of root references. We outline here a brief argument for CROCHET's thread safety, but leave a formal proof to future work. For additional support, our evaluation (§5) extensively tested checkpoint/rollback on multi-threaded applications.

CROCHET uses atomic compare-and-swap (CAS) operations to update values that are visible to other threads. For instance, when checkpointing a non proxied object in `NORMAL.onCheckpoint`, the algorithm first checks if there is any work left to do (line 3). If the check fails, the algorithm uses two CAS operations to update the version and the status from their expected values. A failed CAS just means that another thread performed that CAS; no recovery operation needs to take place, and no CAS' need to be retried.

Note that it is possible that another thread manipulates the same object between the check on line 3 and the first CAS at line 7 more than just performing the two CAS operations. For instance, the other thread may have already created a snapshot of this object by calling `onReadWrite` immediately after `onCheckpoint`, setting the status back to `NONE`. This interleaving is safe as the CAS operation on `onCheckpoint` that updates the version at line 7 will always fail, thus preventing the object from reaching an inconsistent state in violation of an invariant. Unfortunately, the object may cycle between states `CHECKPOINT` and `NORMAL`, which results in wasted work and increased overhead; §3.5 presents an optimization that addresses this problem.

The same argument for thread safety applies to `CHECKPOINT.onCheckpoint`, `CHECKPOINT.onRollback`, `NORMAL.onRollback`, `ROLLBACK.onCheckpoint` and `ROLLBACK.onRollback`. The argument for thread-safety when performing a checkpoint is straightforward. First, the algorithm creates a snapshot, which is private to each thread (line 19). Then, all threads race to update the object snapshot to their private copy with a CAS operation (line 22). One thread wins and keeps its snapshot, all the other threads discard their (equivalent) snapshots.

If an object is arbitrarily changed during the snapshotting process (which results in an invalid snapshot), CROCHET discards that snapshot. Consider two threads racing to update an object's fields: both threads may race through `CHECKPOINT.onReadWrite`. Neither thread will be able to update the object's fields until `CHECKPOINT.onReadWrite` returns. For one thread to mutate that object's fields during the other thread's snapshot stage, the first thread must have completed `CHECKPOINT.onReadWrite`, and recorded its snapshot. Hence, the second (invalid) snapshot is discarded, since it failed its CAS operation at line 22.

Then, the algorithm propagates the checkpoint to all the objects `obj` refers to (line 26). Invariants 1 and 2 ensure that all operations are idempotent, so it is safe for several threads to propagate the checkpoint, even if the object changes as explained in the previous paragraph. Finally, the checkpoint algorithm can revert the status back to `NORMAL` (line 28), because the previous loop ensures that Invariant 3 is not broken by doing so: the frontier of proxy objects has advanced on level forward in the object graph.

The argument for thread-safety when performing a rollback is more complex. It is similar to performing a checkpoint, but overwriting the object with its snapshot is not idempotent: Consider two threads T_1 and T_2 racing to rollback the same object. T_1 performs the rollback to completion and then executes application code that writes to a field of the object. Then, T_2 is scheduled and erroneously overwrites the object once again.

```

1 NORMAL.onCheckpoint(int v) {
2   int curV = this.version;
3   if (curV == version) return;
4   //realV =(curV < 0) ? curV*-1 : curV;
5   CAS(this.version, curV, v*-1);
6   CAS(this.status, NORMAL, CHECKPOINT);
7   CAS(this.version, v*-1, v);
8 }

```

■ **Listing 3** Optimization to avoid redundant proxy propagation.

The algorithm avoids such erroneous executions by acquiring a monitor on the snapshot (line 43). This allows the first thread to overwrite the object with its snapshot *and* set the snapshot to null *in one atomic step*. Contending threads, after acquiring the monitor, realize that the object is already rolled back (line 46) and proceed without changing anything.

The rest of the rollback algorithm is similar to the checkpoint algorithm and the same argument for thread-safety applies. Note that the progress condition of the checkpoint algorithm is *wait-freedom* because, regardless of scheduling, there is always a finite bound on the number of steps for a snapshot to be created and, therefore, for the algorithm to make progress. The algorithm for rollback is, of course, *blocking* due to the use of monitors. In the future, we plan to remove monitors from the algorithm and we speculate that the resulting algorithm will be *lock-free*, and not *wait-free*, due to the non idempotent nature of rollback.

3.5 Optimizations

The algorithm presented in Listings 1 and 2 is correct, but not efficient. In particular, the condition that detects if any work is needed for an object on line 3 fails to detect the case in which an object is re-discovered when propagating a checkpoint. Consider the following example, again from Figure 1: At $t = 4$, the program manipulates a field of Object F , calling `NORMAL.onCheckpoint` on the *root* object. At this point, there is no more work to be done for the *root* object. Yet, the early return on line 4 fails to detect this case, and proxies the *root* object, which will repeat the whole sequence of proxy propagations from $t = 1$ until $t = 4$. Our early experiments showed that this case happens frequently in practice and we added an optimization to improve the efficiency for this common case.

Ideally, checking the version of an object should be enough to decide if there is work to be done. However, removing the second check on line 4 is not safe. Suppose thread T_1 is scheduled out of execution after updating the version of an object o on line 7 but before updating its status on line 8. Thread T_2 now reads the updated version and decides that o does not require more work to be done. The objects that o refers to are now accessible to thread T_2 without a proxy between them and a root reference, i.e. Invariant 3 does not hold.

This problem can be solved trivially by updating both status and version in one atomic step (i.e. double-CAS). Given that support for such an atomic operation is not common, our algorithm uses the optimization shown in Listing 3 instead. The idea is to update the version with an intermediate value first, then update the status, then update the version to the correct value. Any thread that sees the intermediate version can extract the correct version from it. We applied the same optimization to all early return checks on the algorithm (lines 5, 33, and 63). When the actual version is needed (i.e. further uses of variable `curV` outside of a CAS), the pseudo-code on Listing 3 shows how to extract it into variable `realV`.


```

1 class OriginalClass { // Original class
2   OriginalClass f1; int f2;
3   int sum() { return f1.sum() + f2; }
4 }
5 class OriginalClass { // Generated NORMAL class
6   OriginalClass f1; int f2;
7   int sum() { f1.onReadWrite(); return f1.sum() + f2; }
8
9   int \[version; OriginalClass \]snapshot;
10
11  void \[onReadWrite() { /* empty */ }
12  void \[onCheckpoint(int v) { NORMAL.onCheckpoint(v); }
13  void \[onRollback(int v) { NORMAL.onRollback(v); }
14 }
15 class OriginalClass\]PROXY extends OriginalClass {
16 // Generated PROXY class, extends the generated NORMAL class above
17 void \[onReadWrite(){ if (version % 0) ROLLBACK.onReadWrite(this);
18   else CHECKPOINT.onReadWrite(this); }
19 void \[onCheckpoint(int v); // Similar to onReadWrite
20 void \[onRollback(int v); // Similar to onReadWrite
21 }

```

■ **Listing 4** An example class `OriginalClass`, shown before (top) and after (bottom) processing by CROCHET, including generated classes. Receivers `NORMAL`, `CHECKPOINT`, and `ROLLBACK` refer to the algorithms that will follow in Listings 1 and 2.

4 Implementation

CROCHET is implemented entirely within the confines of the JVM. Most of CROCHET is implemented through bytecode instrumentation using ASM [8], but some small components are written in C (mostly for stack introspection and manipulation), using the standard JVMTI [36] interface. While CROCHET can dynamically instrument bytecode on-the-fly (as it is loaded into the JVM), it is necessary to bootstrap the system by (automatically) instrumenting a complete copy of the JVM offline.

4.1 Class modifications and instrumentation

Listing 4 shows how CROCHET modifies the original bytecode of a Java program (shown as source code for the sake of clarity). CROCHET adds two fields to every class: (1) `version`, (2) and `snapshot`. To track the `status` of each object, CROCHET generates a proxy class that extends the original class and overrides the methods `onReadWrite`, `onCheckpoint`, and `onRollback` with the appropriate behavior. To update the value of field `status` of an object, CROCHET changes the class to which that object belongs. As an implementation optimization, CROCHET generates a just single proxy class for both `CHECKPOINT` and `ROLLBACK` and then uses the `version` to decide which state the object is in: Even numbered versions are `ROLLBACK`, odd numbered versions are `CHECKPOINT`. This reduces the overall number of classes that CROCHET needs to generate, which improves performance by minimizing the number of invocation targets that the JVM needs to consider at each call site.

CROCHET uses `sun.misc.Unsafe.defineAnonymousClass`, which loads classes quicker than regular classloaders⁴, performing load-time patching of a class’s constant pool to

⁴ Java 8 implements lambda routines through `defineAnonymousClass`: https://blogs.oracle.com/jrose/entry/anonymous_classes_in_the_vm

efficiently define proxy classes at run time. CROCHET uses a single definition for its proxy class and patches it to load proxy classes that extend different host classes without requiring any additional class generation. CROCHET rewrites the bytecode of the reflection layer and `sun.misc.Unsafe` to intercept field accesses and insert calls to `onReadWrite` dynamically, as required for the algorithm to operate correctly. Similarly, CROCHET’s runtime library intercepts reflective calls that inspect various classes to hide its presence (e.g. masking the extra fields and methods that it creates).

4.2 Changing object types

CROCHET is able to change the class of an existing object by leveraging an observation about the JVM object layout, building on our recent work in Dynamic Software Update systems for Java. Rubah [43] found that the type of an object could be changed at runtime by overwriting the header of that object, replacing the value stored in the *klass* sector. To ensure compatibility, the new class type must have an identical number and layout of fields to the old, but there are otherwise effectively no other restrictions [41]. This change can be made at any point, subject to the limitation that the code that modifies the object cannot become inlined with other code that needs to know its type (which is easily avoided by ensuring that the code making the swap is a sufficiently large method). Hence, CROCHET transitions objects from their regular definition into their proxy type using `sun.misc.Unsafe`’s `putInt` function. The *klass* value is stored 8 bytes into the object header: CROCHET simply replaces the *klass* on each object with the desired *klass*. The format of the object header is well documented [35], and represents the tightest coupling of CROCHET to the JVM, although we expect CROCHET to be adjustable to eventual object layout changes. CROCHET caches instances of both normal objects and their corresponding proxy object to allow it to always be able to map between the desired Java Class and the corresponding *klass* value. CROCHET does not cache *klass* values, as they are subject to change as classes are loaded and unloaded. In addition to the evaluation performed by Rubah [43] of this mechanism, we validated it empirically on the most common JVMs (Oracle and OpenJDK), and found it to hold.

4.3 Static Fields

Static fields are heap roots and pose a unique challenge as they are accessed directly without a receiver object that can be proxied. CROCHET must detect when a static field is first dereferenced (which would then cause it to be copied, either through checkpointing or rolling back). A naïve solution *immediately* checkpoints or rollbacks all static fields, which does not require CROCHET to monitor static fields. However, to do so safely, CROCHET would perform these copies while the entire application execution is paused, likely creating significant performance overhead. Instead, CROCHET wraps accesses to static fields using a special helper class, a `StaticFieldHelper`, which allows it to lazily detect when static fields are accessed. There is one `StaticFieldHelper` generated for each class, which has slots for all of the original class’ static fields to store them as regular instance fields. The `StaticFieldHelper` thus allows CROCHET to treat static fields as any other field: the `StaticFieldHelper` implements all of the methods (e.g., `onCheckpoint`, `onRollback`, etc.) that any other class would, and maintains its own proxy state (*Normal*, *Checkpoint* or *Rollback*). CROCHET generates these helper classes on-the-fly, storing the instance of the class in a static member field of each class for efficient retrieval, and rewrites all bytecode to use the instance fields of static helpers instead of the original static fields.

4.4 Wrapping arrays and non-instrumentable types

CROCHET relies on adding fields and methods to classes in order to modify the behavior of all of the various references that can exist in the JVM to perform its lazy heap traversal. Unfortunately, it is *not* possible to modify the behavior of all possible references directly; in particular, references from an array to its elements, or from an object that CROCHET was unable to modify for other reasons. Arrays in the JVM are lists of contiguous references and have an associated class to represent their type, but that class cannot be modified, and there is no lightweight mechanism to directly apply the proxy concept outlined above. Similarly, there are a small number of classes that cannot be modified due to tight coupling from the JVM internals to the class layouts (e.g., native code accessing hard-coded field offsets in `java.lang.Double`, `java.lang.Object`) [4].

For all non-modifiable types (objects or arrays), CROCHET: (1) creates wrappers that track the state of that reference type, and (2) checkpoints and rollbacks these instances eagerly when they are discovered in a traversal. CROCHET maintains a relatively performant (and thread-safe) lookup table between the non-modifiable instance and its corresponding wrapper using JVMTI's object tagging. Typically in our traversal, accessing a proxied object O_1 with field f pointing to object O_2 would cause O_1 to be copied, and O_2 to become a proxy. However, if O_2 is actually an instance of a non-modifiable type, then O_2 is copied immediately, and anything that O_2 points to is transitioned into a proxy. By eagerly propagating proxies into these types, CROCHET accesses the wrapper only during a checkpoint or rollback traversal. These eager checkpoints and rollbacks are implemented using the same algorithm as for other types, and the same thread safety argument applies to them.

4.5 Finding the Root References

To perform a *complete* heap traversal in the JVM, CROCHET needs to first identify all of the various roots that point into the heap. CROCHET considers as roots: (1) static fields of loaded classes, (2) objects in finalizer queues, (3) live instances of `java.lang.Thread`, and (4) objects referenced in, values held by, and monitors held by stack frames. The simplest type of root to collect are static fields: CROCHET simply enumerates all initialized classes, collecting their static field helpers and transitioning them into proxies. CROCHET performs its traversal of objects in finalizer queues by special-casing the finalizer queue itself, causing it to propagate proxies directly to its referees when they are accessed (rather than only when checkpoint is called). This way, CROCHET can correctly traverse objects as they are removed from the finalizer queue, regardless of what had been previously held a reference to these objects. Each live thread in the JVM has a corresponding `java.lang.Thread` object: CROCHET transitions them all into proxies. Stack references are handled as described below.

4.6 Stack references

CROCHET supports three configurations for handling stack state: (1) capture complete stack state of all threads (the variables in each stack frame, plus the complete stack trace), allowing for execution to be rolled back to that same exact instruction; (2) capture complete state for the current stack frame *only*, allowing execution to be rolled back only within this same method execution; or (3) capture only (developer-specified) stack variables of the current stack frame. In the Configuration 1, CROCHET checkpoints stack variables and the stack trace of each thread, and can roll the application back to an identical state (reproducing the same exact stack trace). CROCHET also captures all active monitors held by each thread, and at rollback ensures that those threads hold exactly the same monitors. Configuration 2

```

1 //Original Code
2 void someFunc(int i, int [] ar)
3 {
4   int j = i + 1;
5   ar[i] = ar[i] - j;
6   j--;
7   otherFunc(j, ar); //Checkpoint
                        //is called by otherFunc
8   ar[i] = 10;
9 }

```

■ **Listing 5** An example function, `someFunc` that will be in the stack trace when a checkpoint and is called (somewhere deeper in the stack, within the invocation of `otherFunc`).

```

1 //Checkpoint code
2 void someFunc(int i, int [] ar)
3 {
4   boolean captureStack = false;
5   int j = i + 1;
6   ar[i] = ar[i] - j;
7   j--;
8   otherFunc(j, ar);
9   if(captureStack)
10    Checkpointer.captureStack();
11   ar[i] = 10;
12 }

```

■ **Listing 6** The same code from Listing 5, but with the checkpoint code added. When checkpoint is called, CROCHET sets the `captureStack` boolean variable in each method active to `true`, causing it to capture stack variables. Not shown: code to capture active values on the operand stack.

```

1 //Rollback code
2 void someFunc(int i, int [] ar)
3 {
4   int j;
5   boolean captureStack = false;
6   if(Rollbacker.doRollback())
7   {
8     i = Rollbacker.localInt();
9     ar = Rollbacker.localIntArray();
10    j = Rollbacker.localInt();
11    Rollbacker.removeRollbackCode();
12  }
13  else
14  {
15    j = i + 1;
16    ar[i] = ar[i] - j;
17    j--;
18    otherFunc(j, ar);
19    if(captureStack)
20      Checkpointer.captureStack();
21  }
22  ar[i] = 10;
23 }

```

■ **Listing 7** The same code as shown in Listing 5, with dynamically generated rollback instrumentation. `doRollback` dynamically decides if the current invocation of this function `someFunc` should jump-forwards; `removeRollbackCode` determines dynamically if this method body has no more rollbacks to perform, and if so, removes the specialized rollback code using bytecode HotSwap. Not shown: code to restore active values on the operand stack.

is a relaxation of Configuration 1: rather than capture *all* stack variables, only the variables of the method that calls `Checkpoint` need to be captured. Configuration 3 is a further relaxation: CROCHET only calls `Checkpoint` on a pre-defined set of variables.

The call stack holds method invocations and their frames; the JVMTI tooling interface [36] allows CROCHET to manipulate (non-native) call stack frames (i.e. read and write local variables, method arguments, and held monitors; and pop frames; similar to how a debugger would do the same). Each stack frame includes an *operand stack*, which is used to pass arguments to JVM instructions and read their results. The operand stack is not accessible by any JVM debugging or reflection interface, so CROCHET accesses and manipulates it through bytecode instrumentation.

Checkpointing: CROCHET pauses all threads and creates a copy of the local variables and operand stack on every call stack frame, calling the normal `onCheckpoint` function for reference types. CROCHET also records each monitor (lock) held in each stack frame. In Configuration 2 (checkpointing only the calling frame), CROCHET makes these transformations only in the calling frame; in the case of the Configuration 3 (checkpointing only selected variables), CROCHET makes none of these transformations. Listings 5 and 6 show an example of how CROCHET modifies the program to support stack checkpoints. CROCHET inserts an extra flag per method as a local variable and checks it after every method invocation. If

the flag is set, the injected code will capture the current operand stack to local variables, checkpoint all local variables (which now includes the operand stack), reset the flag, and restore the operand stack as needed. To checkpoint the stack, CROCHET pauses all threads, toggles this flag for all active frames, and resumes all threads. Each active method will now see the flag set, checkpoint the relevant data, and then continue to execute. If the flag is set in a stack frame that is never re-activated before rollback (e.g. it is deep in the call stack and not returned to), then it would never be possible for those stack variables to have been modified, and hence CROCHET will never checkpoint it. CROCHET thus records the complete stack trace of each thread along with the variables in each stack frame.

Rolling back. In configurations 2 and 3 (which only involve checkpointing in the stack frame that called checkpoint), rolling back is straightforward: CROCHET calls `rollback` on each of the previously checkpointed objects and resets their values. To support Configuration 1 (capturing the complete stack), CROCHET pauses all threads and pops all stack frames so that they can be overwritten by the checkpointed stack. CROCHET then transforms the code for each method on the checkpointed stack trace (using Java's `ReTransformClass` functionality [37]), adding “roll-forward” code to skip all instructions until reaching the correct method invocation (active at the time of the checkpoint), and then restore all local variables and the operand stack with the values on the checkpointed frame. Listing 7 shows an example of this transformation (without locks or operand stack values). Then, CROCHET resumes each thread in sequence, guiding it to the correct point in execution, through calls of method `onRollback` on objects that are replaced on the stack, and then pauses it again. Once the rollback is complete (and all threads have stack frames equivalent to the checkpoint state), CROCHET removes the generated roll-forward code from the active methods and resumes all threads.

4.7 Limitations

Our implementation of CROCHET does not capture native code behavior through JNI. Native code that reads and writes fields does not trigger those references to be traversed, checkpointed, or rolled back. Similarly, root references held by native code are not considered during traversal. During the evaluation we present in §5, we did not find these limitations to be a concern. Still, these limitations can be removed by replacing JNI functions with wrappers (similarly to how CROCHET handles reflection). CROCHET's stack checkpointing does not handle JVM-internal threads that are invisible from Java code (e.g., compiler threads), although it does consider JVM-managed threads (e.g., finalizer threads). CROCHET's approach is fully compatible with the JVM's garbage collector, and functions correctly even while a garbage collection occurs.

CROCHET does not checkpoint state kept in Java class loaders. That is, if an application: (1) performs a checkpoint, (2) loads class *Foo*, and (3) performs a rollback; then class *Foo* will still be loaded. This limitation is due to tight coupling between class loaders and JVM internals. We believe that this is not a significant limitation of CROCHET, and, if desired, classes could easily be re-initialized between checkpoints using a system like VMVM [5].

Checkpoints also impact the garbage collection and finalization of objects. As expected, CROCHET must ensure that all objects reachable at the start of a checkpoint remain reachable until the matching rollback happens, by keeping a reference to each such object. Allowing these objects to be garbage collected (before a rollback) would defeat the correctness of the checkpoint/rollback semantics of CROCHET, and hence, CROCHET retains references to them.

CROCHET’s implementation relies on the “unsupported” `sun.misc.Unsafe` library. While exposed for public use, its use is discouraged by the developers of the JVM, and none of its functionality is guaranteed to continue to exist in future versions of Java. While there has been much controversy around the JDK’s support for `sun.misc.Unsafe` [32], it is still included in Java 9, and there do not appear to be immediate plans to remove or deprecate the specific functionality used by CROCHET.⁵

Our eager handling of arrays and non-modifiable types requires more copies than strictly necessary – an entire array will be copied even if only one element is overwritten. However, given the constraints imposed by the JVM, we found this to be the most efficient approach to capturing references through arrays.

Finally, as discussed in the context of our design goals (§2), we do not consider the state of a system outside of the JVM (leaked through, for instance, file descriptors or network sockets). We envision that CROCHET could be integrated directly with versioning file systems and other low level approaches to capture this state.

5 Experimental Evaluation

We have conducted a thorough evaluation of CROCHET’s performance, using both micro benchmarks that we have crafted to expose specific performance scenarios, and macro benchmarks that simulate realistic workloads on large-scale apps (such as Apache Tomcat). We set out to answer five primary research questions:

RQ1: What is the steady state overhead imposed by the tooling to enable CROCHET’s lightweight heap traversal?

RQ2: What is the cost of performing a checkpoint with CROCHET?

RQ3: What is the cost of performing a complete checkpoint and rollback of a real application?

RQ4: Does CROCHET correctly checkpoint and rollback complicated data structures in the JVM?

RQ5: How does CROCHET compare to state-of-the-art software approaches that provide support for checkpoint/rollback?

We conducted all of our evaluations on a machine equipped with two Intel(R) Xeon(R) CPU E5-2650L v3 (each with 12 physical cores, 24 logical) running Ubuntu 16.04 (Linux 4.4.0-121-generic) and 128GB of RAM. We used Oracle’s HotSpot JVM version 1.8.0_66, as it is the latest version that macro benchmarks *tomcat* and *eclipse* run with.⁶ On all JVM configurations we used a max heap size (`-Xmx`) of 10GB and no other JVM options.

5.1 Microbenchmarks

We used several of the collections classes provided by the Java runtime environment to measure the steady-state overhead that CROCHET introduces, the cost of performing checkpoints, CROCHET’s correctness, and how it compares to CRIU and DeepClone (RQ1, RQ2, RQ4, and RQ5, respectively). We used the following data-structures: `HashMap` (HM in Table 1), `TreeMap` (TM), and `LinkedHashMap` (LHM) from the `java.util` package; and `ConcurrentHashMap` (CHM) from the `java.util.concurrent` package.

Each benchmark execution consists of 3 steps. First, the benchmark fills the data-structure with *SIZE* entries that map random integers, *keys*, to newly created objects with no fields,

⁵ <http://openjdk.java.net/jeps/260>

⁶ https://bugzilla.redhat.com/show_bug.cgi?id=1337940

■ **Table 1** Microbenchmark results showing run time comparison between a baseline Java 8 JVM (shown as runtime in msec with a 95% confidence interval) and the relative slowdowns imposed by CROCHET without checkpoints, and checkpoint/rollback using CROCHET (CROCHET_{CP}), DeepClone, and CRIU. The last row shows the average, minimum, and maximum overhead for each configuration.

Structure	Hotspot 8 (ms)		Relative Slowdown			
			CROCHET	CROCHET _{CP}	DeepClone	CRIU
CHM 10	65.84	(64.30, 67.39)	1.06 (1.03, 1.08)	1.35 (1.31, 1.38)	1.96 (1.90, 2.02)	1.74 (1.70, 1.79)
CHM 25	64.72	(64.34, 65.10)	1.10 (1.09, 1.11)	1.41 (1.40, 1.42)	2.03 (1.99, 2.08)	1.82 (1.80, 1.84)
CHM 50	65.18	(64.82, 65.55)	1.09 (1.08, 1.10)	1.44 (1.44, 1.45)	2.05 (2.00, 2.10)	1.83 (1.82, 1.85)
CHM 100	65.36	(64.99, 65.73)	1.11 (1.10, 1.12)	1.50 (1.49, 1.51)	1.98 (1.94, 2.03)	1.85 (1.84, 1.86)
HM 10	62.11	(61.70, 62.53)	1.08 (1.07, 1.10)	1.31 (1.30, 1.32)	1.92 (1.89, 1.94)	1.51 (1.50, 1.53)
HM 25	64.23	(63.86, 64.60)	1.04 (1.02, 1.06)	1.31 (1.30, 1.31)	1.88 (1.86, 1.90)	1.50 (1.48, 1.51)
HM 50	64.94	(64.59, 65.29)	1.04 (1.01, 1.06)	1.30 (1.29, 1.31)	1.88 (1.86, 1.89)	1.50 (1.49, 1.52)
HM 100	67.14	(66.22, 68.06)	1.01 (0.98, 1.05)	1.29 (1.27, 1.31)	1.83 (1.80, 1.86)	1.49 (1.47, 1.52)
LHM 10	51.70	(51.24, 52.16)	1.12 (1.10, 1.13)	1.54 (1.52, 1.55)	2.25 (2.22, 2.28)	2.04 (1.97, 2.11)
LHM 25	54.39	(53.86, 54.93)	1.09 (1.08, 1.11)	1.53 (1.51, 1.55)	2.15 (2.12, 2.18)	1.62 (1.57, 1.66)
LHM 50	56.54	(56.13, 56.94)	1.08 (1.07, 1.09)	1.48 (1.47, 1.50)	2.10 (2.07, 2.12)	1.62 (1.54, 1.71)
LHM 100	58.49	(57.97, 59.01)	1.07 (1.06, 1.08)	1.49 (1.48, 1.51)	2.07 (2.04, 2.10)	1.56 (1.54, 1.58)
TM 10	151.55	(149.93, 153.17)	1.03 (1.01, 1.04)	1.09 (1.07, 1.10)	1.45 (1.43, 1.48)	1.35 (1.33, 1.37)
TM 25	153.20	(151.34, 155.06)	1.02 (1.01, 1.04)	1.12 (1.10, 1.14)	1.45 (1.43, 1.48)	1.36 (1.34, 1.38)
TM 50	157.10	(154.43, 159.77)	1.03 (1.01, 1.05)	1.17 (1.14, 1.19)	1.45 (1.42, 1.48)	1.36 (1.34, 1.39)
TM 100	161.23	(159.11, 163.36)	1.06 (1.05, 1.08)	1.22 (1.21, 1.24)	1.44 (1.41, 1.47)	1.35 (1.33, 1.37)
Average			1.06 (0.98, 1.13)	1.35 (1.07, 1.55)	1.87 (1.41, 2.28)	1.59 (1.33, 2.11)

values. The range of the keys is $SPACE = [0, SIZE \times 2]$. Second, the benchmark creates a checksum by XOR-ring all the hash-codes of the keys (as defined in `Integer.hashCode`) with the values (identity hash-code). Third, the main loop performs $SIZE$ operations on the map, chosen randomly between get, put, delete, and replace. Note that, by construction, the hit-rate of each operation is 50%. We used the the microbenchmark framework Caliper⁷ to vary $SIZE$ to measure the execution time of a single execution of work. Caliper monitors the JVM and discards individual runs that involve garbage-collection and all runs with non-optimized JIT code, thus reporting execution times with a nanosecond accuracy. It performs warm-up runs to ensure the code is JITed, and then takes 10 trials for each $SIZE$ it selects. We used CROCHET to perform a checkpoint between steps 2 and 3, and a rollback after step 3. On a separate execution, we computed another checksum of the resulting data-structure and compared it with the original checksum, to ensure that the rollback mechanism is working correctly.

To measure the performance when the workload requires only a subset of the data-structure, we configured step 3 to use a subset of $SPACE$: 10%, 25%, 50%, or 100%.

We compare CROCHET with the popular Java DeepClone library,⁸ used by other Java tools that would benefit from our approach [13, 12]. This library immediately copies every field of every object using automatically-generated code. Note that, in this case, the checksums do not match because the cloning mechanism does not maintain the identity hash-code of the serialized objects. We also compare CROCHET with the state-of-the-art process-level checkpoint and rollback tool, CRIU [16]. Since the task is to simply serialize the data structure under test, we use CROCHET's `checkpointObjects(...)` routine to checkpoint only the

⁷ <https://docs.google.com/document/d/1M0e2UNf1ZxixotjB09r4FKzJG07VyhrXxbKqwX1LAzo/pub>

⁸ <https://github.com/kostaskougios/cloning>

data structures under test (and not collect all roots). Note that CRIU is, by definition, doing more work than CROCHET and the DeepClone library, since it is checkpointing the entire process. In the macrobenchmark evaluations that follow (§5.2), we compare it with CROCHET also configured to checkpoint entire applications, but here we want to intentionally demonstrate the overhead of a process-level checkpoint technique when only one portion of an application needs to be checkpointed.

We executed this entire process 20 times for each configuration (on top of the 10 runs that Caliper does for each *SIZE* it selects). Table 1 shows the results of our microbenchmark evaluation, showing the average raw time to run the benchmark (for the baseline, HotSpot 8 configuration) as well as the average slowdown imposed by each configuration ($Time_{configuration}/Time_{HotSpot8}$ with 95% confidence intervals). Each row represents a different data structure paired with a different *SPACE* value (e.g. CHM 25 represents the ConcurrentHashMap benchmark at *SPACE*=25%). When not using checkpoint/rollback, CROCHET imposes a very modest overhead, with a maximum slowdown of 1.11x, averaging 1.06x (RQ1). CROCHET performs checkpoint and rollback at a moderate cost (from 1.09x to 1.54x slowdown) (RQ2) for correct checkpoints, i.e. matching checksums (RQ4). Moreover, CROCHET beats the direct competition, often by a wide margin.

5.2 Macrobenchmarks

Our synthetic microbenchmarks shed light on the raw performance of CROCHET on data structures. To measure CROCHET’s performance in realistic workloads, we also evaluated it using the DaCapo benchmark suite, version 9.12-bach [7]. We followed the benchmark authors’ best practices: We ran a warmup phase before collecting results, repeating the benchmark execution until the measured time reaches a coefficient of variation of 2.0 at most over a sliding window of three executions. We repeated this process 20 times per benchmark and present 95% confidence intervals for timings.

We measured the performance of four configurations: (1) baseline JVM without CROCHET, (2) CROCHET without performing checkpoints, (3) CROCHET calling `checkpointHeapRoots` at the start of the benchmark, and (4) using the system-level checkpoint/restart tool *CRIU* [16] to perform a checkpoint at the start of the benchmark. We excluded the benchmark *tradesoap* because it failed to converge on a stable time even for the normal execution. We did not use the DeepClone library since it was incompatible with most of the benchmarks.

Table 2 shows the results of our macrobenchmark evaluation. We report a 95% confidence interval for the average benchmark execution time in the baseline (HotSpot 8) configuration, and 95% confidence intervals for the average slowdown factor. In the steady-state (not performing any checkpoints), the results mostly mirror the microbenchmark results reported in §5.1, with a maximum slowdown of 1.29x. Note that CROCHET imposes a slowdown higher than 1.07 only on two benchmarks (*jython* and *tomcat*), which can be explained by the common use of reflection in these benchmarks (which CROCHET needs to intercept). On average, over all the benchmarks, CROCHET imposes a slowdown of just 1.06x. CROCHET performs checkpoints at a modest cost, with an overhead ranging from 1.01x–3.45x (RQ2). Note that CROCHET imposes a relatively constant overhead when each class is loaded, which results in a higher slowdown for short benchmarks: *fop* and *xalan*. Overall, CROCHET performs checkpoints at an average cost of 1.49x.

In comparison, CRIU’s checkpoints imposed a slowdown ranging from 1.08x–5.36x, *always* higher than CROCHET. CRIU does not impose any steady-state overhead, and only becomes active during the call to checkpoint. However, CRIU dumps the entire resident heap of the JVM, which causes its performance to vary widely with (1) the size of that resident

■ **Table 2** Macrobenchmark results showing slowdown (newTime/originalTime) comparison between a baseline Java 8 JVM, the CROCHET system (without checkpoint ever called), the CROCHET system (with checkpoint called on all heap roots at the start of the benchmark), and CRIU (with checkpoint called just at the start of the benchmark). For CRIU, we report the size of the dump file. We show 95% confidence intervals for all timings. The last row shows the average, minimum, and maximum overhead for each configuration.

Benchmark	Relative Slowdown						
	HotSpot 8		Crochet		CRIU		
	Run time (ms)		No checkpoint	Checkpoint	Dump		
avroa	4,274	(4,208, 4,341)	1.01 (0.98, 1.03)	1.01 (0.98, 1.04)	1.08 (1.06, 1.11)	192.8 MB	
batik	1,256	(1,242, 1,270)	1.03 (1.01, 1.04)	1.23 (1.21, 1.26)	1.33 (1.31, 1.35)	420.2 MB	
eclipse	18,658	(18,523, 18,794)	1.01 (1.00, 1.02)	1.26 (1.24, 1.27)	1.17 (1.16, 1.18)	3.3 GB	
fop	223	(214, 233)	1.07 (1.02, 1.12)	2.06 (1.94, 2.19)	2.85 (2.73, 2.98)	385.5 MB	
h2	6,451	(6,413, 6,489)	1.04 (1.03, 1.05)	1.15 (1.14, 1.17)	1.21 (1.20, 1.22)	1.8 GB	
jython	3,285	(3,060, 3,510)	1.21 (1.09, 1.34)	1.60 (1.49, 1.72)	1.57 (1.45, 1.71)	1.5 GB	
luindex	761	(749, 773)	1.01 (0.99, 1.04)	1.10 (1.08, 1.12)	1.31 (1.28, 1.33)	176.5 MB	
lusearch	605	(601, 610)	1.01 (1.00, 1.01)	1.11 (1.09, 1.13)	5.36 (5.31, 5.40)	3.7 GB	
pmd	1,417	(1,402, 1,432)	1.04 (1.02, 1.05)	1.12 (1.10, 1.13)	1.63 (1.61, 1.65)	1.0 GB	
sunflow	944	(929, 959)	1.02 (0.99, 1.04)	1.13 (1.10, 1.15)	3.37 (3.27, 3.47)	3.0 GB	
tomcat	988	(979, 996)	1.29 (1.27, 1.30)	1.88 (1.86, 1.91)	2.27 (2.15, 2.40)	1.2 GB	
tradebeans	6,618	(6,535, 6,701)	1.03 (1.01, 1.05)	1.21 (1.19, 1.22)	1.29 (1.26, 1.33)	2.2 GB	
xalan	288	(276, 300)	1.02 (0.97, 1.07)	3.45 (3.28, 3.63)	4.75 (4.55, 4.96)	1.2 GB	
Average			1.06 (0.97, 1.34)	1.49 (0.98, 3.63)	2.25 (1.06, 5.40)		

heap (which may include lots of garbage), and (2) the duration of the benchmark. For instance, in the case of *lusearch* (5.36x slowdown), CRIU had to dump 3.66GB of data, and the underlying benchmark took only 605 msec in the baseline configuration. Compare this to CRIU's performance on *eclipse* (1.17x slowdown), where CRIU dumped a similar amount of data (3.25GB), but where the dump time was hidden in the significantly longer native benchmark execution time (18,658 msec). Note that CROCHET imposed a slowdown of just 1.11x for *lusearch*.

5.3 Transactional benchmarks

State-of-the-art Software Transactional Memories (STMs) intercept all data accesses (i.e. field/arrays/local variables reads/writes) to provide each thread executing a transaction a consistent view of the program state, and to isolate the changes that each thread performs in its separate transaction. Changes made by a transaction T become globally visible to new transactions when T finishes and commits successfully. Depending on the changes made by other transactions that commit between T 's start and finish, T may fail to commit; in which case the STM reverts all changes made by T . Transactions may finish by a voluntary abort, with the same outcome of an unsuccessful commit.

Assuming a single-threaded application, STMs can be used as an implementation of Strawman 2, described in §2, as follows: To checkpoint, start a new transaction; to rollback, abort the current transaction. We used the STMBench7 benchmark [24] to evaluate the feasibility of such an approach in comparison to CROCHET.

STMBench7 creates a realistic object graph that resembles the heap of a Computer Assisted Drawing (CAD) application, and then issues several concurrent operations that manipulate different regions of the object graph. Table 3 shows the results for this experiment, the following text explains each column in detail (higher is better).

■ **Table 3** STM comparison results, showing the baseline number of operations/sec completed without any concurrency control strategy (*HotSpot Ops/Sec*), the fraction (relative to that baseline) of operations completed with a transaction manager enabled but issuing no transactions (*No TX*), the fraction of operations completed with a single transaction issued to implement checkpoint/rollback (*One TX*), and the fraction of operations completed using the STMs to enforce atomicity (*Many TX*). The first row shows the results with CROCHET used to perform checkpoint/rollback (*One TX*), and without performing any checkpoint/rollback (*No TX*). More operations is better.

Configuration	HotSpot Ops/Sec	% of baseline operations/sec					
		No TX		One TX		Many TX	
crochet	243 (242, 244)	0.96 (0.96, 0.97)	0.91 (0.91, 0.91)				
deuce-lsa	237 (237, 238)	1.01 (1.00, 1.01)	0.06 (0.06, 0.06)	0.15 (0.15, 0.16)			
deuce-tl2	237 (237, 238)	1.00 (1.00, 1.01)	0.01 (0.01, 0.01)	0.13 (0.13, 0.13)			
jvstm	237 (237, 238)	0.34 (0.34, 0.35)	0.45 (0.45, 0.45)	0.45 (0.45, 0.45)			

The STMBench7 workload consists of different operations that read and write several different parts of the object graph, and should be atomic. STMBench7 ships with a backend that uses no concurrency control to ensure such atomicity – *no_lock*. This is our baseline, executed on a native JVM and with a single thread: Column **HotSpot Ops/Sec**.

STMBench7 also ships with backends to isolate concurrent operations with transactions using existing STMs: Deuce [21] and JVSTM [10]. Deuce is an STM framework that supports several synchronization algorithms, we used the two it ships with: LSA [47] and TL2 [19]. We used each STM backend as the concurrency control mechanism to ensure that each workload operation is atomic: Column **Many TX**. As before, we used a single thread. Note that this is the typical way to compare STM implementations with STMBench7, but since CROCHET does not (out of the box) enforce atomicity, we cannot use it as a point of comparison.

To perform checkpoint/rollback, we modified STMBench7 to checkpoint the object graph at the start of the workload, and to roll it back at the end: Column **One TX**. We used CROCHET to perform such checkpoint/rollback with the *no_lock* backend. We also used each STM to checkpoint the object graph by issuing a single transaction at the start of the workload, and keeping that transaction active throughout the whole workload. When the workload finishes, we rollback by aborting that single transaction. Again, we used a single thread. Note that we also added a checksum before and after the workload, and used it to ensure that the checkpoint/rollback mechanism worked as intended. Finally, we ran all the STMs experiments without creating any transactions, and CROCHET without performing any checkpoint/rollback, to measure the cost of the ability to perform checkpoint/rollback when not used: Column **No TX**.

We ran Deuce and JVSTM on Java 7 (Oracle HotSpot 1.7.0_80, the latest version), and CROCHET on the same Java 1.8.0_66 from the previous experiments (CROCHET is not backwards compatible with Java 7). We configured STMBench7 to run the *read-write* workload during 30 seconds with structural modifications disabled, as they slow down all STMs excessively. We repeated all runs 20 times and present 95% confidence intervals for all results. Table 3 shows the results for this experiment, showing the baseline number of operations performed with no locking under each baseline JVM and the relative fraction of those operations performed under each configuration (higher is better).

We also compared CROCHET to the recent XJ hybrid software transactional memory system [14]. XJ requires a custom-patched version of OpenJDK (specifically, OpenJDK1.7u40) to leverage hardware transactional memory features. We were not able to successfully run XJ outside of the VM that the authors provided (the custom version of OpenJDK

■ **Table 4** Results of the synchroBench benchmark, with checkpoint/rollback provided by CROCHET and XJ; compared against JDK 1.8.0 without checkpoints. More operations/sec is better.

Datastructure	HotSpot (ops/sec)	% of baseline ops/sec performed			
		Crochet		XJ	
SequentialHashIntSet	11.60 (11.52, 11.67)	0.91 (0.90, 0.92)			
ClosedHashIntSet	1.31 (1.31, 1.32)	0.91 (0.90, 0.92)	0.52	(0.50, 0.54)	

does not build on recent versions of the Linux kernel, ignoring the build error generates a JVM that consumes all available memory for any Java program we tried). Furthermore, XJ requires all programs run with it to be pre-processed, and the pre-processor fails to process STMbench7 our micro-benchmarks and Caliper. Therefore, we were able to conduct an evaluation only inside of the provided VirtualBox VM and only with the provided synchroBench benchmark. XJ requires that data-structures be hand-annotated in order to be checkpointed, and hence was not easily amenable to the STMbench7 workloads that we used. We used two data-structures with synchroBench, both hash-sets that hold integers: `hashtables.sequential.SequentialHashIntSet`, a sequential hash-set; and `hashtables.xj.ClosedHashIntSet`, a hash-set that supports *closed transactions* (i.e. the same type of transactions as JVSTM and Deuce, explained earlier). `SequentialHashIntSet` represents a typical data structure that would be used with CROCHET, while `ClosedHashIntSet` represents a data structure provided by the XJ authors with the correct annotations. We provide both data structures to demonstrate both the performance difference between CROCHET and XJ on a shared data structure, and baseline performance within this configuration. The workload initialized the data-structure with 65,536 elements and issued 95% read, 5% write operations during 5 seconds. We modified it to perform a checkpoint and checksum before the workload, rollback and another checksum after the workload, and ensure the two checksums match. To checkpoint/rollback, we used CROCHET and the single transaction technique described above for STMbench7. We allowed the benchmark to warm up during 60 seconds, and repeated the timed workload 100 times. Table 4 shows the average and 95% confidence interval for each configuration.

This experiment shows the prohibitive performance cost of using STMs to support checkpoint/rollback. Besides the performance penalty, using STMs for this purpose has further limitations: it does not support checkpointing state shared between threads (transactions naturally isolate it), and it requires manual modification of the target application (i.e. identifying transactions and transactional data, or even using different/slower data-structures). CROCHET, on the other hand, has a much lower performance cost, does not require manual changes to the application using it, and supports multi-threaded checkpoints.

6 Case Studies

At its core, CROCHET provides the ability to dynamically change the behavior of any object or code in the JVM, with negligible steady-state overhead, and requiring only a minimal pause to perform the initial update. Whereas the JVM provides a hotswap code functionality to redefine methods of all objects of a given class, CROCHET is able to redefine methods on a *per-object* basis. In particular, CROCHET uses this technique to provide lightweight checkpoint and rollback functionality. However, the technique we present in this paper is more general and, in this section, we describe several of the various applications that immediately stand to benefit from CROCHET, as well as several that stand to benefit from its high level instrumentation approach.

■ **Table 5** Execution time (and relative overhead) of four different fuzzing strategies: native (no isolation between runs), crochet-baseline (no isolation between runs, but with CROCHET running), crochet (with CROCHET providing isolation), and restart (restarting the server between each run).

Configuration	Exec Time (ms)	Rel Overhead
native	41.52 (41.37, 41.66)	
crochet-baseline	42.69 (40.32, 45.07)	1.03 (0.97, 1.09)
crochet	43.77 (43.65, 43.89)	1.05 (1.05, 1.06)
restart	78.33 (78.20, 78.46)	1.89 (1.88, 1.89)

6.1 Fuzzing and Test Generation

There are a wide range of approaches toward generating inputs to test program behavior. For instance, symbolic analysis tools, such as KLEE [11], JPF [53] and CUTE/JCUTE [49] generate new inputs systematically to explore different program behavior based on path constraints. Other tools are search-based, turning input generation into an optimization problem that maximizes code coverage [22]. Yet other tools take a simpler approach: fuzzers perturb known inputs to generate new ones [39, 27]. A key challenge for these tools is scalability: there can be an immense input space to search.

All of these tools typically re-execute a program many times from a given point in execution, for instance, to explore different inputs to a function, or to force a program to follow a different branch. Prior approaches either re-execute the program from the beginning each time, or maintain a symbolic heap. EvoSuite generates entire test stubs, and re-executes those test stubs, changing them between executions to expose new behavior. KLEE, JPF, CUTE and JCUTE all maintain a symbolic heap – a map from variables to a collection of values, one per different program state. Both approaches are inefficient: Re-running a program implies the cost of running that execution, and redirecting all heap accesses through a map adds a significant performance penalty (100x is common [54]).

CROCHET allows these approaches to explore different inputs to the same function much more efficiently: Perform a checkpoint when reaching the function to explore for the first time, explore one point in the input space by calling the function once, observe the results, then rollback to the previous checkpoint, and call the function again with another input. Note that CROCHET is also useful if the function is easy to reach (e.g., request handling loop on a server): Typical black-box fuzzers (i.e. based on the format of the requests accepted by the server) assume that each request is independent from all that precede it. If this assumption does not hold, the fuzzer may discover some input that crashes the server on a fuzzing run, but not in isolation; thus limiting the usefulness of such techniques. Using CROCHET to perform a checkpoint just after the server starts, and rolling back to that pristine program state after each fuzzed command, ensures that all errors found will be reproducible in isolation from just the fuzzed command.

As a case study, we modified an existing FTP black-box fuzzer⁹ to perform the same fuzzing run on three scenarios: (1) fuzz all commands on the same server, (2) restart the server after each fuzzed command, and (3) checkpoint the initial server state and rollback after each command. We modified an FTP server written in Java¹⁰ to checkpoint its heap on startup, and to rollback when signaled by the fuzzer. Table 5 shows the results for a short fuzzing run, with 122 total FTP commands sent, as the average of 100 runs and the

⁹ `ftp_pre_post` shipped with MetaSploit 4.16.31: <https://github.com/rapid7/metasploit-framework>

¹⁰ CrossFTP version 1.07: <https://sourceforge.net/projects/crossftpserver/>

respective 95% confidence interval. The fuzzing run takes takes 41.52 seconds to execute under Scenario 1, 78.33 under Scenario 2 (an increase of 89%), and just 43.77 seconds under Scenario 3. The results are encouraging, showing that that CROCHET imposes no measurable overhead to ensure proper input isolation.

6.2 Checkpoint/Rollback as an Application Service

Transactional applications may benefit from system-provided checkpoint and rollback abstractions. Database applications naturally fit this format. As a case study, we considered the H2 database, which is an in-memory SQL database written in Java.

The DaCapo benchmark suite contains an H2 benchmark, in which the benchmark driver: (1) creates an in-memory database, (2) populates it with test data from the TPC-C benchmark, (3) performs a number of TPC-C operations using multiple threads, (4) computes a checksum of the database and compare it with the expected value, and, finally, (5) restores the initial state of the database after 2. Each iteration of the benchmark repeats steps 3–5.

To reset the state of the benchmark, DaCapo’s H2 benchmark duplicates a number of columns on each table to hold the original data. The benchmark workload does not use those columns. At step 5, the benchmark resets the original columns by copying the data from their duplicate columns. Step 5 is developer-provided customized code to checkpoint and reset the state of the database. Therefore, it provides a perfect opportunity to test CROCHET. We wrote our own version of the H2 benchmark that uses CROCHET’s generic support to checkpoint and rollback in-memory data (i.e. the database tables) *instead* of the customized code that adds the duplicated columns.

We ran this benchmark in the same environment as specified in the previous section. It completed correctly (passing all checks) averaging $8,256 \pm 91\text{ms}$, adding a slowdown of around 1600ms (1.3x). In doing so, CROCHET proxied a total of 4,185,705 objects and copied 1,009,321 objects totaling just over 88MB, yielding a throughput of 90MB/sec copied. CROCHET was not able to beat the performance of the custom, efficient SQL queries that reset the state in bulk, which is not surprising. However, CROCHET’s approach for supporting efficient lightweight checkpoints is general enough for supporting any other in-memory Java implementation of the same TPC-C benchmark, even if it is not a SQL database (i.e. a key-value store).

6.3 Other Applications

There are also a variety of other potential applications for CROCHET which would require additional development, but could be very promising.

Time Travel Debugging. Time travel debuggers [2, 52, 3, 20, 30] allow developers to “step backward” (in execution) while debugging, which is accomplished through a combination of checkpoint-rollback and deterministic replay techniques: checkpoints are taken at regular intervals, and deterministic replay is used to fast-forward from the nearest checkpoint to the desired point of execution. While there are time travel debuggers for other VM-based languages (e.g. TARDIS for .NET [2]; JARDIS and ReJS for JS [3, 52]), there is currently no time travel debugger for the JVM. CROCHET could be used as the underlying checkpoint mechanism for a new time travel debugger for the JVM, *without* requiring changes to the JVM like these other systems.

Fault Tolerance. Existing high-level approaches for fault tolerance can also benefit from CROCHET’s approach. For instance, systems like ASSURE [50], Rx [46], ARMOR [13], and Mx [25] provide fault tolerance by performing regular checkpoints and detecting when the app fails. When a failure is detected, these systems tolerate it by reverting back to the most recent checkpoint and generating error recovery code.

A main limitation in these approaches is the regularity of checkpoints, which are based at the OS or VM level and, as explained in §2, do not map well to managed language and runtime environments, such as the JVM. Using CROCHET could lead to increased performance, and increased precision in the recovery procedure stemming from the fine-grained object-level information available about the data being restored.

Existing tools that are specialized to the JVM employ Java serialization to make frequent snapshots of specific variables that are considered important. ReCrashJ [1] makes a complete copy of method parameters as functions are called, providing developers with a log of the values of each parameter passed to each function if the application crashes. Capturing the complete object graph of each parameter to each function is quite expensive, imposing a slowdown of over 1,000x. Instead of eagerly capturing each parameter, CROCHET could be used to lazily capture them, only just before they are modified.

Smalltalk `become:`. The Smalltalk language provides a unique method, `become:`, that swaps the identities of its receiver and its argument. `become:` is a powerful global operation that updates all variables that refer to the receiver so that now they refer to the argument, and vice-versa. For instance, the Smalltalk implementation uses the `become:` method to increase the capacity of fixed-size of data-structures (e.g., array-lists and hash-maps): it simply allocates a new, larger, data-structure, copies all objects from the smaller data-structure, and then invokes `large become: small` [23]. All references to the original collection are replaced transparently by references to the new one.

Unfortunately, implementing `become:` is costly. Early implementations of Smalltalk keep a global object table, and represent references as indexes into such a table. `become:` was implemented as a simple pointer swap on the object table. However, modern high-level language VMs do not use such a global object table, relying on a garbage-collected heap instead. Instead, we might need to wrap every single object in a proxy object, and correct all references to that object to reference through the proxy. Or perhaps, we could modify the garbage collector to swap all references on the heap to an object `become:`’ing another through a full GC cycle.

Instead, CROCHET allows for an efficient `become:` through a lazy heap traversal that simply compares each object traversed with the receiver of `become:` and replaces it with the argument, and the same for replacing the argument with the receiver. Unlike the traditional approach, this approach does not require a pause that is proportional to the size of the heap.

7 Related Work

There has been a considerable body of research investigating the implementation and application of checkpoint and recovery tools. The tools can generally be divided into those that operate with operating system (and memory management unit) support, and those that operate primarily with developer support. Notable system-level tools include libpkt [44], Jockey [48], ZAP [38] and CRIU [16]. libpkt [44] and Jockey [48] create a fork of the process being checkpointed and use page faults to detect memory writes as they occur, performing incremental checkpoints. CROCHET is similar in spirit to these systems, as it also takes

incremental checkpoints, but checkpoints at the granularity of individual objects in the JVM, rather than entire pages of memory. ZAP [38] and CRIU [16] focus on checkpointing for process migration, while CROCHET's goal is to enable recovery within the same process as the checkpoint. We compared CROCHET with CRIU in §5.2.

Specifically targeting the JVM, Cunei and Vitek proposed an approach to checkpointing that is optimized for latency, utilizing a mirror of memory contents to satisfy both checkpoints and write requests concurrently [18]. While this approach required that the source-code of the JVM was modified, CROCHET requires no modifications to the JVM. Cunei and Vitek argued that checkpointing at the granularity of pages could impose higher than expected overheads when (relatively small) objects were sparsely distributed among (relatively larger) pages [18]. We make the same argument, and checkpoint at the granularity of individual objects, rather than memory pages.

Bringing developers into the loop, Xu et al. requires them to specify checkpoint and rollback locations statically for their Java CheckPoint system (JCP) [55]. JCP then performs an offline static analysis to determine which variables need to be included in each checkpoint. JCP also only supports single threaded applications. On the other hand, CROCHET supports multithreaded applications and does not require static specification of checkpoint and rollback sites. Upon rollback, JCP replays the (as computed) minimal slice of code needed to get the execution back to the same point where the checkpoint was called, while CROCHET generates custom code to bring the execution back to the same point, requiring for a stack depth of n only n method calls.

TARDIS [2] supports efficient time-travel debugging (i.e. 'step backwards') in the .NET CLR by piggybacking opportunistically on the garbage collector to create regular checkpoints. TARDIS requires modifications to the runtime (equivalent to modifying the JVM) to achieve its efficient checkpoints. Similarly, JARDIS [3] and ReJS [52] support time travel debugging in JS (ChakraCore), and collect snapshots very similarly to TARDIS. We believe that CROCHET could be extended (perhaps in combination with some deterministic record-and-replay tool such as Chronicle [6]) to support efficient time travel debugging in the JVM, without requiring modifications to the JVM.

There are also several systems targeting JVM migration, such as JAVMM [26] and ALMA [9], which bring insights from generic VM migration [51] into the JVM. These systems create a complete checkpoint of a running JVM, transfer that checkpoint to another JVM (perhaps on another machine) and then resume execution from that checkpoint. On the one hand, these systems also consider file and network state, and CROCHET does not. On the other, CROCHET works on stock JVMs, whereas both of these systems require modifications to the JVM.

There are also a variety of related systems that capture partial execution information in order to reproduce crashing executions, either capturing partial checkpoints [34, 15] or partial trace information [17, 56, 29, 15]. CROCHET could be used to support fault reproduction tools in the JVM as well.

CROCHET is also related (in implementation and design) to several non-checkpoint JVM-based systems. For instance, Rubah [43] provides dynamic software update in unmodified JVMs, and employs a lazy-heap traversal that inspired CROCHET's approach. CROCHET explores the notion of proxies far more widely and generically, focusing on the general performance and application of proxies to implement object-level page faults. Instant pickles [33] is an approach for pickling (serializing) objects in Scala. Instant pickles uses statically generated code to serialize and deserialize objects in the JVM faster than the JVM's dynamic serialization can. CROCHET uses a similar approach for copying objects.

8 Conclusion

The ability to perform fast, lightweight, fine-grained checkpoints on the JVM is not only useful to provide rich semantics to application developers, but also instrumental to support sophisticated automatic tools for applications such as fuzzing and fault tolerance. In this paper, we presented CROCHET, which significantly improves the state-of-the-art on this topic: CROCHET works on existing stock JVMs through bytecode rewriting and standard debug APIs, and the cost of running CROCHET when not using its checkpoint/rollback capabilities is very low. CROCHET automatically identifies the minimal state to be copied in a checkpoint fully automatically and works correctly with multi-threaded programs. CROCHET enjoys good performance with minimal pauses when performing checkpoints due to its lazy heap traversal algorithm. We believe CROCHET provides an adequate solution to a pressing problem that, in turn, will enable the realistic deployment of other tools that require efficient checkpoint/rollback support on an unmodified JVM.

References

- 1 Shay Artzi, Sunghun Kim, and Michael D. Ernst. Recrash: Making software failures reproducible by preserving object states. In *Proceedings of the 22Nd European Conference on Object-Oriented Programming*, ECOOP '08, pages 542–565, Berlin, Heidelberg, 2008. Springer-Verlag. doi:10.1007/978-3-540-70592-5_23.
- 2 Earl T. Barr and Mark Marron. Tardis: Affordable time-travel debugging in managed runtimes. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 67–82, New York, NY, USA, 2014. ACM. doi:10.1145/2660193.2660209.
- 3 Earl T. Barr, Mark Marron, Ed Maurer, Dan Moseley, and Gaurav Seth. Time-travel debugging for javascript/node.js. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 1003–1007, New York, NY, USA, 2016. ACM. doi:10.1145/2950290.2983933.
- 4 Jonathan Bell and Gail Kaiser. Phosphor: Illuminating Dynamic Data Flow in Commodity JVMs. In *ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 83–101, New York, NY, USA, October 2014. ACM. doi:10.1145/2660193.2660212.
- 5 Jonathan Bell and Gail Kaiser. Unit Test Virtualization with VMVM. In *36th International Conference on Software Engineering*, ICSE 2014, pages 550–561, New York, NY, USA, June 2014. ACM. ACM SIGSOFT Distinguished Paper Award. doi:10.1145/2568225.2568248.
- 6 Jonathan Bell, Nikhil Sarda, and Gail Kaiser. Chronieler: Lightweight recording to reproduce field failures. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 362–371, Piscataway, NJ, USA, 2013. IEEE Press. URL: <http://dl.acm.org/citation.cfm?id=2486788.2486836>.
- 7 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 *OOPSLA '06*, pages 169–190, New York, NY, USA, 2006. ACM. doi:10.1145/1167473.1167488.
- 8 Eric Bruneton, Romain Lenglet, and Thierry Coupaye. Asm: A code manipulation tool to implement adaptable systems. In *In Adaptable and extensible component systems*, 2002.

- 9 Rodrigo Bruno and Paulo Ferreira. Alma: Gc-assisted jvm live migration for java server applications. In *Proceedings of the 17th International Middleware Conference*, Middleware '16, pages 5:1–5:14, New York, NY, USA, 2016. ACM. doi:10.1145/2988336.2988341.
- 10 João Cachopo and António Rito-Silva. Versioned boxes as the basis for memory transactions. *Sci. Comput. Program.*, 63(2):172–185, 2006. doi:10.1016/j.scico.2006.05.009.
- 11 Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association. URL: <http://dl.acm.org/citation.cfm?id=1855741.1855756>.
- 12 Antonio Carzaniga, Alessandra Gorla, Alberto Goffi, Andrea Mattavelli, and Mauro Pezzè. Cross-checking Oracles from Intrinsic Software Redundancy. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE '14, pages 931–942, 2014.
- 13 Antonio Carzaniga, Alessandra Gorla, Andrea Mattavelli, Mauro Pezzè, and Nicolò Perino. Automatic Recovery from Runtime Failures. In *Proceedings of the 35th International Conference on Software Engineering*, ICSE '13, pages 782–791, 2013.
- 14 Keith Chapman, Antony L. Hosking, and J. Eliot B. Moss. Hybrid stm/htm for nested transactions on openjdk. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, pages 660–676, New York, NY, USA, 2016. ACM. doi:10.1145/2983990.2984029.
- 15 Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. Partial replay of long-running applications. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, ESEC/FSE '11, pages 135–145. ACM, 2011. doi:10.1145/2025113.2025135.
- 16 Jonathan Corbet. Checkpoint/restart (mostly) in user space. *LWN.Net*, 2011.
- 17 Olivier Crameri, Ricardo Bianchini, and Willy Zwaenepoel. Striking a new balance between program instrumentation and debugging time. In *Proceedings of the sixth conference on Computer systems*, EuroSys '11, pages 199–214. ACM, 2011. doi:10.1145/1966445.1966464.
- 18 Antonio Cunei and Jan Vitek. A new approach to real-time checkpointing. In *Proceedings of the 2nd International Conference on Virtual Execution Environments*, VEE '06, pages 68–77, New York, NY, USA, 2006. ACM. doi:10.1145/1134760.1134771.
- 19 Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking ii. In *Proceedings of the 20th International Conference on Distributed Computing*, DISC'06, pages 194–208, Berlin, Heidelberg, 2006. Springer-Verlag. doi:10.1007/11864219_14.
- 20 George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. Revirt: enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 5th symposium on Operating systems design and implementation*, OSDI '02, pages 211–224. ACM, 2002. doi:10.1145/1060289.1060309.
- 21 P. Felber, G. Korland, and N. Shavit. Deuce: Noninvasive concurrency with a java stm. In *Electronic Proceedings of the workshop on Programmability Issues for Multi-Core Computers (MULTIPROG)*, 2010.
- 22 Gordon Fraser and Andrea Arcuri. Evosuite: Automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 416–419, New York, NY, USA, 2011. ACM. doi:10.1145/2025113.2025179.
- 23 Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.
- 24 Rachid Guerraoui, Michal Kapalka, and Jan Vitek. Stmbench7: A benchmark for software transactional memory. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European*

- Conference on Computer Systems 2007*, EuroSys '07, pages 315–324, New York, NY, USA, 2007. ACM. doi:10.1145/1272996.1273029.
- 25 Petr Hosek and Cristian Cadar. Safe software updates via multi-version execution. In *International Conference on Software Engineering (ICSE 2013)*, pages 612–621, 5 2013.
 - 26 Kai-Yuan Hou, Kang G. Shin, and Jan-Lung Sung. Application-assisted live migration of virtual machines with java applications. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 15:1–15:15, New York, NY, USA, 2015. ACM. doi:10.1145/2741948.2741950.
 - 27 Hojun Jaygarl, Sunghun Kim, Tao Xie, and Carl K. Chang. Ocat: object capture-based automated testing. In *Proceedings of the 19th international symposium on Software testing and analysis*, ISSTA '10, pages 159–170. ACM, 2010. doi:10.1145/1831708.1831729.
 - 28 Guoliang Jin, Linhai Song, Wei Zhang, Shan Lu, and Ben Liblit. Automated atomicity-violation fixing. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 389–400, New York, NY, USA, 2011. ACM. doi:10.1145/1993498.1993544.
 - 29 Wei Jin and Alessandro Orso. Bugredux: reproducing field failures for in-house debugging. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 474–484, Piscataway, NJ, USA, 2012. IEEE Press. URL: <http://dl.acm.org/citation.cfm?id=2337223.2337279>.
 - 30 Samuel T. King, George W. Dunlap, and Peter M. Chen. Debugging operating systems with time-traveling virtual machines. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 1–1, Berkeley, CA, USA, 2005. USENIX Association. URL: <http://dl.acm.org/citation.cfm?id=1247360.1247361>.
 - 31 Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for \$8 each. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 3–13, Piscataway, NJ, USA, 2012. IEEE Press. URL: <http://dl.acm.org/citation.cfm?id=2337223.2337225>.
 - 32 Luis Mastrangelo, Luca Ponzanelli, Andrea Mocchi, Michele Lanza, Matthias Hauswirth, and Nathaniel Nystrom. Use at your own risk: The java unsafe api in the wild. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 695–710, New York, NY, USA, 2015. ACM. doi:10.1145/2814270.2814313.
 - 33 Heather Miller, Philipp Haller, Eugene Burmako, and Martin Odersky. Instant pickles: Generating object-oriented pickler combinators for fast and extensible serialization. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 183–202, New York, NY, USA, 2013. ACM. doi:10.1145/2509136.2509547.
 - 34 Satish Narayanasamy, Gilles Pokam, and Brad Calder. Bugnet: Recording application-level execution for deterministic replay debugging. *IEEE Micro*, 26(1):100–109, 2006. doi:10.1109/1702.1702003.800.
 - 35 OpenJDK Team. CompressedOOPS. <https://wiki.openjdk.java.net/display/HotSpot/CompressedOops>.
 - 36 Oracle. Jvm tool interface. <http://docs.oracle.com/javase/8/docs/platform/jvmti/jvmti.html>, 2013.
 - 37 Oracle Corporation. Instrumentation API for the Java Platform SE 7. [https://docs.oracle.com/javase/7/docs/api/java/lang/instrument/Instrumentation.html#retransformClasses\(java.lang.Class...\)](https://docs.oracle.com/javase/7/docs/api/java/lang/instrument/Instrumentation.html#retransformClasses(java.lang.Class...)). Accessed on 2018/01/11.

- 38 Steven Osman, Dinesh Subhraveti, Gong Su, and Jason Nieh. The design and implementation of zap: A system for migrating computing environments. *SIGOPS Oper. Syst. Rev.*, 36(SI):361–376, dec 2002. doi:10.1145/844128.844162.
- 39 Carlos Pacheco and Michael D. Ernst. Randoop: feedback-directed random testing for java. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, OOPSLA '07, pages 815–816. ACM, 2007. doi:10.1145/1297846.1297902.
- 40 Yu Pei, Carlo A. Furia, Martin Nordio, Yi Wei, Bertrand Meyer, and Andreas Zeller. Automated fixing of programs with contracts. *IEEE Trans. Softw. Eng.*, 40(5):427–449, 2014. doi:10.1109/TSE.2014.2312918.
- 41 Luís Pina. *Practical Dynamic Software Updating*. PhD thesis, Instituto Superior Técnico, University of Lisbon, 2016.
- 42 Luís Pina and João Cachopo. Atomic dynamic upgrades using software transactional memory. In *Proceedings of the 4th International Workshop on Hot Topics in Software Upgrades*, HotSWUp. IEEE, 2012.
- 43 Luís Pina, Luís Veiga, and Michael Hicks. Rubah: DSU for Java on a Stock JVM. In *OOPSLA*, 2014. doi:10.1145/2660193.2660220.
- 44 James S. Plank, Micah Beck, Gerry Kingsley, and Kai Li. Libckpt: Transparent checkpointing under unix. In *Proceedings of the USENIX 1995 Technical Conference Proceedings*, TCON'95, pages 18–18, Berkeley, CA, USA, 1995. USENIX Association. URL: <http://dl.acm.org/citation.cfm?id=1267411.1267429>.
- 45 Yuhua Qi, Xiaoguang Mao, and Yan Lei. Efficient automated program repair through fault-recorded testing prioritization. In *Proceedings of the 2013 IEEE International Conference on Software Maintenance, ICSM '13*, pages 180–189, Washington, DC, USA, 2013. IEEE Computer Society. doi:10.1109/ICSM.2013.29.
- 46 Feng Qin, Joseph Tucek, Jagadeesan Sundaresan, and Yuanyuan Zhou. Rx: Treating bugs as allergies—a safe method to survive software failures. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles, SOSP '05*, pages 235–248, New York, NY, USA, 2005. ACM. doi:10.1145/1095810.1095833.
- 47 Torvald Riegel, Pascal Felber, and Christof Fetzer. A lazy snapshot algorithm with eager validation. In *Proceedings of the 20th International Conference on Distributed Computing, DISC'06*, pages 284–298, Berlin, Heidelberg, 2006. Springer-Verlag. doi:10.1007/11864219_20.
- 48 Yasushi Saito. Jockey: A user-space library for record-replay debugging. In *Proceedings of the Sixth International Symposium on Automated Analysis-driven Debugging, AADeBUG'05*, pages 69–76, New York, NY, USA, 2005. ACM. doi:10.1145/1085130.1085139.
- 49 Koushik Sen, Darko Marinov, and Gul Agha. Cute: A concolic unit testing engine for c. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ESEC/FSE-13*, pages 263–272, New York, NY, USA, 2005. ACM. doi:10.1145/1081706.1081750.
- 50 Stelios Sidiroglou, Oren Laadan, Carlos Perez, Nicolas Viennot, Jason Nieh, and Angelos D. Keromytis. Assure: Automatic software self-healing using rescue points. *SIGARCH Comput. Archit. News*, 37(1):37–48, 2009. doi:10.1145/2528521.1508250.
- 51 Petter Svärd, Benoit Hudzia, Johan Tordsson, and Erik Elmroth. Evaluation of delta compression techniques for efficient live migration of large virtual machines. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE '11*, pages 111–120, New York, NY, USA, 2011. ACM. doi:10.1145/1952682.1952698.

- 52 John Vilks, James Mickens, and Mark Marron. A gray box approach for high-fidelity, high-speed time-travel debugging. Technical report, Microsoft Research, June 2016. URL: <https://www.microsoft.com/en-us/research/publication/gray-box-approach-high-fidelity-high-speed-time-travel-debugging/>.
- 53 Willem Visser, Klaus Havelund, Guillaume Brat, Seungjoon Park, and Flavio Lerda. Model checking programs. *Automated Software Engg.*, 10(2):203–232, apr 2003. doi:10.1023/A:1022920129859.
- 54 Matej Vitásek, Walter Binder, and Matthias Hauswirth. Shadowdata: Shadowing heap objects in java. In *Proceedings of the 11th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '13, pages 17–24, New York, NY, USA, 2013. ACM. doi:10.1145/2462029.2462032.
- 55 Guoqing Xu, Atanas Rountev, Yan Tang, and Feng Qin. Efficient checkpointing of java software using context-sensitive capture and replay. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, pages 85–94, New York, NY, USA, 2007. ACM. doi:10.1145/1287624.1287638.
- 56 Cristian Zamfir and George Candea. Execution synthesis: a technique for automated software debugging. In *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, pages 321–334. ACM, 2010. doi:10.1145/1755913.1755946.

A Full pseudo-code for the checkpoint/rollback algorithm

```

1 NORMAL.onReadWrite() { /*empty*/ } 53
2 54
3 NORMAL.onCheckpoint(int version){ 55 NORMAL.onRollback(int version){
4   int curV = this.version; 56   int curV = this.version;
5   if (curV==version && 57   if (curV==version &&
6   this.status==CHECKPOINT) 58   this.status==ROLLBACK)
7   return; 59   return;
8 60
9   CAS(this.version, curV, version); 61   CAS(this.version, curV, version);
10  CAS(this.status, NONE, CHECKPOINT); 62   CAS(this.status, NONE, ROLLBACK);
11 } 63 }
12 64
13 CHECKPOINT.onReadWrite() { 65 ROLLBACK.onReadWrite() {
14   int curV = this.version; 66   int curV = this.version;
15 67
16   Object snap = this.snapshot; 68   Object snap = this.snapshot;
17   if (snap==NULL || 69   if (snap != NULL &&
18   snap.version<curV) { 70   snap.version<curV) {
19   // Allocates empty object 71   synchronized (snap) {
20   // Without running constructor 72   snap = this.snapshot;
21   Object newSnap = ... 73   if (snap != NULL &&
22   this.copyFieldsTo(newSnap); 74   snap.version<curV) {
23   newSnap.version = curV; 75   this.copyFieldsFrom(snap);
24   CAS(this.snapshot, snap, newSnap); 76   snap.version = curV;
25   } 77   }
26 78 }
27 79 }
28 80
29   for (Field f in this) 81   for (Field f in this)
30   f.onCheckpoint(curV); 82   f.onRollback(curV);
31 83
32   CAS(this.status, CHECKPOINT, 84   CAS(this.status, ROLLBACK, NORMAL
   NORMAL); 85   );
33 } 86
34 87 ROLLBACK.onRollback(int vers) {
35 CHECKPOINT.onCheckpoint(int vers) { 88   int curV = this.version;
36   int curV = this.version; 89   if (curV == vers) return;
37   if (curV == vers) return; 90
38 91   CAS(this.version, curV, vers);
39   CAS(this.version, curV, vers); 92 }
40 } 93
41 94 CHECKPOINT.onRollback(int vers) {
42 ROLLBACK.onCheckpoint(int vers) { 95   int curV = this.version;
43   int curV = this.version; 96   if (curV==vers &&
44   if (curV==vers && 97   this.status==ROLLBACK)
45   this.status==CHECKPOINT) 98   return;
46   return; 99
47 100
48   ROLLBACK.onReadWrite(); 101
49 102   CAS(this.version, curV, vers);
50   CAS(this.version, curV, vers); 103   CAS(this.status, CHECKPOINT,
51   CAS(this.status, ROLLBACK, 104   ROLLBACK);
   CHECKPOINT);
52 }

```


ThingsMigrate: Platform-Independent Migration of Stateful JavaScript IoT Applications

Julien Gascon-Samson

Electrical and Computer Engineering Department, University of British Columbia,
2332 Main Mall, Vancouver, BC, Canada, V6T 1Z4
julien.gascon-samson@ece.ubc.ca

Kumseok Jung

Electrical and Computer Engineering Department, University of British Columbia,
2332 Main Mall, Vancouver, BC, Canada, V6T 1Z4
kumseok@ece.ubc.ca

Shivanshu Goyal

Electrical and Computer Engineering Department, University of British Columbia,
2332 Main Mall, Vancouver, BC, Canada, V6T 1Z4
shivanshu3@gmail.com

Armin Rezaiean-Asel

Electrical and Computer Engineering Department, University of British Columbia,
2332 Main Mall, Vancouver, BC, Canada, V6T 1Z4
armin.rezaiean.asel@gmail.com

Karthik Pattabiraman

Electrical and Computer Engineering Department, University of British Columbia,
2332 Main Mall, Vancouver, BC, Canada, V6T 1Z4
karthikp@ece.ubc.ca

Abstract

The Internet of Things (IoT) has gained wide popularity both in academic and industrial contexts. As IoT devices become increasingly powerful, they can run more and more complex applications written in higher-level languages, such as JavaScript. However, by their nature, IoT devices are subject to resource constraints, which require applications to be dynamically migrated between devices (and the cloud). Further, IoT applications are also becoming more stateful, and hence we need to save their state during migration transparently to the programmer.

In this paper, we present ThingsMigrate, a middleware providing VM-independent migration of stateful JavaScript applications across IoT devices. ThingsMigrate captures and reconstructs the internal JavaScript program state by instrumenting application code before run time, without modifying the underlying Virtual Machine (VM), thus providing platform and VM-independence. We evaluated ThingsMigrate against standard benchmarks, and over two IoT platforms and a cloud-like environment. We show that it can successfully migrate even highly CPU-intensive applications, with acceptable overheads (about 30%), and supports multiple migrations.

2012 ACM Subject Classification Computer systems organization → Distributed architectures, Computer systems organization → Embedded and cyber-physical systems, Computer systems organization → Dependable and fault-tolerant systems and networks, Software and its engineering → Middleware, Software and its engineering → Process management, Software and its engineering → Functional languages, Software and its engineering → Language features, Software and its engineering → Publish-subscribe / event-based architectures

Keywords and phrases JavaScript, Code Migration, Closures, IoT, Node.js

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2018.18



© Julien Gascon-Samson, Kumseok Jung, Shivanshu Goyal, Armin Rezaiean-Asel, and Karthik Pattabiraman;

licensed under Creative Commons License CC-BY

32nd European Conference on Object-Oriented Programming (ECOOP 2018).

Editor: Todd Millstein; Article No. 18; pp. 18:1–18:33



Leibniz International Proceedings in Informatics

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

Funding This work is supported by a research gift from Intel, a Discovery grant and Post-Doctoral Fellowship from the Natural Sciences and Engineering Research Council of Canada (NSERC), and funding from the Institute for Computing, Information and Cognitive Systems (ICICS) at the University of British Columbia (UBC).

1 Introduction

The Internet of Things (IoT) involves multiple devices across many domains that are interconnected to provide and exchange data. Over the last few years, the IoT market has grown considerably with some estimates putting the number of IoT devices in the tens of billions [28]. IoT devices are becoming more and more powerful, and in many cases they can run full-fledged real-time operating systems (e.g., the popular Raspberry Pi can run full Linux distributions). As a result, future IoT devices will be able to execute stateful, distributed applications written in high-level languages, which provide greater abstraction and portability than those written using platform-specific APIs.

In this paper, we focus on the use of JavaScript for programming IoT devices. While JavaScript has enjoyed wide popularity in the web context for a long time, it is now a mature and rich language in its own right. It has also become more and more prevalent in the world of IoT due to its portability across a wide range of devices [27, 10], as well as its large installed base of libraries and developers who know the language. Further, the language’s asynchronous nature makes it a prime candidate for IoT applications, which are often event-driven, and hence need to be highly reactive.

At the same time, placing more complex applications and long-running tasks on the end-devices themselves, close to the physical data (i.e., edge computing [38]) can incur lower latencies as opposed to running such applications in the cloud. However, as IoT devices are more resource-constrained, IoT applications have to be migrated between devices, and between devices and the cloud, for performance and security reasons. Thus, there is a compelling need to enable automated migration of stateful JavaScript devices between different IoT devices, and to/from the cloud, without requiring programmers to use platform specific APIs or runtimes. This is the main focus of this paper.

Migrating JavaScript applications poses several challenges, due to certain features of the language (i.e., closures) and its event-based nature. Further, due to the heterogeneity of IoT, we need to come up with a solution that does not involve accessing the internal states of the JavaScript Virtual Machine (VM), thus allowing portability across different devices. We tackle these challenges by proposing ThingsMigrate, a comprehensive middleware for the dynamic migration of IoT-based JavaScript applications across heterogeneous devices. ThingsMigrate automatically instruments the code at runtime to avoid modification of the VM while supporting advanced features of JavaScript, serializes its state, and reconstructs it on the target device after migration without any intervention from the programmer.

Other work has attempted to migrate browser-based JavaScript-based applications; however, they either do not fully address some important JavaScript features, such as nested closures ([32]), or they rely on VM-instrumentation [29] – thereby making their approach dependent on a specific VM/browser implementation. *To the best of our knowledge, ThingsMigrate is the first comprehensive high-level framework for migrating stateful JavaScript-based IoT applications, which addresses the aforementioned challenges, and without requiring any modifications to the JavaScript VM, thereby allowing platform-independent migrations.*

In summary, this paper provides the following contributions:

- A comprehensive JavaScript migration approach (Section 4) that is based on high-level code instrumentation and reconstruction, and that does not require VM modification, thereby allowing cross-platform migrations of JavaScript-based IoT applications.
- System implementation (Section 5) that handles many advanced features of the language and environment, such as arbitrarily nested closures, event queues, timers and MQTT-based communication interfaces, and support for multiple migrations.
- Evaluation through the execution of benchmarks across IoT and cloud-based devices (Section 6). Results indicate that ThingsMigrate can instrument arbitrary JavaScript programs, serialize their state and reconstruct them in a reasonable amount of time, while incurring an execution time penalty of 30%. Further, ThingsMigrate was capable of supporting multiple migrations with minimal memory usage increase.
- A case study (Section 7) which describes the experience of applying our approach in a real-world IoT context (motion detection over a video stream), predominantly using third-party libraries developed for server applications.

2 Motivation

Use of JavaScript. We assume that the various software components to be executed on the IoT nodes are written in JavaScript. JavaScript is one of the most popular languages today (in 2018), and ranks sixth in the TIOBE programming languages index [13]. It has also been ranked as the top language on both *GitHub* and *Stack Overflow* for the last five years. While the predominant use of JavaScript is for the web, JavaScript also maps well to the asynchronous, event-based nature of IoT applications [39], which in turn simplifies the development of asynchronous and concurrent applications. Further, similar to other high-level languages, JavaScript runs over a VM and is platform-independent, which allows for run-time code portability. In fact, the use of JavaScript also opens the possibility of easily sharing code, data and development resources between the different components of the IoT and web software stacks (e.g., the client-side and server-side portions of end-user web applications in a Web of Things (WoT) setting could both be written in JavaScript) [26, 21, 31]. Further, as many IoT devices nowadays provide a browser-based interface, it is fair to assume that they will integrate a JavaScript VM.

IoT Devices are Resource-Constrained. As mentioned, there have been many attempts at either adapting existing JavaScript VMs (e.g., Node.js [42] for IoT devices), or developing new JavaScript VMs [27, 10, 3, 12] for the IoT. However, given the resource-constrained nature of IoT devices and the fluid nature of the resource constraints, applications running on such devices might have to be frequently migrated from one device to another. For example, when a device runs low in memory, then the application running on it should be migrated to another device with more available memory to avoid the application from running out of memory and crashing. Similarly, any change to the available bandwidth or to the computational load of a given IoT device might require network and delay-sensitive applications to be migrated to a different device. Migration may also be needed when there are external factors causing device failures (e.g., device gets overheated or physically damaged), or due to security attacks on IoT devices.

Further, while resource management and code migration is a well understood problem in classical and cloud-based distributed systems, we believe that these techniques are not directly applicable to the IoT landscape, as they do not take into account IoT-specific constraints such

as the wide heterogeneity of hardware and software platforms, the highly resource-limited nature of the devices which makes it impractical to introduce additional virtualization layers, and the limited ability to provision resources on demand [41]. In addition, as the compute capacity of IoT devices is dictated by energy efficiency [30], we believe that a great deal of flexibility is required in scheduling. Therefore, given these considerations, we believe a static deployment of IoT applications to devices is insufficient, and that there is a need for applications to be migratable.

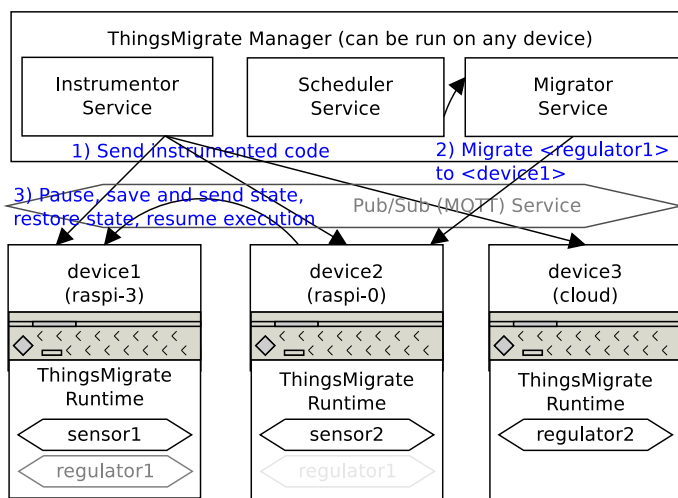
Preserving the State. As IoT devices and applications become more complex, they inherently generate more elements of *state* (i.e., variables, arrays, objects) as part of their execution. For instance, for an application which detects motion patterns in a video stream (Section 7), elements of state would include the pixels of the currently processed video frames (arrays), as well as any intermediate results produced as part of the computation. Considering that migration may be triggered at any time during the application’s execution in response to changing resource conditions or external events such as failures, it is important that there be a transparent mechanism to serialize and deserialize the state of an executing application. This mechanism should be efficient and support general JavaScript applications for IoT with only minimal modifications. Otherwise, developers would be required to implement application-specific serialization and deserialization logic, and for any arbitrary point in the execution, which would complicate the application logic.

Migration support in the VM. While migration support could ultimately be implemented in the VM, we believe that this is unlikely in the near future given the vast heterogeneity in IoT platform ecosystems. Unlike in the web browser space where there are only a handful of dominant players, the JavaScript IoT landscape is much more fragmented, with the availability of a wide variety of JavaScript engines (e.g., [27, 10, 3, 12]). Further, migration support would be required at both ends of the migration process; i.e., at the source device/VM, and at the target device/VM, which may be different from each other. Some of the VMs may be closed-source and hence not easily modifiable. That being said, should full or partial support for migration be provided in the VM or as part of the ECMAScript standard (i.e., by enabling special APIs to access the state of closures), then our technique could be adapted accordingly.

Applicability to other Languages. In this paper, we focus on the migration of JavaScript code for IoT. While our solution is specifically tailored for the challenges raised in migrating JavaScript applications (closures, objects, timers, events, etc.), we believe that some of the techniques that we propose could be adapted to support the migration of code written in other high-level languages. For instance, Python also provides support for closures and hence, we believe that our closure serialization and reconstruction approach could be adapted to Python. However, we do not consider applications written using low-level languages such as C or assembly. We also assume that developers do not use low-level APIs to directly access hardware state of IoT devices (e.g., by reading the pins of a device in a platform-specific manner) as such code would be difficult to migrate.

3 System Model

The system architecture of ThingsMigrate is presented in Figure 1. It is derived from the architecture of ThingsJS, a comprehensive IoT middleware that we presented as a vision



■ **Figure 1** High-Level Architecture of ThingsMigrate.

paper in [26]¹ (more details are given in appendix A). Our system model assumes a set of IoT devices, which are in charge of executing the various components of a distributed IoT application. We describe the systems components in this section.

3.1 ThingsMigrate Manager

The central piece of our architecture, namely the *ThingsMigrate Manager*, manages the execution of distributed IoT applications across the set of available devices. In our model, all communications between the components of the system use the topic-based publish-subscribe (pub/sub) paradigm (also referred to as MQTT) [24], which enjoys widespread usage in the IoT world [25, 40]. This is because it allows decoupling content producers (*publishers*) from content consumers (*subscribers*), and allows for abstracting network considerations. Overall, the *Manager* component has three components:

(1) Scheduler. This component schedules the execution of all IoT components across all devices. For the Scheduler to operate efficiently, developers are encouraged to modularize their IoT applications into a set of components, and to follow the best practices of JavaScript (Section 4.1). Taking into consideration the capabilities of each device, the requirements of the components, and a set of developer-specified *constraints*, the scheduler assigns the execution of each component onto a specific device. Upon the conditions changing, the scheduler can decide to dynamically move some of the components between devices. The migration takes place dynamically, and preserves the state of JavaScript IoT applications, so that the execution can be transparently transferred from one device to another - this is our main contribution. Note that the details of the *Scheduler* are outside the scope of this paper.

(2) Instrumentor. This component is in charge of instrumenting the JavaScript source code of the IoT components, which is the code that is executed by the devices. This is executed at the beginning before running a component on ThingsMigrate.

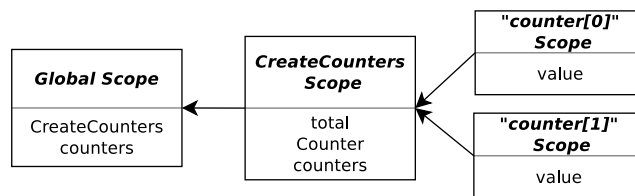
¹ While ThingsJS proposes migration as part of an integrated system, it does not specifically address migration challenges.

```

1  function CreateCounters(n) {
2      var total = 0;
3      function Counter() {
4          var value = 0;
5
6          return function() {
7              value += 1;
8              total += 1;
9
10             // Can access parent variables
11             console.log("val=" + val + " value=" + value + " total=" + total);
12             return value;
13         }
14     };
15
16     var counters = [];
17     for (var i=0; i<n; i++)
18         counters.push( Counter() );
19     return counters;
20 }
21
22 var counters = CreateCounters(2);
23 setInterval(function() {counters[0]},1000);
24 setInterval(function() {counters[1]},500);

```

■ **Figure 2** Counters JavaScript Example.



■ **Figure 3** Counters JavaScript Example Closures and Scopes.

(3) Migrator. The *Migrator* is in charge of transparently migrating the execution of each component from the original to the target device. To migrate a given component (e.g., component *regulator1* on device 2), the migrator issues a migrate command to the *ThingsMigrate Runtime* running on the target device (Section 3.2), with the name of the component to migrate. This then triggers the migration.

3.2 ThingsMigrate Runtime

The *ThingsMigrate Runtime* is a thin JavaScript middleware that executes on each IoT device and manages the local execution of all the components running on the device. It receives and executes the instrumented source code of the various components that it needs to execute from the *Instrumentor*, and awaits migration commands from the *Migrator* over a pub/sub interface. Upon a *migrate* command being received for a component (e.g., for component *regulator1* on device 2), the Runtime first freezes the execution of the component, serializes its state (Section 4.6) and sends it to the target device over the pub/sub interface (e.g., *device1*). When the Runtime on the target device (e.g., *device1*) receives the serialized state for a component, it restores the serialized state by generating appropriate restoration code (Section 4.7), which allows the execution to be resumed with the pre-migration state.

4 Approach

As mentioned, our code migration approach works entirely at the JavaScript layer through code instrumentation and does not depend on the underlying VM (unlike prior work by Kwon et. al. [29]). Thus, to support code migration, we need to instrument the code to expose the internal states (i.e., closures) that are not directly accessible using the JavaScript primitives and reflection APIs (Section 4.5). Code migration works in three phases. First, the component’s code is instrumented to support migration. Second, upon the *Migrator* service triggering a migration, a *snapshot* of the current state is taken (Section 4.6) and transmitted to the target device. Finally, the target device reconstructs the component state based on the snapshot, and resumes execution (Section 4.7).

4.1 Assumptions

We assume that the JavaScript code is compliant with strict-mode ES5 (ECMAScript 5) [5], which has been the de facto standard for many years. Although ES6 ([4]) is gaining momentum, it mostly adds syntactic sugar over ES5. Support for ES6 can easily be provided by leveraging transpilers (e.g., Babel.js [1]) to convert to ES5.

Because JavaScript is event-driven and single-threaded, we assume that developers will avoid blocking the main thread for long periods of time, as this would prevent the migration from being scheduled. Note that this assumption is not specific to ThingsMigrate – in fact, a long-running operation that never yields the control would inhibit the dispatching of any asynchronous event (e.g., timers, messages, I/O). To use ThingsMigrate, developers should follow the best practices and write their code in an event-driven manner, or break long-running operations to yield control (e.g., `setImmediate`) at periodic intervals, so that event processing can take place.

Finally, we assume that the program uses publish-subscribe (pub/sub) for communication, and does not perform write operations to the local file system. The former assumption is common in the IoT world, while the latter assumption requires the programmer to send file system operations over the network (Section 4.8 has more details).

4.2 Motivating Example

The JavaScript language treats functions as data, and provides support for closures, which allows for functions to be defined in other functions and to be bound to variables. As such, like any other object, functions can be passed as parameters and can be *returned* within other functions. JavaScript closures can also access the variables defined in parent functions in addition to their own variables, even if a parent function goes out of scope.

Figure 2 presents a motivating example of using JavaScript closures to implement a simple counter `Counter` (lines 3-14) which returns a *function* (lines 6-13) that increments the counter’s value by 1 (line 7) and prints it (line 11). In addition, there is a global variable `total` that holds the sum of all counters (line 8). Further, we wrap the `Counter` function and the `total` variable in another function, `CreateCounters`, which allows for creating and returning an array of n counters. More precisely, n nested closures are returned, which upon being called, increment the corresponding counter; therefore, the variable `counters` holds an array of $n = 2$ counters (line 20). Note that after line 20, some variables become out of scope (e.g., `total`, and all the copies of `value` for each counter), but they are not garbage-collected as the nested counter incrementation functions (i.e., `counters`) still access them.

Figure 3 visually illustrates the *scopes* of the various closures and their relationship. As can be observed, there are two independent copies of variable `value` each defined in their own scope, but only one copy of `total`, which is defined in the parent scope and is hence *shared* with the two child scopes. Finally, there are two recurrent timers (set using the `setInterval` JavaScript function) which increment the two counters at a regular interval i.e., by invoking nested functions `counters[0]` and `counter[1]` respectively every 1000 ms and 500 ms.

4.3 Challenges

Migrating the execution of a JavaScript program from one VM to another VM on to a different device poses many challenges when it comes to capturing and reconstructing the state. We discuss the challenges below in the context of the motivating example. To support migration, the current state of the application must be serialized. A naive approach to serialization would be to dump the process space of the application, which comprises the heap and the stack. However, such an approach would require serializing the entire memory space of the process, which would be platform-dependent and inefficient. Rather, ThingsMigrate provides a JavaScript-based approach that exploits the specifics of the language. For instance, as mentioned, a JavaScript application is made of objects and closures. More specifically, there is one root object that contains a set of properties, which are in fact objects themselves. As JavaScript treats functions as data, it allows functions to be bound to and stored within objects (i.e., *closures*). However, as shown in the motivating example above, functions can also contain other objects stored in variables.

(1) Closures. While JavaScript includes APIs to recursively and dynamically walk through the properties of objects and serialize them, the state of closures is *hidden* and thus cannot be *accessed* by means of user code. Thus, we need mechanisms to dynamically expose these hidden parts of the state *during* program execution.

(2) Migrating Events. We also need mechanisms to seamlessly transfer the state of pub/sub interfaces (i.e., subscribers and publications) during a migration. In addition, IoT systems often perform delayed executions (i.e., using timers); therefore, we need to support the seamless migration of timer-based events.

(3) Handling the Call Context. As JavaScript is mostly single-threaded and asynchronous (i.e., event-driven), there is no easy way to interrupt the current execution to perform a migration. In addition, as the call stack is not exposed, it is not directly accessible. Therefore, we need to come up with a mechanism to trigger the migration at certain points in the execution of the program.

(4) Reconstructing the state. After migrating the state, the execution must be restored given the serialized state. This is non-trivial, as an equivalent reconstructed program must be generated from the original code *and* the serialized state, and the execution must resume exactly at that state without any side effects (i.e., without re-executing code that can potentially lead to a different outcome).

(5) Enabling Multiple Migrations. As a given program might be migrated multiple times, we need to support multiple migrations. To reach that goal, the reconstructed program must be generated in such a way that it can be migrated again, with low overheads. For example,

the reconstructed program should not incur significantly higher memory or performance overheads than the original program, as such overheads would quickly add up when performing multiple migrations.

4.4 Problem Statement

As mentioned, in order to capture the state of a JavaScript program, one needs to capture the hierarchy of *scopes*, starting from the global scope, as well the data elements (variables and functions) contained within each scope. In other words, ThingsMigrate captures the *structure* and the *values* of the different state elements. More formally, we denote the state of a JavaScript application as $\mathbb{S} = \langle S, F, V, R \rangle$, where S is the set of *scopes*, F is the set of *functions*, V is the set of *variables* (i.e., a tuple of $\langle \text{name}, \text{value} \rangle$) and R is the set of *relations* between scopes and other entities (i.e., a tuple of $\langle \text{scope}, \text{entity} \rangle$).

Taking the code snippet shown in Figure 2 as an example, and assuming that a snapshot of the state is taken after 3250ms, then two instances of the `Counter` function (and their associated scopes) are defined, due to the timer invocations (i.e., `Counter_1` and `Counter_2`). Also, there are two instances of the anonymous function defined inside `Counter` (i.e., `Counter_1_anon` and `Counter_2_anon`). However, there is only one copy of the `CreateCounters` function (i.e., `CreateCounters_1`). Further, there are two copies of variable `value`, each within its own scope (`Counter_1_value` and `Counter_2_value`), and one copy of variable `total` (`CreateCounters_1_total`). The resulting state of the snapshot object $\mathbb{S} = \langle S, F, V, R \rangle$ would respectively contain states S , functions and their definition F (omitted for brevity), variables and their value V , and the set of relationships R between each variable/function and its associated scope:

```

S = {global, CreateCounters_1, Counter_1, Counter_2}
V = {(global_counters[0], Counter_1_anon), (global_counters[1], Counter_2_anon)},
    {(CreateCounters_1_total, 9), (CreateCounters_1_counters[0], Counter_1_anon)},
    {(CreateCounters_1_counters[1], Counter_2_anon), (Counter_1_value, 3)},
    {(Counter_2_value, 6)}
F = {(global_CreateCounters, ...), (CreateCounters_1_Counter, ...), (Counter_1_anon, ...)},
    {(Counter_2_anon, ...)}
R = {(global, CreateCounters_1), (global, global_counters), (global, global_CreateCounters)},
    {(CreateCounters_1, Counter_1), (CreateCounters_1, Counter_2)},
    {(CreateCounters_1, CreateCounters_1_total), (CreateCounters_1, CreateCounters_1_counters)},
    {(CreateCounters_1, CreateCounters_1_Counter), (Counter_1, Counter_1_value)},
    {(Counter_1, Counter_1_anon), (Counter_2, Counter_2_value), (Counter_2, Counter_2_anon)}

```

For more details, we refer the reader to Section 4.7 (Phase 3: Code Restoration), which describes in more detail our algorithmic approach to generating reconstruction code, and which gives an example of the restored code of the same code sample (Figure 2), migrated after the same delay (3250ms). As can be observed, the same functions, scopes and variables, as well as their relationships, are depicted in the restored code sample (Figure 5).

The next sections describe the algorithmic process followed by ThingsMigrate to (1) instrument the code to expose the hidden states, (2) take a snapshot and (3) reconstruct the code at the serialized state.

4.5 Phase 1: Code Instrumentation

In the code instrumentation phase, the ThingsMigrate Runtime augments the input JavaScript source file to allow the state to be dynamically captured (challenge 1), corresponding to the formal model defined in Section 4.4. Our code instrumentation approach is inspired

Algorithm 1: Code Instrumentation.

```

1 function instrumentCode (sourceCode)
2 begin
3   rootNode ← ASTParse(sourceCode)
4   <setupMigrationListener()>
5   <globalScope ← Scope(rootNode.name, null)>
6   instrumentNode(rootNode, null)
7   return ASTGenerate(rootNode)
8 end
9 function instrumentNode (parentNode, parentScope)
10 begin
11   foreach node in parentNode do
12     if node is Function then
13       <scope ← Scope(node.name, parentScope)>
14       instrumentNode(node, scope)
15       <parentScope.addFunction(node)>
16       <scope.checkAndDestroy()>
17     end
18     else if node is VariableDeclaration then
19       <parentScope.addVar(node.name, node.value)>
20     end
21     else if node is VariableAssignment then
22       varScope ← findScope(parentScope, node.name)
23       <varScope.setVar(node.name, node.value)>
24     end
25   end
26 end

```

by the work in Lo et. al. [32], but differs significantly as Lo et. al. [32] only offers limited support for capturing and restoring complex closures. In the example shown in Figure 2, Lo et. al. [32] would be able to capture and restore the scope of the two internal *counter* closures, but would not accurately model the relationship between said scopes in the restored output, so that two instances of the **CreateCounters** scope would be generated rather than one, ending with two distinct **total** variables after restoration. Each nested scope would then update its own **total** variable, which would be inaccurate.

The main aspects of our technique are illustrated in Algorithm 1. To fully capture the state of closures, the *Instrumentor Service* exposes the scope hierarchy by injecting code at relevant locations that will mirror the chaining of scopes and their contents (i.e., functions and variables) in a parallel tree-like data structure, in order to expose and capture the state.

Upon requesting the instrumentation of a given JavaScript source file (lines 1-8), an Abstract Syntax Tree (AST) representation of the code is first generated. The algorithm starts at the root node of the AST tree (line 3) and recursively iterates over the child nodes. Note that as a convention, throughout this algorithm, lines that start with the symbol < and end with > represent code that is *injected* in the form of AST nodes *at that particular location in the AST tree processing*, to augment the input code.

The general idea of the algorithm is that for each function, a **Scope** object is instantiated *in the output code*, and linked to the parent scope, so that an exposed scope tree can be built dynamically *at the time of execution*. Upon the algorithm starting, a **Scope** referring to the global scope is generated (line 5), without any parent scope. Processing then starts from that root node (first invocation of **instrumentNode** at line 9). Then, for each child node in the AST tree, if the child node is a *function*, we generate a new **Scope** linking back to the parent scope (line 13), and we recursively invoke **instrumentNode** again for that node. We also *register* the function in its parent scope, which will allow us to dynamically retrieve it at the

```

1  var global = new Scope("global");
2  function CreateCounters(n) {
3    var createcounters = new Scope(global, "CreateCounters");
4    var total = 0;
5    createcounters.addVar("total", total);
6    function Counter() {
7      counter. = new Scope(createcounters, "Counter");
8      var value = 0;
9      counter.addVar("value", value);
10
11     var anon1 = function() {
12       anon1 = new Scope(createcounters, "anon1");
13       value += 1;
14       anon1.setVar("value", value);
15       total += 1;
16       anon1.setVar("total", value);
17
18       // Can access parent local variables
19       console.log("val=" + val + ", value=" + value + " total=" + total);
20
21       return value;
22     }
23     counter.addFunction("anon1", anon1);
24     return anon1;
25   };
26   createcounters.addFunction("Counter", Counter);
27
28   var counters = [];
29   CreateCounters.addVar("counters", counters);
30   for (var i=0; i<n; i++) {
31     counters.push( Counter() );
32     CreateCounters.setVar("counters", counters);
33   }
34   return counters;
35 }
36 global.addFunction("CreateCounters", CreateCounters);
37
38 var counters = CreateCounters(2);
39 CreateCounters.setVar("counters", counters);
40 ThingsMigrate.setInterval(function() { counters[0] }, 1000);
41 ThingsMigrate.setInterval(function() { counters[1] }, 500);

```

■ **Figure 4** Counters JavaScript Example - Instrumented.

serialization phase (Section 4.6), as otherwise, there would be no way to dynamically access it (as the JavaScript reflection API does not allow access to functions, variables and scopes).

For similar reasons, if the current node being parsed corresponds to the declaration of a new variable, the corresponding variable must be added to the node's current scope (line 19). For an operation that would set the contents of a variable (assignment, incrementation, etc.), we must also refresh the corresponding variable in the scope in which it was declared, to make sure that its content is mirrored in the tree (lines 22-23). Note that this operation first requires finding the scope in which the variable was declared, due to the JavaScript execution model in which a variable defined in any parent scope can be accessed by any child scope (e.g., variable `total` in Figure 2). To that end, the `findScope` function (line 22) walks the tree upwards until it encounters the *most recent* declaration of the variable (up to the global scope). The value of the variable is then updated in that scope.

Figure 4 shows a simplified instrumented version of the original source code shown in Figure 2 (note that in our implementation, many more details are included, which are omitted here for the sake of brevity). The lines of code that are added by the code instrumentation process are shown in grey. As can be observed, all defined scopes (the global scope, then the scopes corresponding to each function definition) are mirrored through an instance of a `ThingsMigrate Scope` object (lines 1, 3, 7 and 12). In addition, each variable definition or assignment gets mirrored in the tree, in the scope at which it is defined (lines 5, 9, 14, 16, 29, 32 and 39). Similarly, functions are also registered (lines 23, 26 and 36).

Instrumenting Timers. Following a similar algorithmic approach as in Lo et. al. [32], ThingsMigrate provides support for saving the state of timer functions, namely `setInterval` and `setTimeout` (challenge 2). This is accomplished in the instrumentation phase by replacing standard timer calls by invocations of our own functions, which expose the state of the timers at serialization time. Thus, at restoration time, the timers resume at the state when serialization took place. For instance, considering our code example, if snapshotting occurs after 250ms, then the first timer (line 23) will first trigger after 750ms, then every second, while the second timer (line 24) will first trigger after 250ms, then every 500ms.

Pub/Sub Interfaces. ThingsMigrate provides support for capturing the state of pub/sub interfaces (challenge 2). Similar to how we handle timers, ThingsMigrate wraps calls to the pub/sub interface (MQTT library) at the instrumentation phase, so that upon a migration being requested, the *list of each topic previously subscribed* by the application gets serialized as part of the snapshot. Then, at the restoration phase, prior to resuming the execution, a subscription is transparently reestablished to each of the previously subscribed topics. To ensure that no publications are lost *during the migration*, we assume that reliable pub/sub is provided by the service [44, 20], so that the latter can retransmit any missed publication sent during the migration.

In addition, as the migration is triggered by a pub/sub publication, the *Instrumentor Service* injects code in the header to setup a pub/sub listener for the migration, when the instrumented program is executed. Upon the specific publication arriving, the framework starts the state serialization process.

Classes and Prototypes. JavaScript ES5 does not support classes *per se* unlike object-oriented languages (e.g., Java). Instead, it provides high-level abstractions that emulate classes by means of *prototypal inheritance* [23]. ThingsMigrate provides support for serializing JavaScript-like ES5 classes by serializing each object's *prototype object*, so that upon restoring the code, the correct prototypal chain can be recreated along with the objects.

Cleaning Orphaned Scopes. During the life cycle of a JavaScript application, scopes are dynamically created, and can sometimes become *orphaned*. Orphaned scopes are scopes for which there are no single remaining reference to them or to one of their child scopes. In the example shown in Figure 2, at each timer iteration (lines 23-24), the function scope that is created on the fly (first argument) becomes orphaned and is therefore destroyed, as its serialization will not be required. Therefore, we need to destroy the scope objects corresponding to orphaned scopes, as they can lead to memory size increase – this problem is exacerbated on multiple migrations (challenge 5).

As a novel contribution, ThingsMigrate provides support for automatically destroying orphaned scopes, to support multiple migrations (challenge 5) on the same application without increasing the snapshot size and incurring additional overhead in the restored code (i.e., scope explosion). In the instrumentation phase, prior to any given function ending or returning, an API call to `scope.checkAndDestroy()` (line 16 of Algorithm 1) is *injected*, for the current `scope` object. At execution time, this function will check whether any other scope or variable depend on this scope. If there are no dependencies, then the scope is destroyed, and therefore it will not be serialized in the snapshotting phase (Section 4.6).

4.6 Phase 2: Snapshotting and Migrating

To trigger a migration, the component that is being executed receives a *migrate* pub/sub command from the *Migrator Service*. Recall that the code instrumentation phase sets up a listener, which initiates the migration (Section 4.5).

Serializing the State. The migration process first involves serializing the state to the JSON format. To do so, the scope tree is recursively walked in a top-down approach, from the global scope. The serialized output includes, for each scope, the variables and parameters, as well as nested scopes and functions. In JavaScript, functions cannot be serialized as-is. Thus, upon encountering a function when walking the scope tree, the function is assigned a unique ID, and the function's source code is added to a table of functions, which is appended at the end of the serialized state. Note that the serialized output also contains the state for special objects that ThingsMigrate addresses, such as timers and pub/sub interfaces.

Handling the Stack. We address the challenge of handling the stack (challenge 3) by exploiting the asynchronous, event-driven nature of JavaScript. Because JavaScript applications are single-threaded and are event-based, the runtime maintains an event queue. We *schedule* code migrations as events so that they get pushed at the end of the event queue and get executed over an *empty* stack. More precisely, as migration requests are sent through the form of pub/sub publications, they are treated as events and pushed to the event queue. Note that we could also accomplish the same behavior by scheduling the migration as a timer-based event.

Sending the Serialized State. Once the snapshot is generated, it is sent over the pub/sub interface to the target IoT node, which will regenerate the code considering the state of the snapshot, and resume execution.

4.7 Phase 3: Code Restoration

Upon a given IoT node receiving a snapshot, it needs to reconstruct the original program *at the exact state* where migration took place (challenge 4). The code restoration process must retain the original program structure, while reassigning the values for constructs holding state, such as variables, parameters and closures, without directly restoring the memory regions - this is important for platform independence and portability.

Reconstructing Closures and Scopes. As in the code instrumentation phase, closures pose unique challenges when it comes to generating restoration code, as they wrap state elements. Because functions can be return values of functions in JavaScript (e.g., as seen in Figure 2), there can exist multiple *copies* of a function sharing the same code, but corresponding to different *states* (i.e., holding different values). The code restoration process needs to generate multiple copies of some of the function trees, as state can be held not only in the functions themselves, but anywhere in parent functions as well, and bind such copies; i.e., to variables or parameters. For instance, in Figure 3, two counters are defined (i.e., `counter[0]` and `counter[1]`), which both point to a function having the same source code (i.e., the anonymous function at lines 13), but holding different states, as the value of `value` defined in the parent function (`Counter`) is different. Thus, in the reconstructed code, two copies of `Counter` and its inner function (i.e., the *chain of functions*) will need to be defined to expose the different scopes of the two counter closures.

Algorithm 2: Code Generation.

```

1 function generateScope (scope, parentScope)
2 begin
3   < (function(){ >
4   <var {scope.name} ← Scope(scope.name, parentScope)>
5   foreach param in scope.params do
6     | <var {param.name} ← param.value>
7   end
8   foreach function in scope.functions do
9     | <functionTables[scope].code>
10  end
11  foreach variable in scope.variables do
12    | if scopeDefinitionExists(variable.value) then
13      | <var {variable.name} ← variable.value>
14    end
15    else
16      | stage2Variables.add(variable)
17    end
18  end
19  foreach child in scope.children do
20    | generateCode(child, scope);
21  end
22  foreach variable in stage2Variables do
23    | <var {variable.name} ← variable.value>
24  end
25  < }() >
26 end

```

Code Generation Algorithm. A simplified version of the code generation algorithm is shown in Algorithm 2. In a nutshell, the algorithm starts with the global scope (function), and recursively reconstructs the scopes in a hierarchical manner. For a given scope, it first injects the parameters defined in that scope with their values at snapshot time (lines 5-7), then injects the full source code for the functions defined in that scope, *including the function headers* (lines 8-10). Then, the variables defined or redefined in that scope are injected and set to their value at snapshot time (lines 11-18). In some corner cases involving JavaScript objects and their prototypes, it might happen that some scopes cannot be resolved for some of the variables, at the first (i.e., top-down) phase. An example would be a case where an object instance is constructed, but the constructor function is defined in a child scope that is not yet generated (i.e., invoked) at the time of assigning the variable. Our approach addresses these situations by placing such variables in a queue, to process them in a later stage (i.e., at stage 2, after the generation of the child scopes - lines 22-24). After generating the variables, the `generateScope` function is recursively called for all child scopes of the current scope (lines 19-21).

Each scope definition is *wrapped* in an enclosed `(function() {...} ())` call, which means the scope definition code (i.e., the output of the algorithm) will be *invoked* when the restored code is executed (lines 3, 25). In other words, the nested scope generation portions of code will be invoked recursively, thereby recreating the scope hierarchy. Note that the *functions* themselves corresponding to each scope are not executed upon restoration, as this could lead to side effects (i.e., non-determinism). It is nevertheless necessary to include their definitions, as they might be invoked later in the code *after restoration*.

Code Restoration Example. Assume that a snapshot was taken after executing the code shown in Figure 2 for 3.25 seconds. Figure 5 illustrates the restored code. Note that while this example has been derived from the output of a real invocation of the code restoration

```

1  /* Original code comes before */
2  function() {
3    function CreateCounters(n) {
4      var n = 2;
5      function Counter_1() {
6        var anon1 = function() { /* ... */ }
7        ThingsMigrate.addFunction("Global/CreateCounters/Counter_1/anon1",
8          anon1);
9        var value = 3;
10       return anon1;
11     }();
12     function Counter_2() {
13       var anon1 = function() { /* ... */ }
14       ThingsMigrate.addFunction("Global/CreateCounters/Counter_2/anon1",
15         anon1);
16       var value = 6;
17       return anon1;
18     }();
19     var total = 9;
20     var counters = [Counter_1, Counter_2];
21   }(2);
22 }();
23
24 ThingsMigrate.setInterval(ThingsMigrate.findFunction("Global/CreateCounters/
25 Counter_1/anon1", 1000, 250);
  ThingsMigrate.setInterval(ThingsMigrate.findFunction("Global/CreateCounters/
  Counter_2/anon1", 500, 250);

```

■ **Figure 5** Counters JavaScript Example - Restored Code.

procedure of ThingsMigrate, some simplifications and adjustments were made for clarity. Also, the names of the various entities within this snippet (i.e., variables, functions, scopes), as well as their relationships, correspond to the state example shown in Section 4.4.

As can be observed, two copies of the Counter closures have been generated: Counter_1 and Counter_2 (lines 5-10 and 12-17), which both wrap the values for variable value: 3 and 6 (as the timers triggering the closure incrementation functions were invoked respectively 3 and 6 times - the former every second, and the latter every 500 ms). Similarly, there is only one instance of the CreateCounters closure, which is the parent of the two Counter functions (line 3). It holds the total variable (total = 3 + 6, line 19). Upon executing the restored code, the CreateCounters, Counter_1 and Counter_2 functions are re-executed, thereby recreating the closures as before.

Note that upon restoring a function, we add it to a function table, which will allow us to refer to it later (not shown in Algorithm 2). As an example, at lines 7 and 14, we add the restored closures that correspond to the anonymous counter incrementation functions to the function table, and we then retrieve them from the table when we restore the timers (i.e., lines 24-25), as the timers periodically invoke these functions.

Multiple Migrations. ThingsMigrate supports transparent multiple migrations without introducing additional overhead (challenge 5). This is accomplished at the code restoration phase by maintaining a unique scope tree structure that is accessed by all the generated closures and scopes, and by re-injecting scope definitions (i.e., variables, parameters, nested functions, etc.) across the regenerated code, following an approach derived from Algorithm 1. Further, relevant pub/sub code is re-injected to support receiving *migrate* messages again. In other words, the output of the code restoration phase is *an alternate code segment* equivalent to the output of the code instrumentation phase, which can hence support further migrations.

4.8 Limitations

Handling External Libraries. ThingsMigrate does not yet provide full support for imported libraries (i.e., the `require` statement). A simple solution would be to directly import the code in the main JavaScript module itself prior to instrumentation. This approach may be inefficient however, if there are multiple levels of nested library imports. Another solution would be for ThingsMigrate to provide a migration interface, and for module developers to implement the interface for either a more optimized migration of the nested libraries, or for supporting libraries exposing native I/O resources, such as file system access. Despite this limitation, we find that ThingsMigrate can support many third-party libraries as we show in Section 7.

Scope Explosion. If programs make use of several levels of nested closures, then the resulting snapshot and restored code can become quite large, due to the phenomenon of *scope explosion*, in which multiple scopes might have to be maintained. However, this problem is symptomatic of bad programming practices and is not specific to ThingsMigrate, as the JavaScript VM itself will have to retain a large amount of scope structures in-memory.

Redirecting I/O Operations. As mentioned in Section 4.1, ThingsMigrate assumes that all communications are done over the pub/sub interface. Further, in the current state, ThingsMigrate does not support file I/O operations, which is non-trivial, as reads and writes must occur where the corresponding files are located. For instance, assume there is a file on device *A* which is read by an application on the same device that gets migrated to device *B*. In order to guarantee consistent reads, one has to migrate not only the current position in the file, which is trivial, but also to guarantee (1) the availability of the file on *B*, or (2) to provide some redirection mechanism.

As JavaScript I/O operations are typically handled through streams, we plan on transparently redirecting streams over the pub/sub interface (solution 2 above), by wrapping the base JavaScript stream API (similar to wrapping timer-based or pub/sub-based APIs). A stream-level solution can support arbitrary stream-based I/O operations, such as files, network, and even HTTP requests. Upon device *A* receiving a migration request to migrate a given app to device *B*, the ThingsMigrate Runtime will generate a unique ID for each currently active stream, and will setup a transparent forwarding mechanism over a pub/sub bridge (i.e., by creating a topic corresponding to that ID that both devices *A* and *B* will subscribe to). Then, upon a read operation being requested by the app on device *B*, for a given stream, the request will be transparently forwarded by the Runtime to device *A*, who will perform the read and send back the results to the Runtime on *B*, who will deliver them to the stream at the application layer. Likewise, any write operation will simply be forwarded from the Runtime on *B* to the Runtime on *A*, who will complete the write.

Nested Timers. A limitation of ThingsMigrate occurs in the handling of some deeply nested timer-related calls (i.e., `setTimeout`, `setImmediate`). Should a *snapshot* command be received while a *timer* is in a *pending* state – i.e., before the callback function is invoked – then the timer gets cleared, the remaining time and the reference to the callback function are serialized, and migration happens normally. However, should the snapshot command be received *after* the callback function is invoked, then a race condition occurs between any asynchronous calls made inside the body of the callback and the snapshot function. Race conditions are sometimes problematic in JavaScript, as the ordering of events can't always be predicted [16, 33]. For instance, should the JavaScript VM event loop process the snapshot

function before the asynchronous calls, then the resulting snapshot will not contain the scopes created by the asynchronous calls, producing an incorrect snapshot. Handling nested timers would require that the snapshot function be delayed until all callbacks have been resolved, which is a non-trivial problem. As a potential solution, we propose to inject, at the instrumentation phase, specific code into the function scope that will signal the function's completion, which would allow us to detect the resolution of nested asynchronous calls.

5 Implementation

ThingsMigrate is implemented in the form of a JavaScript library² that can be included by the application. Its implementation is built over ThingsJS [26] (more details in appendix A). It provides APIs that can be invoked to perform code instrumentation, snapshotting and code restoration. From a higher-level perspective, it also provides an execution environment that replicates the architecture shown in Figure 1. More specifically, it provides a Runtime environment that can be run on IoT devices supporting an appropriate VM (e.g., Node.js on Raspberry Pi Models 3 and 0), as well as a *Manager* component, which is used to transparently instrument JavaScript programs, launch them on specific IoT nodes (decided by a scheduler), monitor them, and trigger a serialization/migration. Internally, our implementation uses the popular *esprima* library [7] to parse JavaScript code into an AST, and the *escodegen* [6] to convert back an AST into JavaScript code.

We also provide a web dashboard to monitor the execution of the application on different devices, and trigger the migration at runtime (more details in appendix B).

6 Experimental Validation

We perform three experiments to validate ThingsMigrate. Experiment 1 (Section 6.2) benchmarks the performance of our code instrumentation algorithm against a set of benchmarks. Experiment 2 (Section 6.3) measures the performance overhead of ThingsMigrate for benchmarks running on different devices. Finally, Experiment 3 (Section 6.4) evaluates the multi-migration capabilities of ThingsMigrate by migrating a benchmark application several times, across several devices.

6.1 Experimental Setup

ThingsMigrate provides JavaScript migration between IoT devices, and between devices and the cloud. To emulate different scenarios, we ran our experiments on two IoT platforms, namely a Raspberry Pi model 3B (quad-core 1.2 Ghz ARM7, 1 GB memory), and a Raspberry Pi model 0W (single-core 1 Ghz ARM6, 512 MB memory), both running the Raspbian Jessie operating system (a Debian Linux variant). We also included a cloud server (Xeon E3-1220 v3, quad-core 3.10Ghz, 32 GB memory). All nodes were running the Node.js VM version 6, which is ES-5 compliant. While we did not test other VMs due to stability issues or to their lack of compliance with the ES-5 standard, the Node.js VMs we used were all compiled differently for each target platform.

Despite an extensive search, we did not find publicly-available sets of IoT-specific JavaScript benchmarks to evaluate our system. Prior work ([39]) has built their own IoT-specific

² <http://www.github.com/karthikp-ubc/thingsjs>

JavaScript benchmarks³. We followed a similar approach and built two IoT-specific benchmarks: (1) a *factorial* application, which computes the factorial of a very large number and uses closures to store the computed digits (i.e., in a very large expanding array), and (2), a *regulator* application, which models an IoT *edge* component which receives temperature measurement data from different sensors⁴ over a pub/sub interface, keeps the previous n values for m sensors, and periodically computes an optimal power adjustment to be sent to an actuator. `factorial` models a CPU and memory-intensive application of a finite duration (experiment 2), while `regulator` models a less intensive (i.e., low CPU and memory usage) application that runs for a long time. We note that the memory usage of the `regulator` is similar to the memory usage of the IoT-specific benchmarks described in [39].

In addition, for experiments 1 and 2, we also used some benchmarks from the Chromium Octane [2] suite, which were originally designed to stress-test the performance of the V8 JavaScript engine in the Chrome web browser. While they are not representative of IoT applications, we nevertheless use them to assess the universality of our framework, and for performance testing of ThingsMigrate under extreme conditions.

6.2 Experiment 1: Code Instrumentation

In this experiment, we consider all the benchmark programs from the Chromium Octane suite that do not depend on a web browser (i.e., accessing the DOM or any other in-browser object), as ThingsMigrate migrates IoT applications rather than in-browser applications. We measure the time it takes to instrument the code for these benchmarks⁵, as well as for our `factorial` and `regulator` applications. In addition, we compare the size of the uninstrumented (raw) code, and the size of the instrumented code. Results are shown in Table 1. As one can observe, the instrumentation algorithm executes quickly (in under 1 second), even for the complex benchmark applications with large code sizes. Further, code instrumentation is a one-time process for any given program.

The increases in the code size due to instrumentation range from 26.9% to 7382.7%, with an average of 1174.1%. This is because the instrumentor assigns human-readable variable and function names in the generated code for debugging purposes - this can be reduced by using a minifier [14]. We do not deploy these techniques. However, the code size has minimal impact on the runtime performance as JavaScript is compiled just-in-time.

6.3 Experiment 2: Performance Overhead

In this experiment, we analyze the performance impact of ThingsMigrate over a set of highly resource-intensive benchmarks. The goal of this experiment is to model the execution of a resource-intensive task of a finite duration (i.e, eventually returns a result) that would be executed over different IoT devices and the cloud server. We selected benchmarks `navier-stokes` and `splay` from the Octane suite, as they respectively model extreme conditions, in terms of CPU usage and memory utilization. Further, we were successful in running these benchmarks on all test devices, unlike most other benchmarks in the suite (even without our instrumentation, most of the benchmarks in the Octane suite were unable to run on the Raspberry Pi 0 due to its limited capabilities). We also used our `factorial` application.

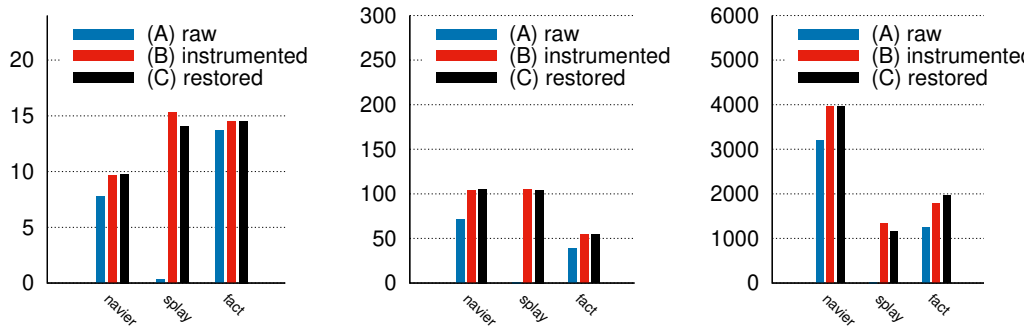
³ The source code is not publicly available, and hence we cannot use them.

⁴ We fed the application with random values, as the computed result itself is not part of the experiment.

⁵ Measurements were taken on our cloud server.

■ **Table 1** Code Instrumentation Results (with a confidence interval of 95%).

Benchmark	Program Size (kb)	Instrumented Size (kb)	Instr. Time (ms)
navier-stokes	9.985	122.263	135.67 ± 4.5
splay	6.573	45.984	86.18 ± 2.7
deltablue	1.5452	115.623	120.65 ± 0.6
crypto	39.028	276.763	194.05 ± 1.6
box2d	357.169	2773.027	821.95 ± 3.0
earley-boyer	159.794	574.463	301.9 ± 0.8
raytrace	24.998	31.720	64.2 ± 0.5
richards	8.302	59.922	87.4 ± 0.5
typescript	2.138	10.541	31.75 ± 0.2
factorial	0.952	5.526	28.21 ± 0.2
regulator	1.855	15.594	42.1 ± 0.4



(a) Cloud Server (Xeon)

(b) Raspberry Pi 3

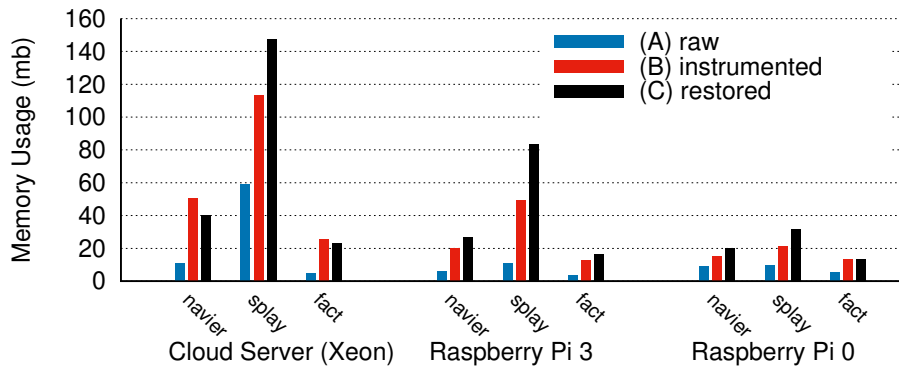
(c) Raspberry Pi 0

■ **Figure 6** Execution Time (in seconds). Margins of errors were below 1.5% for most of our results, and up to 6% for some of our results on the Pi 0, for a confidence interval of 95%.

For each benchmark program, we measure and compare the time to complete its execution. For each benchmark, and for our 3 target devices (Raspberry Pi 3, Pi 0 and our cloud server), we run (A) the non-instrumented (raw) code, (B) the instrumented code, and (C) the code generated after migration⁶. Further, we measure the average memory usage of each application for the same three versions of the benchmark (raw, instrumented and generated) to determine memory overheads. Finally, we report the time taken to serialize and generate restoration code. We ran each benchmark until 50% of its execution time, took a snapshot, generated restoration code from the snapshot, and then resumed the execution with the restored code. Each benchmark was executed multiple times on each platform, and the times averaged over the executions were reported.

Execution Time. Execution time results are shown in Figure 6. For both `navier-stokes` and `factorial`, we observe an execution time overhead ranging from 5% and up to 40% compared with the raw code (A), for all devices, which is due to the overhead of our injected instrumentation code. This is because these applications have a significant amount of state.

⁶ As results for the restored code (C) were only available after completing a migration (i.e., at a given time t during execution), results for (A) and (B) were considered also only after time t in their respective runs, for a fair comparison.



■ **Figure 7** Memory Usage (mb). Margins of errors are not shown, as the results show the averaged memory usage for all runs, averaged over the duration of the experiment.

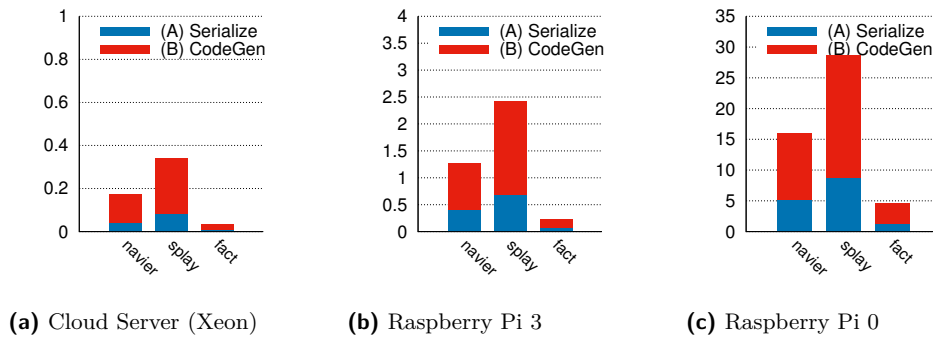
As for the `splay` benchmark, the performance of the instrumented (B) and the restored code (C) was significantly degraded. This is due to the extreme amount of memory operations that the benchmark performs, which significantly slows down the execution. This slowdown is amplified by the mirroring of the scope tree, which consumes even more memory. Further, as our *Pi* devices have much slower memory, compared to our cloud server, the performance overhead is higher. We stress however that these benchmarks were specifically designed to model *extreme* conditions on desktop computers, and are not typical applications to be run on IoT end nodes, which are much more resource constrained.

We also observe that the performance of the instrumented code (B) and the restored code (C) is roughly similar across all benchmarks. As the restored code is *semantically equivalent* to the original code, but with instrumentation to enable further migrations, we obtain similar performance as the instrumented *non-migrated* code. These results indicate that the performance (i.e., execution time) *will not* degrade after migration (Section 6.4).

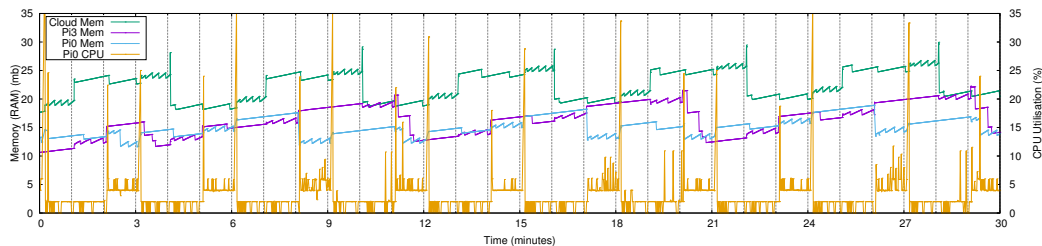
Unfortunately, we cannot perform direct comparisons with prior work in terms of execution time overhead, as Lo et. al. [32] measured such overheads for web applications on desktop computers, which do not exhibit the same workload characteristics as our benchmarks, and Kwon et. al. [29] did not report the execution time overheads of their programs at all.

Memory Usage. Our memory overhead results are depicted in Figure 7. For each benchmark and device, we averaged the memory usage over time across the duration of each execution, and we report averaged results for all experimental runs. Overall, our results reveal that executing the instrumented code (B) significantly increases the memory usage compared to the non-instrumented code (up to 6 times). This is expected, as we do not rely on JavaScript VM instrumentation at runtime (unlike Kwon et. al. [29]), therefore many more elements of state must be captured during execution and mirrored. In addition to maintaining the scope hierarchy that mirrors the closures, the instrumented code also maintains a copy of every variable, parameter and function, which increases the memory usage. The results for `factorial` exhibit a similar trend across all devices, with the restored code (C) having a slightly lower memory footprint compared to the instrumented code (B). After restoration (C), we start with a fresh *instrumented* copy of the code (at snapshot time), without the *past* states that potentially contain portions that were not yet garbage collected.

On the other end, `navier-stokes` and `splay` exhibit much higher memory usage for the restored code (C) compared to the instrumented code (B). As these benchmarks are more aggressive in stressing the memory (i.e., by allocating and deallocating scopes), the resulting reconstruction code is very verbose (i.e., due to the explicit definition/duplication



■ **Figure 8** Serialization / Code Generation Time (in seconds). Margins of errors were between 0.5% and 5% for all results, for a confidence interval of 95%.



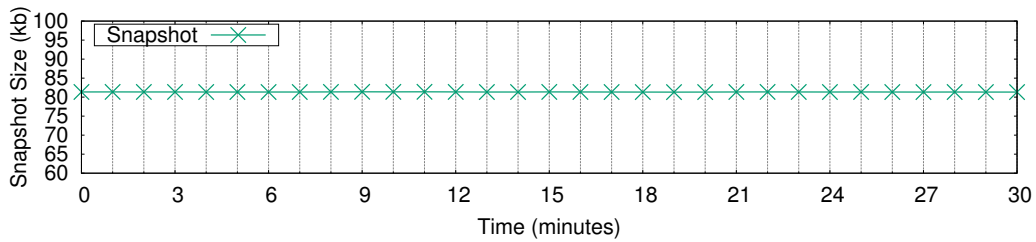
■ **Figure 9** Multi-Hop Migration Analysis (*regulator* application).

of nested closures as shown in Figure 5). Therefore, despite being *semantically* equivalent as the instrumented code at snapshot time, the reconstructed code may be harder to optimize by the VM. The execution of the JavaScript Garbage Collector (GC) can also be a cause of the overhead. Experiment 6.4 discusses the effects of the GC on the live execution of JavaScript applications. We stress again that such benchmarks represent extreme, non-typical conditions. Nevertheless, they could run even on low-end devices (e.g., the Raspberry Pi 0), and the average memory footprint remained under 30 MB.

We also note that the memory usage for both Raspberry Pi devices are much lower than the cloud server. This is attributable to the *bitness* of the devices; i.e., our cloud server has a 64 bit processor, while the other Pi devices are 32 bits, and a more aggressive GC execution on the Pi devices, as they are more memory-constrained.

Finally, prior work (i.e., [32, 29]) did not evaluate the runtime memory overhead of their approach and hence, we cannot compare our results against them.

Serialization and Code Reconstruction Time. Considering the same experimental setup (i.e., same devices and same benchmarks as above), we measured the time it took to serialize the state at mid-experiment, and to generate restoration code from the state. Results are shown in Figure 8. The results exhibit the same trend across all platforms, and vary based on the size and complexity of the application. Overall, the serialization and snapshot times are reasonable, albeit slightly higher on the Raspberry Pi 0 and for the two Octane benchmarks. We stress that the Raspberry Pi 3 device is roughly 7 times slower than our cloud device, while the Pi 0 device is roughly 90 times slower than our cloud server.



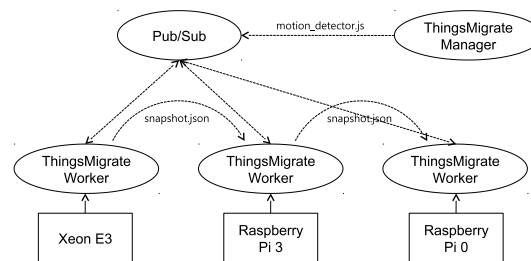
■ **Figure 10** Snapshot Size over Time (*regulator* application).

6.4 Experiment 3: Multiple Migrations

In this experiment, we analyze the global behavior and performance over time of ThingsMigrate, when multiple migrations are performed between the *edge* and the *cloud*. More precisely, we analyze the effects of migrating a long-running task that is not computationally expensive from one device to another. None of the benchmarks used in Experiment 2 fit this description, nor could we find publicly available JavaScript-based IoT benchmarks that satisfy this criteria (Section 6.1). Therefore, we developed and used our own benchmark – the *regulator* application that satisfies this criteria (by design). We first deploy the regulator application on our cloud server, then we migrate it to the *edge* devices (i.e., the Raspberry Pi 3 device, after one minute, and then to the Pi 0 device, after one minute). The application is then pushed back to the cloud server. This cycle is repeated 10 times (30 migrations over 30 minutes), and the CPU and memory utilization are measured in each instance.

The memory utilization results are shown in Figure 9, for the duration of the experiment (30 min). The migration cycles are denoted by a vertical bar (every minute), and an oscillating variation pattern can be observed during the time periods for which each device was executing the regulator application. As can be observed, the memory usage fluctuates, for all devices, but remains overall stable, as each successive code restoration does not consume additional memory (assuming the memory needs of the application do not increase). The step-like appearance of the memory curves are explained by the JavaScript garbage collector (GC), which regularly claims small amounts of memory (i.e., during execution of the *regulator* – small pikes), and which periodically runs a more thorough collection (bigger drops). However, we also observe that the memory tends to very slowly increase over time, but this is not due to the multiple migrations – rather, this is an artifact of the experimental data collection process, which logs memory and CPU usage at a frequent interval (every 200ms) and keeps the data in memory. This is supported by Figure 10, which plots the snapshot size at each successive migration, which remains constant at 83kb. Finally, as in Experiment 2 (Section 6.3), the memory usage on the cloud server is higher than on the pi devices.

The CPU usage is shown on the same Figure (9). For simplicity, we show CPU usage results only for one device (i.e., Pi 0, which is the most resource constrained), but the trend is similar on the others. As can be observed, the CPU usage peaks at about 4%-5% when the Pi 0 device is executing the application, and is close to 0% otherwise. The CPU usage during execution remains constant across the different executions. The short spike *before* execution corresponds to the code reconstruction, and the short spike *after* execution corresponds to the serialization process, for which a small memory surge can also be observed.



■ **Figure 11** Case study setup.

6.5 Summary

Overall, our results demonstrate that ThingsMigrate can enable the cross-platform migration of IoT JavaScript-based applications with acceptable performance overhead ($\sim 30\%$ for normal cases), and without any modifications to the underlying VM. While the memory overheads are more significant, we believe that this is an acceptable tradeoff given the goal of our approach to rely purely on code instrumentation. We also believe that memory gains could be achieved by optimization techniques such as storing references to variables rather than copying them within the scope tree. However, this is a subject for future exploration. Further, our results show that ThingsMigrate was able to handle multiple-hops migrations while keeping the CPU and memory usage almost constant.

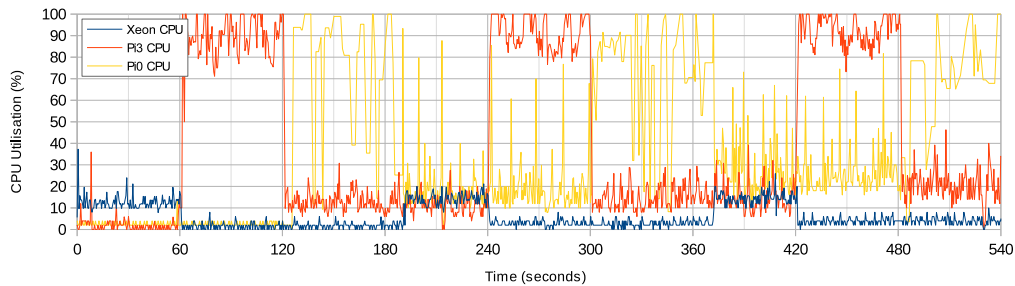
7 Case Study: Motion Detector

In this section, we describe our experience with using ThingsMigrate to build a realistic IoT application for video surveillance by adapting third-party JavaScript components developed for standalone node.js applications. These components were not designed with ThingsMigrate in mind, and as a result, we had to make (minor) modifications to make them work with our system. We also evaluate this application using application-specific metrics that are more likely to be of interest to end users rather than CPU/memory usage (unlike Section 6).

7.1 Experimental Setup

We set up an IoT network with four devices to build a surveillance system. Figure 11 shows the setup. The application logic is modularized into two components: a video streamer component that captures images from a video source such as a webcam, and a motion detector component that processes the images to detect motion. Unlike the video streamer, which is bound to a single device by the peripheral from which it needs to capture video, the motion detector can be run on any device as it performs computations on the image data. We measured the behavior of the system over a series of migrations of the motion detector across the three systems (Raspberry Pi 3, Pi 0, and the cloud server from Section 6).

Video-streamer: We used FFmpeg [8], a popular open-source software for handling multimedia, to capture individual frames from a video stream. For the purpose of the experiment, the component was configured to stream from a video file instead of a peripheral such as a webcam, so that we have a deterministic and reproducible sequence of frames. To interface with the FFmpeg process from the JavaScript layer, we adapted a third-party NPM library called `fluent-ffmpeg` [9], that we use to capture individual frames and publish them over the pub/sub interface. The capture-and-publish routine was written as a single JavaScript



■ **Figure 12** CPU Usage over time – Motion Detector component.

function `captureFrame` that was passed into a `setInterval` call with interval set to 200ms (i.e., a rate of 5 frames per second). We used the cloud server to serve as a surveillance camera and run the video streamer component.

Motion Detector: This component was written entirely in JavaScript without having to interface with any external software. We integrated a third-party NPM module called `jump` [11], which provides an API to read `Buffer` objects (i.e., received from the pub/sub overlay) and perform image processing tasks. The component stores binary frame data for the n latest frames.

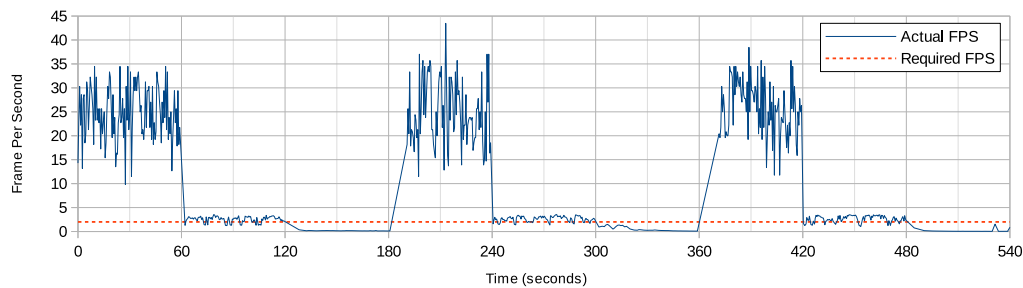
The motion detection logic (i.e., function `detectMotion`) iterates through the array of images and computes the difference between subsequent frames by calling `jump.diff()`. The binary difference between the frames is published over the pub/sub interface. In addition, if more than 10% of the pixels are altered, a motion *detected message* is also published. The `detectMotion` function is passed to a `setInterval` call with the interval set to 500ms - this is lower than the frame rate of the video streamer (Section 7.2 explains why). Since the `detectMotion` works by retrospective inspection of past frames, the array of `Buffer` objects containing the image data needs to be migrated. Otherwise, the restored component would need to wait for the buffer of past frames to be filled again – thereby skipping the motion detection process for a given time window, and missing potentially important motion.

Although we do not fully support the migration of external libraries (Section 4.8), it was possible to integrate the third-party NPM libraries as the objects they created were native JavaScript objects and the API calls were limited to stateless operations. For instance, since the `Buffer` objects are native objects, they could be easily serialized and migrated. The function call to `jump.diff()` is a stateless operation, since it does not create any additional scopes and its execution context is destroyed after it returns. Such stateless operations do not affect the migration process because we do not need to serialize their scopes.

Finally, we collected performance statistics by subscribing to a pub/sub topic at which each ThingsMigrate runtime publishes its CPU and memory usage. To monitor and verify that the motion detection was working correctly, we used our web dashboard, which displays the images by converting the data into a base64 encoded PNG image.

7.2 Results

To automate our migration test in a controlled fashion, we wrote a Node.js script to send commands to the IoT devices over the pub/sub interface. We sent a *migrate* command every 1 minute to the Cloud Server, Pi 3, and Pi 0, and back. We repeated the cycle 3 times.



■ **Figure 13** FPS over time – Motion Detector component.

Figure 12 shows the CPU usage over time as the application is migrated between the devices. The collected data for CPU and memory usage across the three devices exhibit a similar pattern to the regulator component discussed in Section 6.4. The CPU usage on a device has a spike upon receiving a snapshot and just before sending a snapshot, remains high while it is running a component, and stays near 0 during the idle state. The memory consumption stays within a narrow range, with the garbage collector being triggered more frequently while a device is running a component, and occasionally while it is idle.

However, on the Raspberry Pi 0’s console, we observed error messages showing that the process failed at regular intervals. This is because the asynchronous call to *detectMotion* took much longer than the set interval of 500ms, due to the limited computational capacity of the Pi0, which led to the event queue accumulating faster than the JavaScript VM could consume, which eventually led to overflowing and halting of the program.

Figure 13 shows the frame rate measured in Frames Per Second (FPS) on each device over time. The FPS was calculated using the formula $\frac{1}{\Delta t}$ where Δt is the time taken to execute the *detectMotion* function. The figure shows the FPS dropping below the required FPS of 2 over the periods between 120 and 180, 300 and 360, and 480 and 540 seconds, during which the Pi 0 was running the motion detector component. We can also observe the *detectMotion* function blocking the thread at 180 seconds and 360 seconds, preventing the migration from being triggered. The FPS drops below 2 occasionally during Pi 3’s execution, but for most frames it is able to process within the time interval, compensating for the delay overall.

In summary, this case study shows that we were able to successfully migrate a third-party application with minimal modifications between different devices. We also found the frame rate measured in FPS was acceptable in most cases for the application. However, special considerations might have to be taken for low-end devices, as migration requests can be delayed due to delays in running more intensive tasks. These issues should be considered when designing a system-level solution as discussed in Section 3.1.

8 Related Work

There has been some prior work in the area of migrating the execution of JavaScript code. However, they focus on migrating web applications between web browsers [18, 32, 36, 29], and hence have very different constraints from IoT devices. Imagen [32] migrates web applications across heterogeneous browsers without altering the VM, and address some of the challenges specific to web applications (e.g., the DOM, HTML5 media elements, timers). However, their handling of nested closures is limited (Section 4.5). In [29], extending their prior work in [36], the authors provide deeper support for serializing and reconstructing closures for migrating

web applications, but they require VM instrumentation to access the internal scope tree, which makes their approach less portable as it is bound to a specific browser version of an open-source Webkit browser. In contrast, our approach does not require any modifications to the JavaScript VM, and is hence platform neutral. Further, we provide explicit support for multiple migrations, which prior work does not.

As an alternative to capturing and restoring the state of the web application, deterministic replay techniques can be used to replay an exact sequence of actions leading to the current state [17, 34, 37, 19, 15]. However, these approaches focus on capturing and replaying web browser events, and are not directly applicable to IoT environments. Further, they may even be impractical for IoT environments which have limited resources, as the sequence of events to be captured and replayed often grows rapidly over time [32].

There have been many attempts at providing low-level code migration techniques that directly save and restore the process memory space, and are hence programming language independent [35, 45, 46]. Such techniques could be applied for migrating JavaScript code, but they would require serializing the state of the JavaScript virtual machine (VM) itself, which can incur significant overheads on IoT devices. Further, considering as per our model that one VM might host several components, this would make it difficult to separate the per-component state. Finally, providing platform independent migration would not be possible, as even the same version of a given VM might have different cross-platform implementations and memory layouts due to hardware differences. Note that similar challenges can be found in migrating virtualized OSes across devices [22, 43].

9 Conclusion and Future Work

In this paper, we presented ThingsMigrate, a middleware layer that provides VM-independent migration of stateful JavaScript applications across IoT devices. ThingsMigrate uses code instrumentation to expose the hidden states of a JavaScript application, thereby allowing its state to be captured and serialized, without requiring VM instrumentation. ThingsMigrate then generates a reconstructed version of the same application *at the serialized state*, allowing its execution to continue on a different device. We built an implementation of ThingsMigrate and evaluated it on three different devices, and against both standard benchmarks and custom applications. Our results show that ThingsMigrate can instrument, serialize and reconstruct JavaScript applications within reasonable time bounds, depending on the state and complexity of the input application. Further, we find that ThingsMigrate imposes an average 30% execution time overlay at runtime, which is reasonable given the non-reliance on VM-dependant low-level techniques (i.e., VM instrumentation). Finally, we show that ThingsMigrate supports multiple migrations across different devices without incurring additional overheads.

As future work, we intend on improving support for more complex cases of classes and prototypes, as well as supporting the features of the newer ECMA standards. We would also like to accomplish migration without interrupting the execution flow (i.e., seamless migration). Another interesting area would be to adapt our approach to provide fault tolerance in an IoT setting. While this could be provided by periodically saving the state to a reliable entity, we would like to explore the problem of *dynamically* serializing the state to persistent storage *during* execution.

References

- 1 Babel.js JavaScript Compiler, 2017. URL: <https://babeljs.io/>.
- 2 Chromium octane benchmark suite, 2017. URL: <https://github.com/chromium/octane>.
- 3 DukTape, 2017. URL: <http://www.duktape.org/>.
- 4 ECMAScript 2015 Language Specification, 2017. URL: <http://www.ecma-international.org/ecma-262/6.0/>.
- 5 ECMAScript 5.1 Language Specification, 2017. URL: <http://www.ecma-international.org/ecma-262/5.1/>.
- 6 escodegen: ECMAScript code generator, 2017. URL: <https://github.com/estools/escodegen>.
- 7 esprima: ECMAScript parsing infrastructure for multipurpose analysis, 2017. URL: <http://esprima.org/>.
- 8 FFMpeg Website, 2017. URL: <https://www.ffmpeg.org>.
- 9 Fluent ffmpeg-API for node.js, 2017. URL: <https://github.com/fluent-ffmpeg/node-fluent-ffmpeg>.
- 10 Intel XDK, 2017. URL: <https://software.intel.com/en-us/xdk>.
- 11 Jimp: JavaScript Image Manipulation Program, 2017. URL: <https://github.com/oliver-moran/jimp>.
- 12 mjs, 2017. URL: <https://github.com/cesanta/mjs>.
- 13 TIOBE Index, 2017. URL: <https://www.tiobe.com/tiobe-index/>.
- 14 node-minify - npm, 2018. URL: <https://www.npmjs.com/package/node-minify>.
- 15 Saba Alimadadi, Sheldon Sequeira, Ali Mesbah, and Karthik Pattabiraman. Understanding javascript event-based interactions. In *Proceedings of the 36th International Conference on Software Engineering*, pages 367–377. ACM, 2014.
- 16 Esben Andreasen, Liang Gong, Anders Møller, Michael Pradel, Marija Selakovic, Koushik Sen, and Cristian-Alexandru Staicu. A survey of dynamic analysis and test generation for javascript. *ACM Computing Surveys (CSUR)*, 50(5):66, 2017.
- 17 Silviu Andrica and George Candea. Warr: A tool for high-fidelity web application record and replay. In *Dependable Systems & Networks (DSN), 2011 IEEE/IFIP 41st International Conference on*, pages 403–410. IEEE, 2011.
- 18 Federico Bellucci, Giuseppe Ghiani, Fabio Paternò, and Carmen Santoro. Engineering javascript state persistence of web applications migrating across multiple devices. In *Proceedings of the 3rd ACM SIGCHI symposium on Engineering interactive computing systems*, pages 105–110. ACM, 2011.
- 19 Brian Burg, Richard Bailey, Andrew J Ko, and Michael D Ernst. Interactive record/replay for web application debugging. In *Proceedings of the 26th annual ACM symposium on User interface software and technology*, pages 473–484. ACM, 2013.
- 20 Tiancheng Chang and Hein Meling. Byzantine fault-tolerant publish/subscribe: A cloud computing infrastructure. In *Reliable Distributed Systems (SRDS), 2012 IEEE 31st Symposium on*, pages 454–456. IEEE, 2012.
- 21 Ioannis K Chaniotis, Kyriakos-Ioannis D Kyriakou, and Nikolaos D Tselikas. Is node.js a viable option for building modern web applications? a performance evaluation study. *Computing*, 97(10):1023–1044, 2015.
- 22 Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pages 273–286. USENIX Association, 2005.
- 23 Douglas Crockford. *JavaScript: The Good Parts: The Good Parts*. " O'Reilly Media, Inc.", 2008.

- 24 Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, 2003. doi:10.1145/857076.857078.
- 25 Paul Fremantle, Benjamin Aziz, Jacek Kopecký, and Philip Scott. Federated identity and access management for the internet of things. In *Secure Internet of Things (SIoT), 2014 International Workshop on*, pages 10–17. IEEE, 2014.
- 26 Julien Gascon-Samson, Mohammad Rafiuzzaman, and Karthik Pattabiraman. Thingsjs: Towards a flexible and self-adaptable middleware for dynamic and heterogeneous iot environments. In *Proceedings of the 4th Workshop on Middleware and Applications for the Internet of Things, M4IoT '17*, pages 11–16, 2017.
- 27 Evgeny Gavrin, Sung-Jae Lee, Ruben Ayrapetyan, and Andrey Shitov. Ultra lightweight javascript engine for internet of things. In *SPLASH Companion 2015*, pages 19–20, New York, NY, USA, 2015. ACM.
- 28 Jayavardhana Gubbi, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswami. Internet of things (iot): A vision, architectural elements, and future directions. *Future generation computer systems*, 29(7):1645–1660, 2013.
- 29 Jin-woo Kwon and Soo-Mook Moon. Web application migration with closure reconstruction. In *Proceedings of the 26th International Conference on World Wide Web, WWW '17*, pages 133–142, Geneva, Switzerland, 2017.
- 30 Amit Levy, Bradford Campbell, Branden Ghena, Daniel B Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. Multiprogramming a 64kb computer safely and efficiently. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 234–251. ACM, 2017.
- 31 Jimmy Lin and Kareem El Gebaly. The future of big data is... javascript? *IEEE Internet Computing*, 20(5):82–88, 2016.
- 32 James Teng Kin Lo, Eric Wohlstadter, and Ali Mesbah. Imagen: Runtime migration of browser sessions for javascript web applications. In *Proceedings of the 22Nd International Conference on World Wide Web, WWW '13*, pages 815–826, New York, NY, USA, 2013. ACM.
- 33 Magnus Madsen, Ondřej Lhoták, and Frank Tip. A model for reasoning about javascript promises. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):86, 2017.
- 34 James W Mickens, Jeremy Elson, and Jon Howell. Mugshot: Deterministic capture and replay for javascript applications. In *NSDI*, volume 10, pages 159–174, 2010.
- 35 Dejan S. Milošević, Fred Douglass, Yves Paindaveine, Richard Wheeler, and Songnian Zhou. Process migration. *ACM Comput. Surv.*, 32(3):241–299, 2000. doi:10.1145/367701.367728.
- 36 JinSeok Oh, Jin-woo Kwon, Hyukwoo Park, and Soo-Mook Moon. Migration of web applications with seamless execution. In *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE '15*, pages 173–185, New York, NY, USA, 2015. ACM. doi:10.1145/2731186.2731197.
- 37 Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. Jalangi: A selective record-replay and dynamic analysis framework for javascript. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 488–498. ACM, 2013.
- 38 Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5):637–646, 2016.
- 39 Dongig Sin and Dongkun Shin. Performance and resource analysis on the javascript runtime for iot devices. In *International Conference on Computational Science and Its Applications*, pages 602–609. Springer, 2016.

- 40 Meena Singh, MA Rajan, VL Shivraj, and P Balamuralidhar. Secure mqtt for internet of things (iot). In *Communication Systems and Network Technologies (CSNT), 2015 Fifth International Conference on*, pages 746–751. IEEE, 2015.
- 41 A. Taivalsaari and T. Mikkonen. A roadmap to the programmable world: Software challenges in the iot era. *IEEE Software*, 34(1):72–80, Jan 2017. doi:10.1109/MS.2017.26.
- 42 Stefan Tilkov and Steve Vinoski. Node.js: Using javascript to build high-performance network programs. *IEEE Internet Computing*, 14(6):80–83, 2010.
- 43 Timothy Wood, Prashant J Shenoy, Arun Venkataramani, Mazin S Yousif, et al. Black-box and gray-box strategies for virtual machine migration. In *NSDI*, volume 7, pages 17–17, 2007.
- 44 Young Yoon, Vinod Muthusamy, and Hans-Arno Jacobsen. Foundations for highly available content-based publish/subscribe overlays. In *Distributed Computing Systems (ICDCS), 2011 31st International Conference on*, pages 800–811. IEEE, 2011.
- 45 Amirreza Zarrabi. A generic process migration algorithm. *International Journal of Distributed and Parallel Systems*, 3(5):29, 2012.
- 46 S. Zhongyuan, Q. Jianzhong, L. Shukuan, and Z. Qiang. Use pre-record algorithm to improve process migration efficiency. In *2015 14th International Symposium on Distributed Computing and Applications for Business Engineering and Science (DCABES)*, pages 50–53, Aug 2015. doi:10.1109/DCABES.2015.20.

A ThingsJS and ThingsMigrate

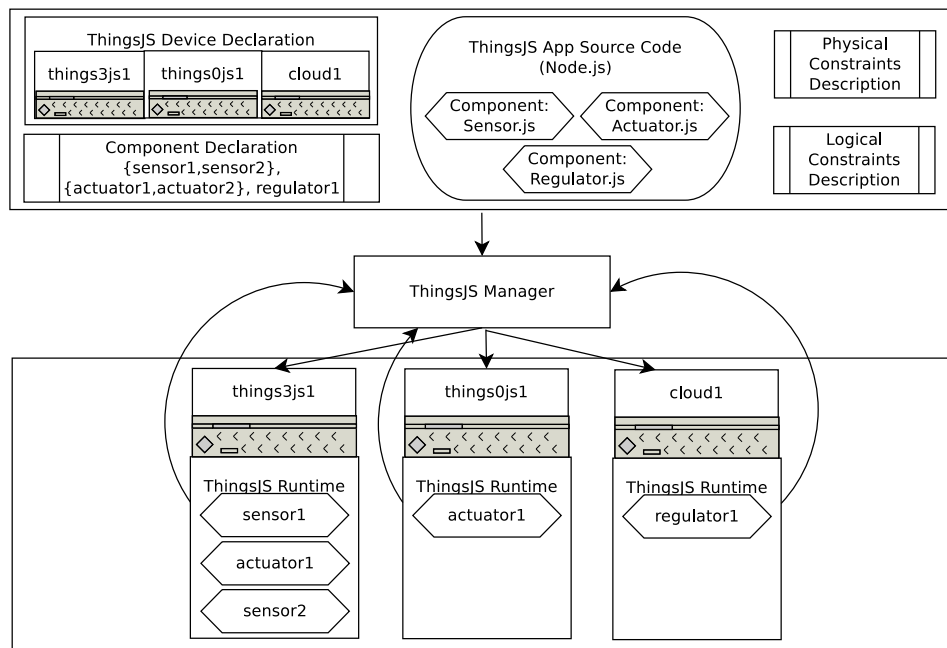
As mentioned previously, the implementation of our system (ThingsMigrate) was realized over ThingsJS [26], our general-purpose framework for executing high-level *edge* applications on IoT devices. In this appendix, we provide a brief summary of ThingsJS below, and its relationship with ThingsMigrate. We then give some details on our open-source implementation of ThingsMigrate/ThingsJS.

A.1 Architecture

The high-level architecture of the ThingsJS Framework consists of several distributed components and is presented in Figure 14. From a holistic point of view, a ThingsJS environment comprises a highly-distributed ThingsJS *Application*, and dynamically manages its execution over a set of heterogeneous *devices* through the ThingsJS *Framework*. More details on ThingsJS are available in our vision paper [26].

ThingsJS Manager. The ThingsJS Manager is the center-piece of our system architecture, and manages the execution of all components across all devices. It takes as input a ThingsJS application, written in JavaScript, and schedules and monitors its distributed execution across all participating ThingsJS devices. The Manager can decide to trigger the migration of a given application towards another device – to accomplish this, it uses the ThingsMigrate APIs.

ThingsJS Runtime. An instance of the ThingsJS Runtime is present on every device and acts as a thin hypervisor layer. It locally manages the execution of all components on the device, and gathers detailed statistics during execution (CPU, memory and bandwidth usage) which are fed to the *Manager*. Upon the migration of a given application being requested, ThingsMigrate coordinates the migration through the *Runtime* components on both devices involved.



■ **Figure 14** High-Level Architecture of ThingsJS.

Inter-Component Communications. ThingsJS provides a pub/sub-based communication substrate (MQTT), and requires that all inter-component communications follow that model. The choice of this model was primarily motivated by the logical decoupling of content producers from content consumers that it provides. Like any other component, ThingsMigrate makes use of the pub/sub communication substrate for all communications (i.e., migration commands and snapshots are transferred directly between relevant Runtime nodes through the pub/sub interface).

ThingsMigrate. As a subcomponent of ThingsJS, ThingsMigrate provides support for dynamically migrating JavaScript IoT applications between devices, and is the focus of this paper. More details on the architecture on ThingsMigrate and its components are given in Section 3 and Figure 1 of this paper. In our specific implementation, as scheduling considerations are outside the scope of our paper, we let the user deploy applications manually, monitor their state and trigger migrations, through a web dashboard interface (appendix B).

A.2 Implementation as an Open-Source Project

As mentioned, we implemented ThingsMigrate as an open-source project (built over ThingsJS). The version of ThingsMigrate/ThingsJS that correspond to this paper has been tagged as `ecoop2018` in our GitHub repository and can be accessed at: <https://github.com/karthikp-ubc/ThingsJS/tree/ecoop2018>. Given the availability of similar hardware and software configurations, it can be used to reproduce the results that we obtained.

As ThingsJS provides a IoT-based middleware, it needs be installed and run on every device. Each device then executes a *worker* node (corresponding to the ThingsMigrate/ThingsJS Runtime in our architecture) that hosts one or more JavaScript applications. Worker nodes listen for appropriate `start` and `stop` commands over the pub/sub interface to respectively start and stop the execution of applications, as well as `migrate` commands to trigger the migration between two nodes.

Detailed installation and usage instructions can be found at our project page⁷ and in our wiki⁸. In addition, we also provide a video demo of ThingsMigrate⁹, as well as a ready-to-use VirtualBox Virtual Machine image¹⁰ that can be used to try ThingsMigrate and our web dashboard (appendix B).

We encourage the reader to try out our system – using either IoT-like devices (i.e., Raspberry Pi, Beaglebone, etc.), or regular computers (multiple *worker* instances can be launched on the same machine to emulate additional devices).

B Web Dashboard

In addition to implementing the ThingsMigrate system over ThingsJS, as well as the code instrumentation, snapshotting and code generation algorithms as a set of command-line tools, we also implemented a *Web Dashboard* as a user-friendly interface to interact with ThingsMigrate. This appendix highlight the main features of our dashboard.

B.1 Interface Overview

Our web dashboard can be used to view the status of the currently available devices and applications, and can also launch, migrate and stop the execution of applications across devices. Given that the scheduling infrastructure is not yet implemented, the dashboard plays the role of the *ThingsMigrate Manager* by allowing the the user to control the deployment and the migration of applications to IoT devices manually.

Screenshots of the dashboard are shown in Figure 15. The dashboard provides three different views, which can be selected through the menu:

1. **Main (default) view** (Figure 15a): this view provides a holistic view of all the nodes (devices – IoT or cloud-based), their detailed status, performance (CPU and memory usage) and console output. In addition, specific to the video streaming / motion detection case-study application (Section 7), this view allows one to observe the raw video stream as well as the detected motion patterns.
2. **Codes view** (Figure 15b): this view provides a code viewer and editor to view and edit the source code of the IoT apps to be run on the devices. Developers can write their apps there directly, or cut-and-paste from their favorite editor / IDE into the code editor.
3. **Debug view**: this view provides network debugging support by showing the flow of pub/sub messages across the pub/sub substrate.

B.2 Main Features

In a nutshell, our dashboard provides the following features:

- **Viewing the state of worker nodes** (*Main view*, similar to Figure 15a): in the top-left portion of the main view, a list of all registered worker nodes is shown, with their status:
 - **Green**: the node is currently available and idle (i.e., not executing any application)¹¹
 - **Grey**: the node is currently available, but busy (i.e., executing another application)
 - **Red**: the node is currently unavailable

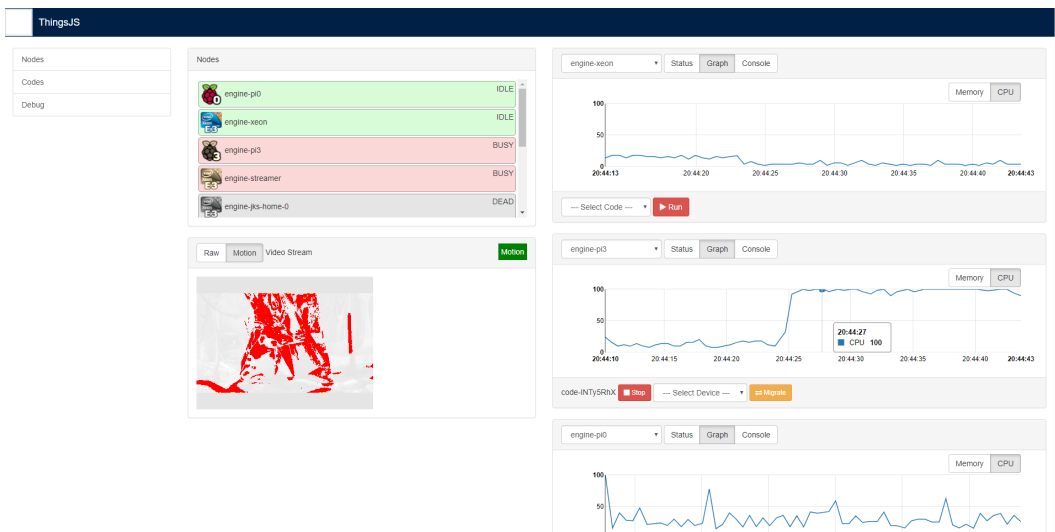
⁷ <https://github.com/karthikp-ubc/ThingsJS>

⁸ <https://github.com/karthikp-ubc/ThingsJS/wiki>

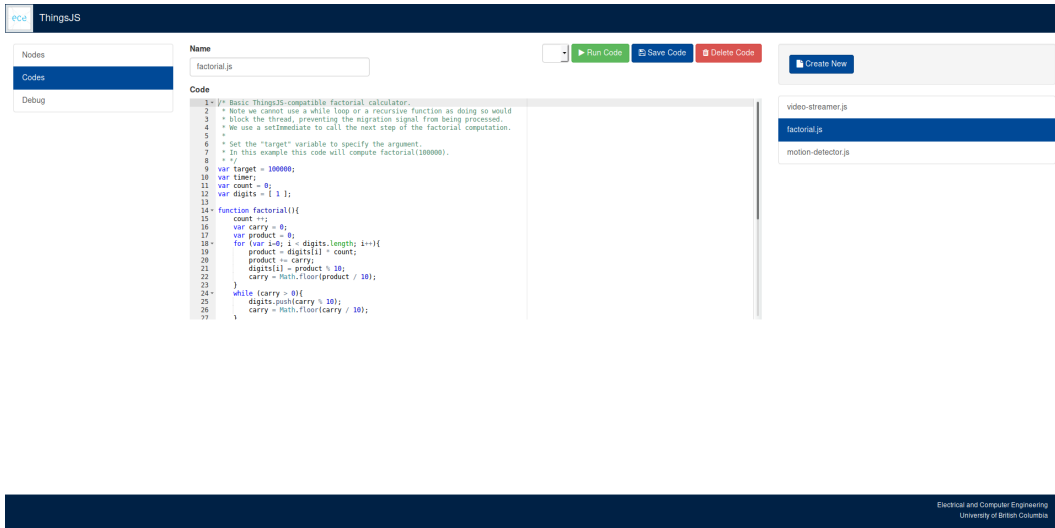
⁹ <http://ece.ubc.ca/~karthikp/ThingsMigrate/ecoop2018.html>

¹⁰ Idem.

¹¹ Although our framework supports executing several applications on a given worker node, our web dashboard currently allows for only one application to be mapped to a given worker node.



(a) Overview.



(b) IoT Apps Source Code.

■ **Figure 15** ThingsMigrate Dashboard.

- **Viewing detailed worker status (Main view):** on the right side, the interface provides three panes that can be used to show detailed information on a given worker (IoT/cloud) node. Each status pane provides three different views:
 - **Status:** shows the status of the node
 - **Graph:** shows a graph of the memory or CPU usage of the node
 - **Console:** shows the output of the application that the node is currently executing
- **Launching applications on devices (Main view):** the dashboard can be used to launch any application defined in the **Codes** section of the dashboard. The code is *instrumented* on-the-fly by ThingsMigrate (Section 4.5) in order to support migration, and is launched through the Runtime on the target device.
- **Stopping applications (Main view):** the execution dashboard can instruct the Runtime on any target device to stop the execution of a given application.

- **Migrating applications between nodes** (*Main view*): the dashboard can trigger the migration of an application from one device to another. When doing so, it instructs the Runtime on the first device to initiate the migration. The Runtime will then pause the execution, take a snapshot of the current state, send the state to the Runtime of the second device, and instruct the second device to resume the execution (Figure 1).

Automating Object Transformations for Dynamic Software Updating via Online Execution Synthesis

Tianxiao Gu

State Key Laboratory of Novel Software Technology, Nanjing University, Nanjing, China
tianxiao.gu@gmail.com

Xiaoxing Ma¹

State Key Laboratory of Novel Software Technology, Nanjing University, Nanjing, China
xxm@nju.edu.cn

Chang Xu

State Key Laboratory of Novel Software Technology, Nanjing University, Nanjing, China
changxu@nju.edu.cn

Yanyan Jiang

State Key Laboratory of Novel Software Technology, Nanjing University, Nanjing, China
jyy@nju.edu.cn

Chun Cao

State Key Laboratory of Novel Software Technology, Nanjing University, Nanjing, China
caochun@nju.edu.cn

Jian Lu

State Key Laboratory of Novel Software Technology, Nanjing University, Nanjing, China
lj@nju.edu.cn

Abstract

Dynamic software updating (DSU) is a technique to upgrade a running software system on the fly without stopping the system. During updating, the runtime state of the modified components of the system needs to be properly transformed into a new state, so that the modified components can still correctly interact with the rest of the system. However, the transformation is non-trivial to realize due to the gap between the low-level implementations of two versions of a program. This paper presents AOTES, a novel approach to automating object transformations for dynamic updating of Java programs. AOTES bridges the gap by abstracting the old state of an object to a history of method invocations, and re-invoking the new version of all methods in the history to get the desired new state. AOTES requires no instrumentation to record any data and thus has no overhead during normal execution. We propose and implement a novel technique that can synthesize an equivalent history of method invocations based on the current object state only. We evaluated AOTES on software updates taken from Apache Commons Collections, Tomcat, FTP Server and SSHD Server. Experimental results show that AOTES successfully handled 51 of 61 object transformations of 21 updated classes, while two state-of-the-art approaches only handled 11 and 6 of 61, respectively.

2012 ACM Subject Classification Software and its engineering → Software evolution

Keywords and phrases Dynamic Software Update, Program Synthesis, Execution Synthesis

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2018.19

¹ Corresponding author.



Acknowledgements We are grateful to the anonymous reviewers for their insightful comments. This work was supported by the National Natural Science Foundation of China (Grant Nos. 61690204, 61472177), the 973 Program of China (Grant No. 2015CB352202), and the Collaborative Innovation Center of Novel Software Technology and Industrialization.

1 Introduction

Today's industry definitely requires high availability of software systems. One of the major losses of availability is caused by system shutdowns for installing software updates that fix bugs and security vulnerabilities. *Dynamic Software Updating* (DSU) can eliminate this loss by updating running software systems without stopping them.

Modern operating systems and programming language virtual machines provide powerful runtime code manipulation facilities such as dynamic linking [8], dynamic class loading [28], on stack replacement [37] and live patch [2]. With these facilities, it is not difficult to update the code of a running program. In addition to code replacement, DSU also needs to ensure that the new loaded code can execute properly with the existing runtime state in the memory (e.g., heap objects) after the dynamic update.

Existing DSU supporting systems, e.g., Ginseng [33], Jvolve [41] and Javelus [15], ensure only syntactical correctness, i.e., no type error would be caused by the update. To preserve semantics correctness, the DSU system should apply an additional state transformation that maps the runtime state left by the old code to a proper state with which the new code continues. To realize the state transformation, developers usually specify a manually-prepared state transformation function, i.e., *state transformer*.

However, it is difficult and error-prone to develop and test state transformers. Software is seldom developed with DSU in mind but is assumed to start from scratch. The internal states between different versions of a program can be incompatible, although the external behavior of the two versions is similar. For example, the internal representation of a container may be an array in the old version but a linked list in the new version. As a result, transformer programming not only requires a thorough understanding of implementation details in both versions, but also has to break the principle of information hiding and manipulate low-level data representations.

In this paper we aim at automating the state transformations for DSU. In theory, it is not possible to automatically generate correct transformers for dynamic updates of arbitrary programs [17]. Nevertheless, in many practical cases, for particular software patches and particular dynamic update points, state transformations can be automatically derived with sophisticated program analysis under some proper assumptions. This kind of techniques can help reduce service disruption caused by software updates, and are useful for application domains where high availability is the major concern and occasional errors are tolerable or compensable.

Our approach, named AOTES, is designed for DSU of object-oriented programs, or more specifically, Java programs. In object-oriented programming, an important principle is to use *information hiding* and *encapsulation*. An object should be interacted with only via its methods, where methods are closely related to the behavior of the object. Based on this principle, we have the following observations. First, the current state of an object is a conclusion of its past method invocations. Second, the current state is also the basis of the future method invocations. Third, the behavior of an object, or specifically the history of method invocations, is mostly unchanged during updating, especially when the patch does not include new behaviors. Thereby, the new state of a stale object (i.e., an instance of an

updated class) can be synthesized by replaying its past method invocations with the new version of methods. In this way we can avoid direct mapping of concrete states between two program versions with different implementations.

Specifically, AOTES abstracts the runtime state of a stale object as a *history of method invocations* (i.e., invocation history for short) that can produce the current state from an *initial (object) state*. For example, suppose that an array based container object with three elements e_1 , e_2 and e_3 is created by a history of `add(e_1)`, `add(e_2)`, `remove(e_2)`, `add(e_3)` and `add(1, e_2)`. Now a dynamic update requires transforming the array based container into a linked list based one. We can easily know that the new linked list based container can be naturally acquired by applying the invocation history with the new version of methods on an empty linked list based container.

The main challenge of this approach is to obtain the invocation history for a stale object. Recording every method invocation on every potentially updated object is prohibitively expensive. Moreover, the actual history may contain redundant elements, e.g., `remove(e_2)` and `add(1, e_2)`. To address these problems, we try to synthesize an equivalent but more compact invocation history from the current state. For the previous example, we can synthesize a history of `add(e_1)`, `add(e_2)` and `add(e_3)` instead of the actual one.

Unfortunately, it is hardly feasible to synthesize an invocation history using methods of real world programs. First, synthesizing a single method invocation is difficult because a method invocation generally requires arguments, which usually lie within a large value space (e.g., $[-2^{31}, 2^{31} - 1]$ for `int` in Java). Second, searching for a valid invocation history is time-consuming because method invocations need to be ordered properly to form a valid invocation history. In the scenario of dynamic updating, an additional challenge is that the synthesis is performed online and must be completed very quickly.

AOTES addresses these challenges by combining the power of symbolic execution, program synthesis and execution synthesis. Specifically, AOTES first distills a set of promising execution paths for each method by an offline symbolic execution technique. During dynamic updating, AOTES uses the selected execution paths only to realize a backward online execution synthesis technique. To reuse techniques of forward execution synthesis, AOTES synthesizes an inverse method for each selected execution path. We tried out AOTES on 21 real updates of widely used open source software. AOTES correctly handled 51 of 61 different transformations, while two state-of-the-art methods handled 11 and 6 of 61, respectively.

The paper makes the following primary contributions:

- We propose a mechanism to synthesize method invocation histories that can be used to recreate objects.
- We use the object recreating mechanism to automate object transformations for DSU.
- We implement the mechanism and evaluate it with updates taken from widely used open source systems.²

The rest of this paper is organized as follows. We first give an introduction to DSU and AOTES using an illustrative example in Section 2 and then a detailed overview of AOTES in Section 3. Next, we describe the offline analysis in Section 4 and online synthesis in Section 5. Then, we illustrate the implementation of AOTES in Section 6 and evaluate AOTES with updates from real-world software in Section 7. We summarize related work in Section 8 and conclude in Section 9.

² All source code and tests are publicly available at <http://moon.nju.edu.cn/dse/aotes>.

```

1 class DefaultSshFuture {
2     SshFutureListener firstListener;
3     List otherListeners;
4     void addListener(SshFutureListener listener) {
5         if (firstListener == null) {
6             firstListener = listener;
7         } else {
8             if (otherListeners == null) {
9                 otherListeners = new ArrayList(1);
10            }
11            otherListeners.add(listener);
12        }
13    }
14 }

```

(a) The old version of DefaultSshFuture.

```

1 class DefaultSshFuture {
2     Object listeners;
3     void addListener(SshFutureListener listener) {
4         if (listeners == null) {
5             listeners = listener;
6         } else if (listeners instanceof SshFutureListener) {
7             listeners = new Object[]{listeners, listener};
8         } else {
9             // Check the array bound
10            // Expand the array if necessary
11            // Append the listener
12        }
13    }
14 }

```

(b) The new version of DefaultSshFuture.

■ **Figure 1** An update (rev. b98694) of class DefaultSshFuture of Apache SSHD Server.

2 Illustrative Example

In this section, we present an introduction to DSU and AOTES using an illustrative example.

2.1 Dynamic Software Updating and Its Challenges

Software is subject to changes and evolution: Bugs are fixed and new features are introduced by applying *software updates*. Figure 1 shows a real-world motivating example of software update, which will be discussed throughout the paper. The update is from the Apache SSHD Server. Class `DefaultSshFuture` provides a method `addListener` to add listeners (Figure 1a). For most cases, there is only one or two listeners but the implementation should support adding more. To save memory, the old version saves the first-added listener to `firstListener` and others into `otherListeners`, which is an auto-expanding list container (`ArrayList`). The new version (Figure 1b) only uses a single field `listeners` and a raw array to handle all situations.

To allow long-running programs to receive timely updates without restart, dynamic software updating migrates the running program from the old version to a new version. Specifically, a DSU system takes over the execution of a running program, transforms the runtime state at a properly determined update point (e.g., when no updated method is active) to a new state conforming to the new version, and then continues executing with the new version [23, 41].

```

1 void update(DefaultSshFuture1 o, DefaultSshFuture2 n) {
2   if (o.firstListener != null) {
3     if (o.otherListeners == null) {
4       n.listeners = new Object[] {o.firstListener};
5     } else if (o.otherListeners.size() > 0) {
6       int length = o.otherListeners.size() + 1;
7       n.listeners = new Object[length];
8       n.listeners[0] = o.firstListener;
9       for (int i = 0; i < o.otherListeners.size(); i++) {
10        n.listeners[i + 1] = o.otherListener.get(i);
11      }
12    }
13  }
14 }

```

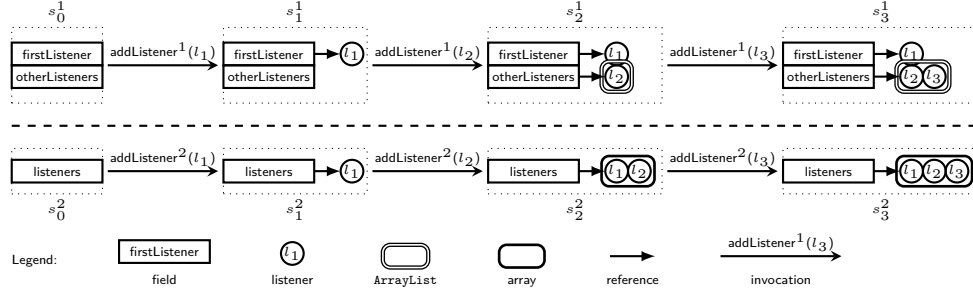
■ **Figure 2** A user-defined transformer for the example in Figure 1.

A major challenge in DSU is the state transformation at the update point. A runtime system's state consists of the *code*, the *stacks* and the *heap*. As new code can be easily dynamically re-loaded and stacks mostly remain unchanged at the update point, the main challenge is the state transformation of the heap. We restrict our discussions to object-oriented programming languages (e.g., Java) and thus the heap state transformation is particularly referred to as the *object transformation*.

An object transformation takes the current state of a stale object as input and produces the new state as output. The transformation must be *consistent*: The future execution must be able to continue from the transformed state and take over the ongoing business smoothly. None of existing approaches [41, 42, 31, 38] is capable of automatically conducting object transformations beyond trivial cases. Most state-of-the-art DSU systems [41, 42, 38] provide *default transformations* that simply copy the values of unchanged fields from a stale object to its corresponding new object, and initialize all new fields with *type-specific default values*, e.g., 0 for `int`.

TOS [31] is the only known approach to automating object transformations, which embodies the idea of learning-by-example. A transformation example consists of an old-version object, which is collected during running a test over the old version of the program, and a new-version object, which is collected during running the same test over the new version of the program at the corresponding time point [31]. After collecting sufficient examples, TOS inductively composes a function following a set of predefined rules until the composed function can realize the transformations between all examples. However, TOS relies on the high quality tests in terms of covering transformations not only in testing but also in production. Even though there are sufficient good examples, TOS may easily fail due to its poor predefined rules.

Both default transformations and TOS do not work for our motivating example because they only use *matched fields* (i.e., fields with the exactly same name and type) to transfer information from the stale state to the new state. In other words, neither of them can find the relation between *unmatched fields*, i.e., old fields (e.g., `firstListener` and `otherListeners`) and new fields (e.g., `listeners`) that have different names or types. The only solution before this paper is to ask the developer to provide an object state transformer, which is a non-trivial procedure tightly coupled with program semantics and low-level implementations. Figure 2 presents a manually prepared transformer for the update in Figure 1. Even though there may be only a single stale state at the updating point, the transformer has to handle various stale object states and produces the new object states accordingly by directly manipulating the data structure of the object.



■ **Figure 3** Object state evolution of `DefaultSshFuture`. s_i^v denotes the i -th state of the object in version v . Each state is depicted with a graphical representation of its data structure.

2.2 Object Transformation Using Method Invocation History

Object transformations will be easy if the method invocation histories of objects are available. A *method invocation* consists of a method and a sequence of arguments, which may be empty if the method requires no arguments. A method invocation usually accepts some specific *input state* of the receiver object and produces an *output state* accordingly.

A *method invocation history* (invocation history for short) is a sequence of method invocations. Similarly, an invocation history accepts some specific *initial state*, i.e., the input state of the first invocation, and produces the *final state*, i.e., the output state of the last invocation. During replaying an invocation history, every method invocation must produce a valid output state as the input state for the consecutive method invocation in the history.

Two invocation histories are *equivalent* if they can yield the same final state when applied to the same initial state. For every object, there is a unique *actual invocation history*, including all method invocations applied to the object in the chronological order. An invocation history is *complete* if it can yield the current state of an object from the empty state, e.g., the actual history. Note that nested methods are not included in an invocation history. For example, method `add` in Figure 4 can be included in an invocation history but nested methods such as `ensureCapacityInternal` cannot.

We have the following two *assumptions* for our approach:

1. The current state of an object is a summary of its past past method invocations. We can also recreate the current state of an object from its past method invocations, i.e., replaying every method invocation on an object from the initial state.
2. The “role” or the behavior of an object is not changed during update [31]. The method invocation history usually keeps unchanged for such objects during updates that introduce no new functionalities, e.g., bug fixes or performance improvements. Hence, the invocation history of the new state can be easily derived from the invocation history of the stale state.

Now, the idea can be explained in Figure 3 by an update of `DefaultSshFuture`. The program invokes `addListener1` (In this paper superscripts denote program versions). with l_1 , l_2 and l_3 , respectively, and the update point (s_3^1) is reached. The transformed new-version object is synthesized by applying the invocation history, i.e., invoking the new-version `addListener2` with l_1 , l_2 and l_3 on a newly allocated new-version object (s_0^2). State s_3^2 contains exactly the same sequence of listeners as s_3^1 , indicating that this is a semantically correct state transformation. In contrast to default transformations and TOS, which cannot connect unmatched fields, AOTES finds the relations between them by matching arguments of matched methods.

One limitation of our approach is that methods in the invocation history should be available in both versions. There is always a method with the same name and signature in the new version of an object whose interface is *not* changed. These methods are named *matched methods* for later discussion. AOTES does not insist that *every changed* class should preserve the binary compatibility. If some changed classes are binary incompatible, their callers must fix the incompatibility. In practice, there must be a direct or an indirect binary-compatible caller within a small scope, including one or two callers, since dynamic updating is often used for evolutionary changes (e.g., build-to-build) rather than revolutionary changes (e.g., release-to-release), and build-to-build changes usually do not introduce large patches. Suppose that a stale object's interface is changed. The invocation of an unmatched method on the object must be eventually enclosed in an invocation of a matched caller. We can enlarge the scope of the state being transformed to include all objects subject to the invocation of the matched caller and use the matched caller for history synthesis.

2.3 Synthesizing the Equivalent Invocation History

Recording method invocation histories for object transformations is *impractical* for long-running programs, because (1) we could not predict which objects were to be updated, thereby all method invocations on all objects would have to be recorded, which would introduce significant runtime overhead; (2) the log would be prohibitively expensive to store; and (3) replaying a long history could lead to a large service disruption during updating.

Alternatively, we try to find an *equivalent invocation history* that yields the same object state as the actual invocation history when applied to an object in the initial state, but is more *compact*, in terms of as few as possible redundant method invocations. For example, listeners can be added and removed for millions of times for a `DefaultSshFuture` object. However, at a specific execution point, only limited listeners are expected in the data structure. Any invocation history that yields exactly the same set of listeners suffices for a consistent object transformation.

AOTES synthesizes an object's equivalent invocation history from its current state without any logging. No history data need to be kept at runtime and no overhead is introduced when the program is not being updated. However, the synthesis of an invocation history is non-trivial because we need to first derive the arguments for a single method invocation and then find a valid history of method invocations. That means each method invocation must produce a valid output state as the input state for its consecutive method invocation in the synthesized history and the final state must be the current state of the object.

A naïve approach would enumerate all possible combinations of methods and arguments to determine the set of all possible method invocations. Since this searching space of invocation histories is huge, it is expensive to find an invocation history that can realize the state transition from the empty state to the current state of a given object. To narrow down the searching space for arguments [44, 5], execution synthesis techniques leverage on symbolic execution and constraint solvers. However, these approaches aim at searching for an execution path that reaches a particular statement, while AOTES aims at searching for an execution path that produces the given output state on a given input state of the receiver. In addition, these approaches are used for offline scenarios such as crash reproduction and search in the space of all execution paths, which may lead to a potentially unbounded searching time for real-world programs.

We observed that multiple execution paths of a method actually have the same purpose. To derive arguments of a method invocation, AOTES applies an offline analysis to populate a set of promising execution paths before dynamic updating, and during dynamic updating

```

1 class ArrayList<E> {
2     int size; E[] elements = {};
3     public boolean add(E e) {
4         ensureCapacityInternal(size + 1);
5         elements[size++] = e;
6         return true;
7     }
8     private void ensureCapacityInternal(int minCapacity) {
9         if (minCapacity - elements.length > 0)
10            grow(minCapacity);
11    }
12    private void grow(int capacity) {...}
13    public boolean addAll(Collection<? extends E> c) {...}
14 }

```

■ **Figure 4** A simplified version of class `ArrayList` in JDK.

```

1 public boolean add(E e) {
2     if (size + 1 < elements.length)
3         elements[size++] = e;
4     return true;
5 }

```

■ **Figure 5** The simplified equivalent version of method `add`.

applies an online execution synthesis that considers these execution paths only. Typically, a method with multiple paths usually has a *fast path* that handles the most common situation and many *slow paths* that handle the rest cases. Moreover, the length of the execution path is usually guided by some input. We found that first the fast path is sufficient during execution synthesis for some methods, and second a long execution path guided by a large input can be replaced by many short execution paths guided by a small input.

Take the program in Figure 4 as an example. Method `add` has a fast path that appends the added element directly into the array (at line 5) and many slow paths that need to additionally calculate the new array size and expand the array (at line 10). AOTES can use the fast path only to synthesize the invocation history as if the array is initially allocated in the current size without any expansion. By this way, we can exclude the slow path (i.e., the call to `grow`) during execution synthesis. The fast path of `add` can be expressed by the method in Figure 5. Besides, an invocation of `addAll` with a large input collection in the actual history can be replaced by many invocations of `addAll` with a small input collection, or even many invocations of `add` with a single element.

To avoid backtracking, AOTES conducts a greedy backward searching starting from the current state instead of a forward searching starting from the initial state. Instead of every step searching for a method invocation that is applicable to a given input state, AOTES searches for a method invocation that can produce a given output state. This is because the initial input state (i.e., the empty state) has zero information to guide the search, while the final output state (i.e., the current state) has fruitful information. For example, if we synthesize a history for an array list from the empty state, we may include many method invocations that add or remove irrelevant elements. But if we synthesize the history backward from the current state, we can require that every method invocation must at least contribute to a field with a non-default value in the current output state.

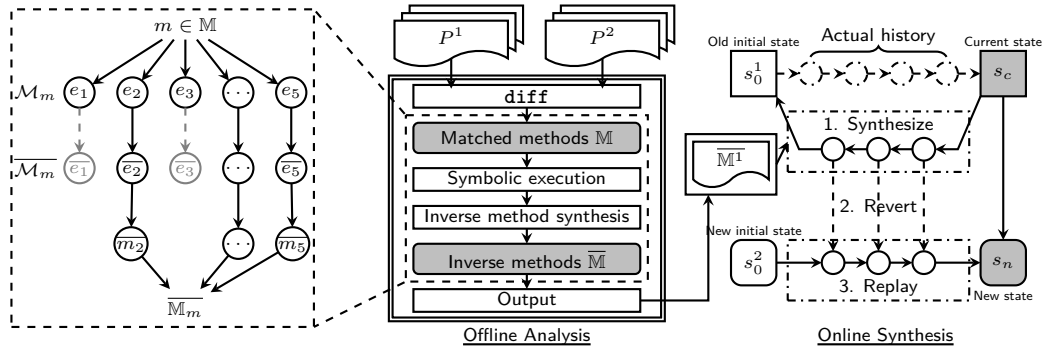
To facilitate the backward searching, AOTES converts each execution path into a separated *inverse method* by the offline analysis. The benefit is that we can simply make use of existing

```

1 Object[]  $\overline{\text{add}}$ () {
2   size--;
3   assert(size + 1 < elements.length);
4   return new Object[] { elements[size] };
5 }

```

■ **Figure 6** The inverse method of `add` shown in Figure 5.



■ **Figure 7** System overview.

symbolic execution techniques to realize backward execution synthesis. Figure 6 shows an inverse method $\overline{\text{add}}$ of the method `add` shown in Figure 5. Here, an inverse method generally takes no arguments but only the receiver as input, reverts the receiver to a previous state (e.g., line 2 in Figure 6), and finally returns an array of values (e.g., line 4 in Figure 6). The return values can be used as the arguments to replay the invocation history of original methods. For example, suppose that an object of `ArrayList` is in state o , which contains two elements e_1 and e_2 . After invoking $\overline{\text{add}}$ on the object, the state o becomes o' , which contains only a single element e_1 , and the return value of $\overline{\text{add}}$ is e_2 . If we invoke `add` (in Figure 4) on o' with e_2 , the state will be updated from o' to o again. Here, the input of a method consists of both the arguments and the state of the receiver, and the output of a method consists of both the return value and the final state of the receiver.

3 Approach Overview

Figure 7 presents an overview of our approach. AOTES aims at automatically constructing the new state s_n based on the current state s_c only. There must be an actual history \mathcal{H}_a^1 that leads to s_c . Instead of recording \mathcal{H}_a^1 from scratch, AOTES tries to synthesize a history \mathcal{H}_s^1 that can also lead to s_c . As the state of an object is a summary of its past method invocations, \mathcal{H}_a^1 and \mathcal{H}_s^1 should encode the same behavior accordingly. We assume that the role and the behavior of an object is unchanged during updating. Therefore, \mathcal{H}_s^1 can be used to recreate the new state s_n .

AOTES first conducts an offline analysis, which takes two versions of a program (P^1 and P^2) as input, and tries to produce the following output:

- A number of *execution summaries* (e_m) for each matched method m in P^1 . Using symbolic execution, AOTES builds a map $\mathcal{M}_m : \mathcal{S} \times \mathcal{P} \rightarrow \mathcal{S}$ from the symbolic pre-state $\Sigma_{pre} \in \mathcal{S}$, i.e., symbolic object state before applying this method, and the symbolic arguments $\Psi \in \mathcal{P}$, to the symbolic post-state $\Sigma_{post} \in \mathcal{S}$, i.e., symbolic object state after applying this method.

- A number of *inverse execution summaries* $\overline{e}_m \in \overline{\mathcal{M}}_m : \mathcal{S} \rightarrow \mathcal{S} \times \mathcal{P}$, each of which corresponds to an execution summary in \mathcal{M}_m . An inverse execution summary takes a concrete state s aligned with the symbolic post-state Σ_{post} and computes a concrete state s' aligned with the symbolic pre-state Σ_{pre} and concrete arguments p such that applying m with p on an object in state s' will change its state to s .

To facilitate the online searching, AOTES serializes an inverse execution summary into an *inverse method*. An inverse method \overline{m} takes only the receiver as input and reverts the state of the receiver to the state just before invoking the original method m . Moreover, it also returns all arguments required by the invocation of the original method m . For example, suppose that `addListener1` is an inverse method of `addListener1`. We can obtain s_2^1 if we apply `addListener` with l_2 to s_1^1 . Conversely, we can obtain s_1^1 and l_2 if we apply `addListener1` on s_2^1 .

Next, AOTES attempts to synthesize an *inverse method invocation history* (*inverse invocation history* for short) for the object and revert the object to the initial state. An inverse invocation history is a sequence of inverse methods and return values (i.e., the corresponding reverted arguments). For example, $\langle \text{addListener}^1/l_2, \text{addListener}^1/l_1 \rangle$ is a synthesized inverse invocation history for s_2^1 . The inverse invocation history can be synthesized by concatenating inverse methods as all inverse methods only take the object as input. Specifically, the inverse invocation history $\overline{\mathcal{H}}_s^1$ is synthesized by searching for a sequence of inverse execution summaries $\overline{e}_{m_i}, \overline{e}_{m_{i-1}}, \dots, \overline{e}_{m_1}$, such that $\overline{e}_{m_1}(\overline{e}_{m_2}(\dots(\overline{e}_{m_i}(s_c)))) = s_0$ and as well $e_{m_i}(e_{m_{i-1}}(\dots(e_{m_1}(s_0)))) = s_c$.

Finally, AOTES constructs a new invocation history by inverting the inverse invocation history, substituting every inverse method with the new version of its original method, and using the return value of each inverse method as the arguments. The new state can be reified by applying the new invocation history on the new initial state. For example, an invocation history $\langle \text{addListener}^2(l_1), \text{addListener}^2(l_2) \rangle$ can be constructed by reverting the inverse invocation history $\langle \text{addListener}^1/l_2, \text{addListener}^1/l_1 \rangle$.

If the input of the original method cannot be derived by executing the inverse method, AOTES introduces a fresh symbolic variable for the input and leverages symbolic execution techniques to derive its value during online execution synthesis. AOTES considers a set of *short* execution paths as the promising candidates for execution synthesis. This is because long execution paths generally produce long path constraints that may not be solved by a constraint solver. Moreover, short paths can also help to mitigate the problem of long-running loop and deep recursive methods whose executions are guided by some input [43]. A long execution path guided by a very large input is replaced by many short execution paths, each of which is guided by a small input. An example about loop and recursion with detailed explanation is available in Section 4.4.

Figure 8 shows three inverse methods of `addListener1` (in Figure 1a) generated by AOTES. Note that the generated code has been simplified for brevity. We can obtain s_0^1 and l_1 when applying `addListener1` to s_1^1 . Specifically, `addListener1` first loads l_1 from `firstListener` at line 2 and returns it at line 5, and then updates `firstListener` with a fresh symbolic value (denoted by a wild-card `*`) at line 3. The assertion at line 4 restricts the fresh symbolic value to be `null`, which can be derived by a constraint solver. Thereby, `firstListener` is `null` and the state becomes s_0^1 if the previous state is s_1^1 .

Similarly, we can obtain s_1^1 and l_2 by applying `addListener2` to s_2^1 . Let's first analyze the original execution trace, i.e., lines 5, 8, 9, and 11 in Figure 1a. Specifically, the argument `listener` is l_2 , and the input state is s_1^1 , in which `firstListener` references l_1 and `otherListeners` is `null`. Then, `otherListeners` is assigned to a newly allocated


```

1 Object [] addListener11() {
2   v0 = firstListener;
3   v1 = firstListener = *;
4   assert(v1 == null);
5   return { v0 };
6 }
7 Object [] addListener21() {
8   v0 = firstListener;
9   v1 = otherListeners;
10  v2 = v1.size;
11  v3 = v1.elements;
12  v4 = v3.length;
13  v5 = v3[0];
14  v3[0] = *;
15  assert(v4 == 1);
16  assert(v2 == 1);
17  v6 = otherListeners = *;
18  assert(v0 != null);
19  assert(v6 == null);
20  return { v5 };
21 }
22 Object [] addListener31() {
23  v0 = otherListeners;
24  v1 = v0.size;
25  v2 = v0.elements;
26  v3 = firstListener;
27  v4 = v1 - 1;
28  v0.size = v4;
29  v5 = v2[v4];
30  v2[v4] = *;
31  assert(v0 != null);
32  assert(v3 != null);
33  return { v5 };
34 }

```

■ **Figure 8** Inverse methods of `addListener` in Figure 1a generated by AOTES. The execution traces of lines 5 and 6, lines 5, 8, 9, and 11, and lines 5, 8, and 11 in Figure 1a are postfixed with 1, 2 and 3, respectively. Note here we have simplified the generated code for brevity.

`ArrayList`, in which `size` is initialized to 0 and `elements` is assigned to an empty array. After executing `otherListeners.add`, `elements` is expanded to an array of length 1, `size` is updated to 1, and the argument l_2 is placed at the first (index 0) slot of `elements`. Note that the `ArrayList` is the simplified implementation shown in Figure 4. Now we analyze the inverse method `addListener2`¹ shown in Figure 8. Lines 8 and 18 assert that `firstListener` should not be `null`. Line 17 reverts `otherListeners` to a fresh symbolic value, which is further assigned to `null` by the assertion at line 19. Line 13 retrieves l_2 from `elements` by index 0 and line 20 returns l_2 . We can easily verify that the state of the receiver is updated to s_1^1 at last, where `firstListener` still references l_1 and `otherListeners` is `null`. The third inverse method `addListener3`¹ is similar, where the return value l_3 is retrieved from `elements` by index `size - 1`.

AOTES can sacrifice the completeness because it does not aim at synthesizing all possible transformations. The objects that AOTES cannot handle may be disposed at a later update point. On the other hand, a fixed number of inverse methods are sufficient for all transformations in practice. In Section 5.1, we will show that three inverse methods are sufficient for all transformations of `DefaultSshFuture`.

4 Inverse Program Synthesis

The insight of AOTES is to synthesize an inverse method from a symbolic execution trace not from all traces. We first give a high level overview of the symbolic execution technique of AOTES followed by a detailed description before illustrating the details.

4.1 Symbolic Execution of AOTES

In general, *symbolic execution* [25] is a technique to interpret a program with symbolic values instead of concrete values. A symbolic value is a formula over a set of *symbolic inputs*, and can be evaluated to a concrete value by substituting symbolic inputs with concrete values and then evaluating the formula.

AOTES populates a certain number of symbolic execution traces from a matched method to generate inverse methods. The symbolic execution technique of AOTES needs to allocate objects with explicit types, because dynamic method dispatching should know the type of each object. This requirement makes it non-trivial to symbolically execute an arbitrary method of an object, because the heap, or at least the receiver, must be instantiated in to a proper shape before execution. We name this heap *pre-heap* (Π_{pre}).

For example, suppose that a symbolic execution trace of `addListener` in Figure 1a explores lines 5, 8 and 11. Invoking `add` at line 11 should trigger a `NullPointerException` (NPE) if `otherListeners` does not reference an object. Otherwise, we have no idea about exploring which `add` method. To avoid the NPE and continue the execution, one can allocate an object in the pre-heap for `otherListeners` before the execution. However, we have no idea about the type for object allocation as there may be numerous subclasses of `List`. The type for object allocation should be as exact as possible. Here, the type must be `ArrayList` not any other type.

During the symbolic execution, an object is either *pre-allocated* in the pre-heap before execution or *newly allocated* during execution. The type of a newly allocated object is known at its allocation site. For pre-allocated objects, AOTES maintains a shared dictionary \mathbb{S} that maps an *access path* (e.g., `this.otherListeners`) to a set of types. A type is randomly picked out for the pre-allocated object if there are multiple types for an entry. AOTES will produce inverse methods for each randomly chosen type. During runtime execution synthesis, only inverse methods that match the actual type are applicable.

The dictionary \mathbb{S} is empty at first and updated by traversing the heap at the end of every successful symbolic execution, which is named *post-heap* (Π_{post}). Note that at any time, only *live* objects in a heap are of interest. The execution trace that explores lines 5, 8 and 11 depends on the type at `this.otherListeners` in \mathbb{S} . Hence, the execution should be first suspended at line 11 and resumed until some other symbolic execution trace updates the entry. Fortunately, the execution that explores lines 5, 8, 9 and 11 can update the entry. Line 9 allocates an `ArrayList` for `otherListeners`. The entry at `this.otherListeners` in \mathbb{S} is updated by `ArrayList`.

To update missing entries in \mathbb{S} , AOTES dynamically collects extra methods to execute. If the missing entry is rooted at the receiver, all methods of the receiver are added. If the missing entry is rooted at an argument of the entry method of the symbolic execution, all callers of the entry method are added. Callers of a method are determined by a call graph. AOTES constructs a static call graph at first and refines it when invoking a method during symbolic execution.

4.2 Program and Execution Definitions

This subsection gives a detailed description of the symbolic execution technique of AOTES. A program in AOTES is a set of classes. A class \mathbb{C} is a set of fields \mathbb{F} and a set of methods \mathbb{M} , which also include those inherited from super classes. Every method has a *receiver* and an optional sequence of *parameters*. A method is a sequence of Java virtual machine bytecode instructions [29]. A bytecode instruction may allocate new objects, create new values, copy or move existing values, and evaluate branch conditions and change control flow accordingly.

We group all bytecode instructions into seven groups, which are shown in Table 1. A bytecode instruction may have one operand encoded with it. In a nutshell, this kind of operand may be an array index i , a field f , a method m , a class \mathbb{C} , a constant c , or an offset ρ of instruction index. AOTES can handle almost all bytecode instructions, except `invokedynamic`. This is because `invokedynamic` usually needs to execute a piece of custom

■ **Table 1** Bytecode instructions.

Type	Instructions
Stack & Local	<code>ldc c</code> , <code>load i</code> , <code>store i</code>
Array Access	<code>aload</code> , <code>astore</code>
Field Access	<code>getfield f</code> , <code>putfield f</code>
Allocation	<code>new C</code> , <code>newarray</code>
Binary Operator	<code>add</code> , <code>sub</code> , <code>mul</code> , <code>div</code> , <code>rem</code>
Branch	<code>ifgt ρ</code> , <code>ifeq ρ</code> , <code>iflt ρ</code> , <code>goto ρ</code>
Invoke & Return	<code>invoke m</code> , <code>return</code>

code to resolve the callee. We list a single `invoke` instruction only without showing various method dispatching semantics (e.g., `invokevirtual` and `invokespecial`), because AOTES tracks the type of every object and method dispatching for these `invoke` instructions is straightforward if we know the receiver type.

An object, i.e., an *instance* of a class or an *array*, is defined as a tuple of (\mathbb{C}, \mathbb{L}) , in which \mathbb{C} is its type and \mathbb{L} is its heap locations. \mathbb{C} is either the class of the instance, or a generic array type, which means that we do not distinguish array element types. A variable that can appear in symbolic input and output (actually as a fresh symbolic value) is represented by a *location* θ , which may be a *named location* or a *heap location* of an object. A named location is either the receiver or a method parameter of the entry method of the symbolic execution. A heap location is either an *object field* or *array element* and denoted by (o, α) , in which o is the reference of the object, α is either a field f or a symbolic value i representing the array index.

AOTES organizes symbolic values into a *value graph*. A node in the value graph represents a symbolic value, and is a tuple of (t, P, a) , in which t is the *type* of the node, P is a set of predecessors, and a is an optional type-specific attribute associated with this node. There are six types of nodes.

1. *Constant*: $(\text{CONST}, \emptyset, c)$, where c is the constant literal in an `ldc` instruction.
2. *Reference*: (REF, \emptyset) . The heap is a mapping between reference values and objects. A `new` or `newarray` instruction allocates a new object and creates a reference value for the object to retrieve the object from the heap,
3. *Expression*: $(\text{EXPR}, \{v_1, v_2\}, op)$, where v_1 and v_2 are two operand values and op is a binary operator, i.e., one of `+`, `-`, `*`, `/`, and `%`.
4. *Assertion*: $(\text{ASSERT}, \{v_1, v_2\}, op)$, where v_1 and v_2 are two operand values and op is a relational operator, i.e., one of `>`, `>=`, `==`, `!=`, `<` and `<=`. An *opposite* operator (e.g., `<=` for `ifgt`) is used when a `false` branch is taken.
5. *Input*: $(\text{INPUT}, \emptyset, \theta)$, where θ is a location in the pre-heap or a method parameter.
6. *Output*: $(\text{OUTPUT}, \{v_\theta\}, \theta)$, where θ is a location in the post-heap and v_θ is the value of θ .

Figure 9 summarizes the effects of each bytecode instructions in terms of the modification of a *configuration*. A configuration is a reflection of the runtime of a running Java program, and is composed of the following components, denoted as $(\mathcal{F}, \Pi, \Phi, \Sigma)$ for brevity.

- \mathcal{F} , the stack for method frames.
- Π , the symbolic heap, a mapping from values to objects.
- Φ , the path condition, actually a sequence of `ASSERT`.
- Σ , the symbolic state, a mapping from variable (locations) to values (nodes).

■ **Table 2** Symbols used in the rules.

Symbol	Description
σ, σ'	a configuration $(\mathcal{F}, \Pi, \Phi, \Sigma)$
$\langle \text{ldc } c, \sigma \rangle \Rightarrow \sigma'$	a rule for the instruction <code>ldc c</code>
v, o, i	a generic value, a reference value and an index value, respectively
$\Pi[[o]]$	obtain the object referenced by o
$\Pi[[o = (C, L)]]$	update the heap and make o reference the object (C, L)
$\Sigma[[o, i]]$	read the value of the location (o, i)
$\Sigma[[o, i] = v]$	update the value of the location (o, i)
$\mathcal{F} \cdot (m, pc, \mathcal{L}, \mathcal{E})$	push a frame $(m, pc, \mathcal{L}, \mathcal{E})$ to the method frame stack \mathcal{F}
$\mathcal{E} \cdot v$	push a value v into the expression stack

A method frame is denoted by a tuple $(m, pc, \mathcal{L}, \mathcal{E})$ of the method m , the current bytecode index pc , the local variables array \mathcal{L} , and the expression stack \mathcal{E} for bytecode instructions [29]. Since most instructions are intra-procedure, we ignore the method m and a configuration is also denoted by a sextuple $(pc, \mathcal{L}, \mathcal{E}, \Pi, \Phi, \Sigma)$.

Rules in Figure 9 actually define an structural operational semantics [35] of each bytecode instruction over the node (t, P, a) . The detailed semantics of each bytecode instruction can be found in [29]. Table 2 summarizes the symbols used in describing every rule.

Not all bytecode instructions produce values, e.g., an `invoke` only copies arguments from the caller to the callee. For the entry method, AOTES creates a `CONST` and allocates a pre-allocated object for its receiver, and creates an `INPUT` for each method parameter of it. `INPUT` in pre-allocated objects are created when first used. At the end of a *normally terminated* execution, AOTES creates an `OUTPUT` for every heap location in objects reachable from the receiver. Exceptional executions are abandoned.

Note that in symbolic execution, which branch (i.e., `true` or `false`) is taken is not determined by evaluating the condition to a concrete value but by a strategy. AOTES takes a random strategy to explore branches and collect path conditions. First, it randomly takes an *unvisited* branch. After all branches have been visited, it then randomly takes a *visited* branch. For any method, we only collect a path condition of a limited length. Loop and recursion are discussed in detail in Section 4.4.

Finally, we create a value graph to summarize an execution. There are two kinds of edges between nodes, representing *value dependency* or *location dependency*. The location dependency tracks values in heap locations of `INPUT` and `OUTPUT` (e.g., object reference and array index), and is used to align the symbolic post-heap to a concrete heap. The value graph is constructed as follows. Initially, the value graph is empty. Then, all `INPUT`, `OUTPUT`, `ASSERT` are first added to the value graph. Other nodes are recursively added by following the two kinds of dependency edges.

Figure 10 depicts three value graphs of `addListener` in Figure 1a. Note that we simplify the implementation of method `add` of class `ArrayList` for brevity but AOTES can handle the actual one. Lets take the left-most graph as an example to illustrate the semantics of a value graph. The value graph contains the following nodes and edges.

- Two `CONST` nodes w.r.t. `this` and `null`.
- Two `INPUT` nodes w.r.t. `firstListener` and the parameter.
- An `OUTPUT` w.r.t. `firstListener`. This `OUTPUT` has a location dependency edge from `this` (`CONST`) and a value dependency edge from the parameter (`INPUT`).

$$\begin{array}{c}
\text{Stack \& Locals} \\
\langle \text{ldc } c, (pc, \mathcal{L}, \mathcal{E}, \Pi, \Phi, \Sigma) \rangle \Rightarrow (pc + 1, \mathcal{L}, \mathcal{E} \cdot (\text{CONST}, \emptyset, c), \Pi, \Phi, \Sigma) \\
\langle \text{load } i, (pc, \mathcal{L}, \mathcal{E}, \Pi, \Phi, \Sigma) \rangle \Rightarrow (pc + 1, \mathcal{L}, \mathcal{E} \cdot \mathcal{L}[[i]], \Pi, \Phi, \Sigma) \\
\langle \text{store } i, (pc, \mathcal{L}, \mathcal{E} \cdot v, \Pi, \Phi, \Sigma) \rangle \Rightarrow (pc + 1, \mathcal{L}[[i=v]], \mathcal{E}, \Pi, \Phi, \Sigma) \\
\text{Array Access} \\
\langle \text{aload}, (pc, \mathcal{L}, \mathcal{E} \cdot o \cdot i, \Pi, \Phi, \Sigma) \rangle \Rightarrow (pc + 1, \mathcal{L}, \mathcal{E} \cdot \Sigma[[o, i]], \Pi, \Phi, \Sigma) \\
\langle \text{astore}, (pc, \mathcal{L}, \mathcal{E} \cdot o \cdot i \cdot v, \Pi, \Phi, \Sigma) \rangle \Rightarrow (pc + 1, \mathcal{L}, \mathcal{E}, \Pi, \Phi, \Sigma[[o, i]=v]) \\
\text{Field Access} \\
\langle \text{getfield } f, (pc, \mathcal{L}, \mathcal{E} \cdot o, \Pi, \Phi, \Sigma) \rangle \Rightarrow (pc + 1, \mathcal{L}, \mathcal{E} \cdot \Sigma[[o, f]], \Pi, \Phi, \Sigma) \\
\langle \text{putfield } f, (pc, \mathcal{L}, \mathcal{E} \cdot o \cdot v, \Pi, \Phi, \Sigma) \rangle \Rightarrow (pc + 1, \mathcal{L}, \mathcal{E}, \Pi, \Phi, \Sigma[[o, f]=v]) \\
\text{Allocation} \\
\langle \text{new } \mathbb{C}, (pc, \mathcal{L}, \mathcal{E}, \Pi, \Phi, \Sigma) \rangle \Rightarrow (pc + 1, \mathcal{L}, \mathcal{E} \cdot o, \Pi[[o=(\mathbb{C}, \mathbb{L})]], \Phi, \Sigma) \wedge o \leftarrow (\text{REF}, \emptyset) \\
\langle \text{newarray}, (pc, \mathcal{L}, \mathcal{E} \cdot v, \Pi, \Phi, \Sigma) \rangle \Rightarrow (pc + 1, \mathcal{L}, \mathcal{E} \cdot o, \Pi[[o=(\mathbb{A}, \mathbb{L})]], \Phi, \Sigma[[o, \iota]=v]) \wedge o \leftarrow (\text{REF}, \emptyset) \\
\text{Binary Operator} \\
\langle \text{add}, (pc, \mathcal{L}, \mathcal{E} \cdot v_1 \cdot v_2, \Pi, \Phi, \Sigma) \rangle \Rightarrow (pc + 1, \mathcal{L}, \mathcal{E} \cdot (\text{EXPR}, \{v_1, v_2\}, +), \Pi, \Phi, \Sigma) \\
\text{Branch} \\
\langle \text{ifgt } \rho, (pc, \mathcal{L}, \mathcal{E} \cdot v_1 \cdot v_2, \Pi, \Phi, \Sigma) \rangle \Rightarrow (pc + \rho, \mathcal{L}, \mathcal{E}, \Pi, \Phi \cup (\text{ASSERT}, \{v_1, v_2\}, >), \Sigma), \text{ if true} \\
\langle \text{ifgt } \rho, (pc, \mathcal{L}, \mathcal{E} \cdot v_1 \cdot v_2, \Pi, \Phi, \Sigma) \rangle \Rightarrow (pc + 1, \mathcal{L}, \mathcal{E}, \Pi, \Phi \cup (\text{ASSERT}, \{v_1, v_2\}, <=), \Sigma), \text{ if false} \\
\langle \text{goto } \rho, (pc, \mathcal{L}, \mathcal{E}, \Pi, \Phi, \Sigma) \rangle \Rightarrow (pc + \rho, \mathcal{L}, \mathcal{E}, \Pi, \Phi, \Sigma) \\
\text{Invoke \& Return} \\
\langle \text{invoke } m, (\mathcal{F} \cdot (m', pc', \mathcal{L}', \mathcal{E}' \cdot o \cdot v_1 \cdots v_n), \Pi, \Phi, \Sigma) \rangle \Rightarrow (\mathcal{F} \cdot (m, 0, \mathcal{L}[[0=o]][[1=v_1]] \cdots [[n=v_n]], \emptyset), \Pi, \Phi, \Sigma) \\
\langle \text{return}, (\mathcal{F} \cdot (m', pc', \mathcal{L}', \mathcal{E}' \cdot o \cdot v_1 \cdots v_n) \cdot (m, pc, \mathcal{L}, \mathcal{E} \cdot v), \Pi, \Phi, \Sigma) \rangle \Rightarrow (\mathcal{F} \cdot (m', pc', \mathcal{L}', \mathcal{E}' \cdot v), \Pi, \Phi, \Sigma)
\end{array}$$

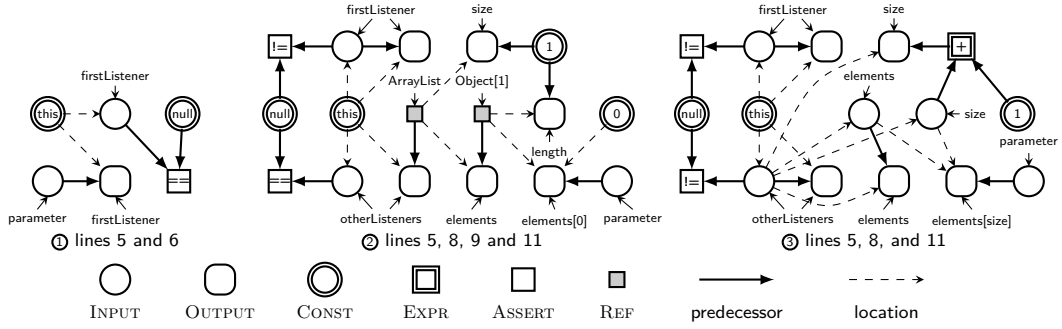
■ **Figure 9** Rules describing effects of bytecode instructions. Each rule is in the format $\langle \text{inst}, \sigma \rangle \Rightarrow \sigma'$, where **inst** is a bytecode instruction, σ and σ' are the configurations before and after execution the instruction, respectively.

- An ASSERT w.r.t. the if statement at line 5 in Figure 1a. This ASSERT has value dependency edges from `firstListener` (INPUT) and `null` (CONST).

AOTES aims at deriving values for the two INPUT nodes from a given concrete object. The derivation is conducted by traversing the graph. First, AOTES derives the concrete value for OUTPUT by loading `firstListener` after aligning `this` to the concrete object. Next, AOTES derives the value for the parameter (INPUT) using the value of `firstListener` (OUTPUT) directly. Note that we need to check whether the ASSERT satisfies. The INPUT (w.r.t. `firstListener`) has been overridden during forward execution. We then create a fresh symbolic value for the INPUT and try to use the constraint solver to derive a value for it. All deriving steps are serialized into a method to facilitate the online execution synthesis, which will be discussed in the next subsection.

4.3 Inverse Method Synthesis

We say that a node is *resolved* if its concrete value has been derived. An inverse method is created by resolving all INPUT. An OUTPUT can be directly resolved if its symbolic location can be *aligned* to a concrete location. `this` can be directly aligned to the receiver. An object field can be aligned if its object reference is resolved. Thus, all fields accessed from `this` can be aligned. An array element can be aligned if both the object reference and index are resolved. AOTES translates every resolution and alignment into a statement. All statements finally make up the inverse method. AOTES supports four kinds of resolution methods.



■ **Figure 10** Value graphs of three execution traces of `addListener` in Figure 1a. Their inverse methods are in Figure 8.

1. *Direct Resolution*: All CONST and aligned OUTPUT can be directly resolved.
2. *Forward Resolution*: A node is resolved if all predecessors are resolved. For example, if $a = b \times c$, and b and c are resolved, then a can also be resolved by evaluating $b \times c$ again.
3. *Backward Resolution*: If an EXPR and one of its predecessors are resolved, we can resolve the other predecessor by these two nodes. For example, if $a = b - c$, and a and b are resolved, then c can be resolved by evaluating $b - a$. We treat $+$, $-$, $*$, and $/$ *invertible* due to the aggressive nature of AOTES.
4. *Aggressive Resolution*: As an inverse method is used for execution synthesis, we can aggressively guess a value for an INPUT by assigning a fresh symbolic value to it. Besides, we can guess an index for an array element if its object reference has been resolved.

Algorithm 1 aims at resolving all nodes of a value graph. The algorithm maintains a sequence of statements \bar{m} , and two sets of nodes, R and U , i.e., the sets of resolved and unresolved nodes, respectively. At first, R and \bar{m} are empty, and U contains all nodes in the value graph. The four resolution methods try to apply rules defined in Figure 11 and return `true` if there is an applicable rule, which means some nodes have been resolved. Every successful resolution appends a statement to \bar{m} (denoted by \uplus). In theory, we can continue to apply aggressive resolution to resolve every INPUT and then use forward resolution to resolve all unresolved nodes in the value graph. The algorithm can finally terminate when R is fixed, since a value can never be moved from R to U . \bar{m} is successfully generated only if U is empty. We then decorate \bar{m} into a valid Java method. This method has no parameter and returns all reverted arguments.

Every node is indeed converted into a variable with a unique name. Every resolved INPUT must be aligned first and its concrete location is also updated with the resolved value. For presentation, this requirement is not expressed in the rules. A fresh symbolic value is denoted by $*$ but in fact produced by a runtime method. We also provide a runtime method `guess` that chooses an index in a given array.

Figure 11 presents rules that are used to resolve a node. A rule takes a node from the value graph and the currently visiting status (i.e., the tuple (R, U, \bar{m})) as input to update the visiting status for next visiting and produce a statement for the inverse method \bar{m} as output. Each rule has a precondition that should be checked first. Basically, the precondition at least ensures that each node is resolved once by a rule. Take rules of direct resolution as an example. To resolve a CONST, the rule only checks whether the node being resolved has been resolved. To resolve an OUTPUT, the two rules further check whether the location has been aligned.

Algorithm 1: Resolution of a value graph.

Input: (V, E) , the value graph.
Output: (R, U, \bar{m}) , where R is the set of resolved nodes, U is the set of unresolved nodes, and \bar{m} is the inverse method.

```

1  $(R, U, \bar{m}) \leftarrow (\emptyset, V, \emptyset)$  foreach  $v \in U$  do DIRECT( $v, (R, U, \bar{m})$ )
2 repeat
3   repeat
4     foreach  $v \in U$  do FORWARD( $v, (R, U, \bar{m})$ )
5     foreach  $v \in R$  do BACKWARD( $v, (R, U, \bar{m})$ )
6   until  $R$  is fixed
7   foreach  $v \in U$  do
8     if AGGRESSIVE( $v, (R, U, \bar{m})$ ) then break
9 until  $R$  is fixed
10 return  $(R, U, \bar{m})$ 

```

Figure 8 shows the inverse methods created by resolving nodes from value graphs in Figure 10. We have simplified the output, e.g., remove redundant variables for `CONST`. Algorithm 1 and rules in Figure 11 ensure that a node is only resolved once. We randomly visit nodes and thereby, a node can be resolved via different rules and nodes. For consistency, AOTES attempts to resolve every node using a different method at last and adds assertions to ensure that all resolved concrete values must be equal, e.g., lines 15 and 16.

4.4 Loop and Recursion

AOTES takes a single-path symbolic execution and limits the length of the path condition. Hence, the loop and recursive method invocations are unrolled for a limited length. A set of short execution paths is considered as the promising candidates for online execution synthesis. Obviously, there do exist long-running loops and deep recursions. Thus, the symbolic execution trace may be infeasible for some inputs (pre-heaps).

Actually, the problem of loop and recursion may not be as critical as it seems to be. Recall that AOTES has no need to synthesize an inverse method for all execution traces. Besides, a fixed number of inverse methods are sufficient sometimes. In comparison with existing whole program execution techniques [44, 5], the insight of invocation history synthesis is that it infers the sequence from the state only and requires no complete control flow and call graph. Moreover, many loops and recursive methods are guided by some input [43]. AOTES can split a loop with a very large input in the actual history into many loops with a small input in the synthesized history.

Take the class in Figure 12 as an example. Method `addN` has a loop and also recursively calls itself. AOTES can easily populate the execution trace where `n` is 1 and also synthesize an inverse method for it, because `addN(a, 1)` is equivalent to `elements.add(a)`. We have shown that AOTES can easily handle `ArrayList`. Hence, no matter how divergent the actual history is, AOTES can always guarantee to synthesize a history `addN(e0, 1), ..., addN(ei, 1), ..., addN(ek-1, 1)`, where `ei` is the *i*-th element in `elements` and *k* is the size of the list `elements`. For example, an actual history composed of an `addN([a, b], 2)` would be replaced by the following synthesized history: `addN(a, 1), addN(a, 1), addN(b, 1), addN(b, 1)`, where both of them fill the `ArrayList` referred to by `elements` with the sequence `[a, a, b, b]`.

5 Execution Synthesis

This section depicts the online synthesis and replaying of invocation histories.

$$\begin{array}{c}
\text{Direct Resolution} \\
\frac{v \notin R}{\langle v = (\text{CONST}, \emptyset, c), (R, U, \overline{m}) \rangle \Rightarrow (R \cup \{v\}, U \setminus \{v\}, \overline{m} \uplus v = c;)} \\
\frac{v \notin R \wedge \theta = (o, i) \wedge o \in R \wedge i \in R}{\langle v = (\text{OUTPUT}, \{v_\theta\}, \theta), (R, U, \overline{m}) \rangle \Rightarrow (R \cup \{v\}, U \setminus \{v\}, \overline{m} \uplus v = o[i];)} \\
\frac{v \notin R \wedge \theta = (o, f) \wedge o \in R}{\langle v = (\text{OUTPUT}, \{v_\theta\}, \theta), (R, U, \overline{m}) \rangle \Rightarrow (R \cup \{v\}, U \setminus \{v\}, \overline{m} \uplus v = o.f;)} \\
\text{Forward Resolution} \\
\frac{v \notin R \wedge v_1 \in R \wedge v_2 \in R}{\langle v = (\text{EXPR}, v_1, v_2, +), (R, U, \overline{m}) \rangle \Rightarrow (R \cup \{v\}, U \setminus \{v\}, \overline{m} \uplus v = v_1 + v_2;)} \\
\frac{v \notin R \wedge v_1 \in R \wedge v_2 \in R}{\langle v = (\text{ASSERT}, v_1, v_2, >), (R, U, \overline{m}) \rangle \Rightarrow (R \cup \{v\}, U \setminus \{v\}, \overline{m} \uplus \text{assert}(v_1 > v_2);)} \\
\text{Backward Resolution} \\
\frac{v_1 \notin R \wedge v \in R}{\langle v = (\text{OUTPUT}, \{v_1\}, \theta), (R, U, \overline{m}) \rangle \Rightarrow (R \cup \{v_1\}, U \setminus \{v_1\}, \overline{m} \uplus v_1 = v;)} \\
\frac{v_2 \notin R \wedge v \in R \wedge v_1 \in R}{\langle v = (\text{EXPR}, v_1, v_2, +), (R, U, \overline{m}) \rangle \Rightarrow (R \cup \{v_2\}, U \setminus \{v_2\}, \overline{m} \uplus v_2 = v - v_1)} \\
\text{Aggressive Resolution} \\
\frac{\theta = (o, i) \wedge o \in R \wedge i \notin R}{\langle i, (R, U, \overline{m}) \rangle \Rightarrow (R \cup \{i\}, U \setminus \{i\}, \overline{m} \uplus i = \text{guess}(o);)} \\
\frac{v \notin R \wedge \Pi_{pre} \llbracket v \rrbracket = \emptyset \wedge \theta = (o, i) \wedge o \in R \wedge i \in R}{\langle v = (\text{INPUT}, \emptyset, \theta), (R, U, \overline{m}) \rangle \Rightarrow (R \cup \{v\}, U \setminus \{v\}, \overline{m} \uplus v = o[i] = *;)} \\
\frac{v \notin R \wedge \Pi_{pre} \llbracket v \rrbracket = \emptyset \wedge \theta = (o, f) \wedge o \in R}{\langle v = (\text{INPUT}, \emptyset, \theta), (R, U, \overline{m}) \rangle \Rightarrow (R \cup \{v\}, U \setminus \{v\}, \overline{m} \uplus v = o.f = *;)}
\end{array}$$

■ **Figure 11** Rules for resolving nodes and generating statements. Each rule is in the format $\frac{P}{\langle v, (R, U, \overline{m}) \rangle \Rightarrow \langle R', U', \overline{m}' \rangle}$, where P is the precondition of applying the rule, v is the node that we attempt to resolve, R is the set of resolved nodes, U is the set of unresolved nodes, \overline{m} is the sequence of generated statements, and R' , U' and \overline{m}' are new versions after applying the rule.

5.1 Online Synthesis of Invocation Histories

AOTES uses a greedy strategy to search for an inverse invocation history. As shown in Algorithm 2, AOTES first collects all applicable inverse method invocation (\mathcal{T}), and uses a heuristic method to rank them (by function RANK). Intuitively, a better inverse method should revert more INPUT in the Π_{pre} from non-default values to default values and preserve more locations in the Π_{pre} , which is the Π_{post} for the next step. Hence, AOTES prefers the inverse method with no aggressively resolved INPUT first, then more live locations after execution, and finally more reverted INPUT. The searching stops when the object is in the empty state or there is no applicable inverse method. AOTES executes an inverse method in two ways, i.e., TESTAPPLY, which will restore the object state for applying next inverse method, and APPLY, which will retain the modification.

For example, suppose that an object of `DefaultSshFuture` is in state s_3^1 . `addListener2`¹ is not applicable as line 16 in Figure 8 fails, i.e., the `size` is not 2. Both `addListener1`¹ and `addListener3`¹ are applicable, but we prefer `addListener3`¹ over `addListener1`¹ as it reverts more locations and also preserves `otherListeners`. The object state then becomes s_2^1 . `addListener2`¹ is still not applicable on s_2^1 as line 15 fails, i.e., the array has been expanded and its length is not 1. We then prefer `addListener3`¹ for the same reason and apply it to obtain s_1^1 . Now, only `addListener1`¹ is applicable on s_1^1 . `addListener3`¹ is inapplicable on s_1^1 as line 28 attempts to revert `size` from 0 to -1. At last, there is no applicable inverse method and the synthesis terminates.


```

1 class LoopAndRecursion {
2   List elements = new ArrayList();
3   void addN(Object o, int n) {
4     if (n < 1) {
5       return;
6     } else if (o instanceof Object[]) {
7       for (Object e : (Object[]) o) {
8         addN(e, n);
9       }
10    } else {
11      elements.add(o);
12      addN(o, n-1);
13    }}

```

■ **Figure 12** An example of loop and recursion.

Algorithm 2: History synthesis.

Input: o , the receiver object for history synthesis, $\overline{\mathbb{M}}$, the set of inverse methods.
Output: \mathcal{H} , the invocation history for o .

```

1  $\overline{\mathcal{H}} \leftarrow \emptyset$ 
2 while isNOTEMPTYSTATE( $o$ ) do ▷ Stop once the object is reverted into the empty state.
3    $\mathcal{T} \leftarrow \emptyset$  ▷ The set of applicable inverse method invocation at this step.
4   foreach  $\overline{m} \in \overline{\mathbb{M}}$  do
5      $\mathbf{a} \leftarrow \text{TESTAPPLY}(\overline{m}, o)$  ▷ Apply  $\overline{m}$  to  $o$  and restore  $o$  afterwards.
6      $\mathcal{T} \leftarrow \mathcal{T} \cup \{(\overline{m}, \mathbf{a})\}$ 
7   if  $\mathcal{T} = \emptyset$  then
8     break ▷ Stop when there is no applicable inverse method.
9    $(\overline{m}, \mathbf{a}) \leftarrow \text{RANK}(\mathcal{T})$  ▷ Heuristic-based ranking.
10   $\overline{\mathcal{H}} \leftarrow \overline{\mathcal{H}} \cup \{(\overline{m}, \text{APPLY}(\overline{m}, o))\}$  ▷ Apply  $\overline{m}$  to  $o$  without restore.
11 return REVERT( $\overline{\mathcal{H}}$ ) ▷ Revert  $\overline{\mathcal{H}}$  and replace every  $\overline{m}$  by its original method  $m$ .

```

5.2 Realizing Object Transformations

AOTES realizes object transformations as follows. Given a stale object, we first try to synthesize an inverse invocation history for it. If the history is empty, then we fall back to default transformations. Otherwise, we apply a default transformation to the object after reverting its state by applying the inverse invocation history. Finally, we invert the inverse invocation history to build a new history and apply it to the object. Note that the synthesized history is not necessarily to be complete.

6 Implementation

We implemented AOTES, including the symbolic execution engine, inverse method synthesizer and invocation history synthesizer, in about 25K lines of Java code. AOTES is fully automated and only takes binary class files as input and thus requires no source code, no test case and no human specified update points.

We implemented a trivial single-variable solver. For example, a fresh symbolic value for `int` includes all integers in `[MIN_INT, MAX_INT]`. As such a symbolic value is mostly used in assertions and pre-states. Therefore, it narrows its range towards passing the assertion and to the default value during evaluation. For example, suppose that a variable `v1` has a fresh symbolic value for `int`. After evaluating the assertion `assert(v1 == 0)`, its range is narrowed to `[0, 0]`, which means that this symbolic value can only be 0. Currently, we are working on integrating Z3 [9] as the solver to further improve the effectiveness of AOTES.

The main limitation of AOTES in analyzing real-world applications is uninterpreted native methods. The current implementation of AOTES only handles a small part of native methods that we have encountered during evaluation, among which some are re-implemented using Java, e.g., `arraycopy`, and others are manually marked as operators, e.g., `sin` and `identityHashCode`. Operator methods are not interpreted during symbolic execution and their effects are recorded like an operator (i.e., creating an `EXPR`). During synthesis, they can be executed as all arguments are available at present. We allocate a phantom object for every class as the container for static fields. The reference of the phantom object is treated as a `CONST` and thus a static field can be easily aligned.

7 Experiments

We evaluated AOTES's effectiveness with real-world updates and performance in synthesizing long histories using a micro benchmark, respectively. All experiments were conducted on an Intel Core i7 3.4GHz machine with 20 GB memory running 64-bit Windows 10. The offline synthesis was conducted on JDK 1.8.0_65 and the dynamic updating was carried out on Javelus. We forced AOTES to only explore at most 20 different traces for a method and 1000 branches for a trace.

7.1 Real-world Updates

We collected 21 updated classes from Apache Commons Collections, Apache FTP Server, Apache SSHD Server and Apache Tomcat, which are all widely used common libraries and server applications under years of active development. These updates were chosen for the following reasons. First, the two versions of all updates must be successfully compiled. Second, all updates *must* involve field changes otherwise would require no transformation. We classified these fields changes into the following four types:

1. VALUE CHANGE: with no field added, but the values of some fields need to be updated.
2. NAME CHANGE: with a field renamed only.³
3. TYPE CHANGE: with a type-changed field only.
4. COMPLEX CHANGE: any other changes.

Third, an updated class must not invoke uninterpreted native methods beyond those handled by the current implementation of AOTES. Finally, we also excluded rare cases in which the stale state does not contain sufficient information to determine the new state. In this situation, even a programmer may not be able to provide a transformer based on the stale state without additional information, not to mention TOS or AOTES.

In addition to existing test cases, we additionally wrote a few test cases for some updated classes under our test frame work designed for DSU, because most updated classes have no test case and some existing test cases were insufficient to detect improper object transformations. Every test created an object with one or a few method invocations before dynamic updating. Then, we triggered the dynamic updating and applied the transformation to the object. Every dynamic update was verified as follows [33, 30]. That is, the state after dynamic updating of the old version must be equivalent to a state that can be achieved by executing the same methods on the new version.

³ Note that if either the name or type of a field is changed, it is considered as deleted and a new field with the new name or type is added.

■ **Table 3** Results of real-world updates.

Type	Update	Tests	AOTES	Default	TOS
VALUE CHANGE	tomcat-dd741c	6	4 ^{α)}	4	N.A.
NAME CHANGE	ftp-5d5592	4	3 ^{α)}	0	N.A.
	sshd-6f8507	2	2	1	N.A.
	tomcat-951e08	2	2	1	N.A.
	tomcat-b75f5c	2	2	1	N.A.
TYPE CHANGE	collections-0e1140	1	1	0	N.A.
	collections-cdacd4	1	1	0	N.A.
	ftp-f8110b	5	5	0	N.A.
	sshd-054334	3	3	1	N.A.
	tomcat-480edc	3	3	0	N.A.
COMPLEX CHANGE	collections-b7327a	2	2	0	N.A.
	ftp-43ff5f	4	2 ^{β)}	0	N.A.
	ftp-4907aa	4	2 ^{β)}	0	N.A.
	ftp-5df186	4	2 ^{β)}	0	N.A.
	sshd-009d83	5	5	0	N.A.
	sshd-1487b0	1	1	0	1
	sshd-2297b2	1	1	0	1
	sshd-b98694	7	7	2	2
	sshd-eeeeec6	1	0 ^{α)}	0	1
	tomcat-24bc4d	2	2	1	1
	tomcat-2db0f7	1	1	0	N.A.
Total		61	51/83.6%	11/18.3%	6/9.8%

a) α and β indicates inconsistent and incomplete synthesized history, respectively.

We ran all tests with dynamic updating for both AOTES and default transformations on Javelus. All results are shown in Table 3. AOTES succeeded in 51 (83.6%) updates and failed in other 10 updates due to *incomplete* or *inconsistent* synthesized histories. An incomplete history cannot update all fields as the searching also stops when no applicable inverse method is found. This is mainly because many native methods prevent AOTES from generating sufficient inverse methods. We plan to model more native methods in future. An inconsistent history leads to the same state as the actual history in the old version but different states in the new version. We will discuss this limitation with examples in the following paragraphs.

We did not run TOS with dynamic updating as TOS is not fully automated and requires extra training tests and manually specified update points. Instead, we used our validation tests to train TOS and hope that it could synthesize a conditional transformer for each update that can realize transformations for all test cases. TOS failed to synthesize a function for 16 of 21 updates (marked with N.A. in Table 3). For the rest 5 updates with 12 tests in total, TOS even failed in validating its output against 6 training tests.

As shown in Table 3, almost all updates have field name changes or type changes. These updates are the majority of updates that require transformations in practice. Default transformations and TOS failed to derive a valid transformation/transformer even for many name and type changes because they cannot find the relations between fields with different names or types. AOTES can infer their relations when changed fields used the same arguments in matched methods. Moreover, both default transformations and TOS used a set of predefined simple rules, and cannot handle transformations involving custom type conversions (e.g., `ArrayList` to `ConcurrentHashMap`). AOTES leveraged program code to infer custom type conversions when objects of different types used the same arguments in matched methods for initialization.

We discuss the limitation and effectiveness of AOTES with the following four examples. AOTES failed in the first two examples and succeeded in the last two examples.

7.1.1 Value Change: Tomcat dd741c

```
1 - private String jmxNameBase = "pool";
2 + private String jmxNameBase = null;
```

This update only changes the initial value of `jmxNameBase` in the constructor. If the setter of `jmxNameBase` is not invoked, the transformation should update the value to `null`. AOTES failed in two test cases due to inconsistent histories. That is, both the constructor and the setter method can assign "pool" to `jmxNameBase` in the old version but `null` and "pool" in the new version. In fact, even a programmer cannot write a *general* transformer here because using the current state only cannot distinguish different actual histories that lead to the same current state.

7.1.2 Name Change: FTP 5d5592

```
1 - private int maxIdleTimeMillis = 10000;
2 + private int idleTime = 300;
3   public void setIdleTime(int idleTime) {
4     - maxIdleTimeMillis = idleTime * 1000;
5     + this.idleTime = idleTime;
6   }
```

Except this update, we can just copy the value from a new field to the old field for all NAME CHANGE updates. AOTES failed in the only test for the same reason as Tomcat dd741c. `maxIdleTimeMillis` was set to 10000 by the constructor in the old version but in the new version `idleTime` should be 300.

7.1.3 Type Change: FTP f8110b

```
1 class DefaultFtpletContainer {
2   - private List ftplets = new ArrayList();
3   - class FtpletEntry { String name; Ftplet ftplet; }
4   + private Map ftplets = new ConcurrentHashMap();
5   public void addFtplet(String name, Ftplet ftplet) {
6     - ftplets.add(new FtpletEntry(name, ftplet));
7     + ftplets.put(name, ftplet);
8   }
9 }
```

Type conversions between built-in types are easy, e.g., `int` to `long`. However, for this update, we need the key to convert an `ArrayList` to a `ConcurrentHashMap`. AOTES can synthesize inverse methods for `addFtplet` and the constructor and also a history using them. The key can be inferred from the parameter `name` of the new version of `addFtplet`.

7.1.4 Complex Change: SSHD 009d83

```
1 class AgentImpl {
2   private List keys = new ArrayList();
3   - private boolean closed;
4   + private AtomicBoolean open = new AtomicBoolean(true);
5   public void close() throws IOException {
6     - closed = true;
```

```

7 - keys.clear();
8 + if (open.getAndSet(false)) {
9 +   keys.clear();
10 + }
11 }
12 public void addIdentity(KeyPair key, String comment) {
13   keys.add(new Pair(key, comment));
14 }
15 }

```

This update changes both the type and the name of a field. Method `close` removes all elements in `keys`. Hence, the synthesized history is a constructor and a `close` if the last method in the actual history is a `close`. For example, suppose that the actual history includes a constructor of `AgentImpl`, an `addIdentity`, and a `close`. However, a critical field `modCount` in `ArrayList`, which is used to avoid concurrent modification during iterating the list, prevents AOTES from applying the inverse constructor of `AgentImpl` if its value cannot be reverted to 0. Fortunately, the inverse method of `clear` decrements `modCount`. As a result, the synthesized history is a constructor followed by two invocations of `close`.

7.2 Micro Benchmark

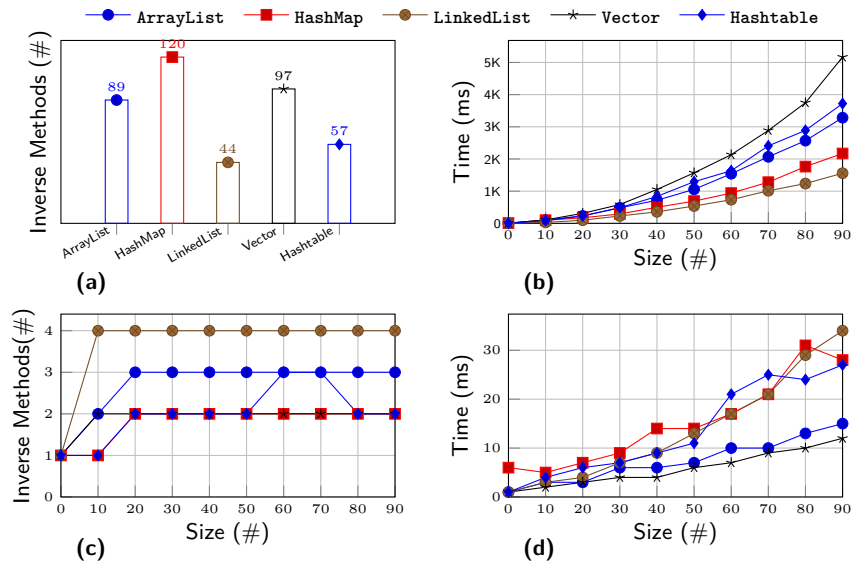
We selected five classes of commonly used collections and designed a micro benchmark to evaluate the synthesizing time of AOTES. Theoretically, the synthesizing time only directly depends on the current object state and the number of inverse methods but not the actual history. Thereby, we first conducted experiments with all synthesized inverse methods and then repeated the experiments with a small set of inverse methods. The results help us to reveal solutions that can optimize the synthesizing time of AOTES.

The micro benchmark created an object of each class, filled it with a number of elements (ranged from 0, 10, ..., 90), and finally synthesized a history for it. We also repeated the procedure for 10 times first to warm up the JVM. Figure 13a shows the distribution of inverse methods generated by AOTES for all *public* methods of every class. We first ran the benchmark on *all* inverse methods and then repeated the benchmark with *only previously used* inverse methods.

The synthesizing time using all inverse methods is shown in Figure 13b. The time depends on the size of elements in a collection. AOTES spent more than 5s in the worst case for `Vector`. This is mainly because our implementation heavily uses reflections and exceptions. Besides, most of the inverse methods of these classes were indeed redundant.

The number of inverse method candidates has an impact to the searching time. Figure 13c shows the number of *different inverse methods* (not *different inverse method invocations*) appeared in each history. No more than four inverse methods were actually used for all histories. That means the online synthesis wasted a certain amount of time on trying out redundant inverse methods. Note that here an inverse method may be invoked for many times. The actual number of method invocations in total were mostly the same as the number of elements.

Pruning redundant inverse methods can speed up the online history synthesis. In practice, we can prune redundant inverse methods using an automatic random testing tool [36]. Figure 13d shows the synthesizing time using only previously used inverse methods. AOTES only spent 35ms in the worst case for `LinkedList` and only 12ms for `Vector`. We believe that this synthesizing time is acceptable for practical usage and can also be further reduced with a more efficient implementation of AOTES.



■ **Figure 13** Results of micro benchmark.

7.3 Discussion

AOTES can be used as a complement to existing techniques such as default transformations, TOS, and manual approaches, especially for name changes, type changes and complex changes. While both default transformations and TOS cannot find the relations between old fields and new fields that have different names or types, AOTES can find the relations by matching the arguments in matched methods. Different from TOS, which requires test cases and manual efforts during collecting transformation examples, AOTES is *fully* automated and works purely on binaries without source code and test cases. AOTES is the only approach that can leverage the program code to infer powerful transformations. Moreover, it is an on-demand dynamic approach and can avoid synthesizing transformation that are hard to be automated but may not be encountered during dynamic updating.

The two assumptions of AOTES (Section 2.2) may be a threat. Techniques such as random testing [36] can help to reveal the violation of assumptions before updating. The time of online execution synthesis may be another threat to AOTES, particularly when there are many stale objects. AOTES tackles this challenge by adopting a lazy updating mechanism and sharing searching strategies across different transformations. Specifically, AOTES first mitigates the disruption caused by synthesis using a lazy updating technique, which is naturally supported by Javelus. Second, AOTES can try out methods that have already been used only. The effectiveness has been demonstrated in the micro benchmark. In other words, AOTES shares the searching strategy across different transformations, while TOS and manual approaches share the transformer.

8 Related Work

We survey related work in this section, including dynamic software updating, and program and execution synthesis.

8.1 Dynamic Software Updating

In general, DSU systems can be divided in two types, *intra-process state transformation* and *inter-process state transfer*. Many challenges in implementing inter-process state transfer have made it not so popular in building DSU systems [21, 12]. Not all programs can run multiple instances of multiple versions simultaneously, particularly in a production environment [21], e.g., the Linux kernel. In contrast, this approach has been extensively studied in live migration of virtual machines [6].

The majority of DSU systems apply transformations to the intra-process states. These approaches can generate default transformations and support programmers specified transformations as well [41, 13, 15]. Besides, they also provide a safety guarantee, e.g. *type safety*, to facilitate developing transformers [23, 34, 4, 32, 41, 14]. The transformations can be taken eagerly [41, 42], or lazily [14, 15], or both [38]. Although the size of a valid transformer may not be great [2], it should be delivered with extremely timing constraints, e.g., for security patches.

Automated approaches such as TOS [31] and TTST [12] require a pair of matched objects as example and infer transformations from these examples. AOTES requires no example as it uses matched methods, which makes it able to handle non-trivial cases that TTST and TOS cannot handle. Other approaches for debugging prefer no user intervention by sacrificing the flexibility or validity [11, 24]. Gupta et al. have proved that the validity of general dynamic updating is undecidable [17]. Besides, existing programming techniques can also help transformer programming, e.g., formalization and verification [20, 26, 45] and software testing [19, 22].

8.2 Program and Execution Synthesis

Program Synthesis and *Execution Synthesis* [44] have been extensively studied for years. Among them most related to AOTES are inverse program generation [10, 40] and data transformations [16, 18, 27, 39]. AOTES combines the program synthesis and execution synthesis. AOTES indeed makes use of reverse execution [3, 7, 1] over *symbolic execution traces* to generate an inverse program.

9 Conclusion

AOTES is an experimental approach to automating object transformations for dynamic software updating. It preserves the continuity of stateful behavior of objects whose classes are changed at runtime. The novelty of AOTES is to synthesize a method invocation history that can produce the current object state in the old version, and replay the history to get the desired state for the new version. Our preliminary evaluation shows that AOTES has the promising ability to handle software updates taken from real-world software systems. Although the current implementation of AOTES is for Java only, we believe that the general idea of AOTES can also apply to other object-oriented programming languages. In the future, we plan to improve AOTES by supporting more native methods and searching strategies, and also conduct a thorough evaluation of AOTES with more real-world updates.

References

- 1 Tankut Akgul and Vincent J. Mooney III. Assembly instruction level reverse execution for debugging. *ACM Transaction Software Engineering Methodology*, 13(2):149–198, 2004.
- 2 Jeff Arnold and M. Frans Kaashoek. Ksplice: Automatic rebootless kernel updates. In *Proceedings of the 4th ACM European Conference on Computer Systems*, pages 187–198, 2009.
- 3 Bitan Biswas and R. Mall. Reverse execution of programs. *SIGPLAN Notices*, 34(4):61–69, 1999.
- 4 Haibo Chen, Jie Yu, Rong Chen, Binyu Zang, and Pen-Chung Yew. POLUS: A powerful live updating system. In *Proceedings of the 29th International Conference on Software Engineering*, pages 271–281, 2007.
- 5 N. Chen and S. Kim. STAR: Stack Trace Based Automatic Crash Reproduction via Symbolic Execution. *IEEE Transactions on Software Engineering*, 41(2):198–220, 2015.
- 6 Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*, pages 273–286, 2005.
- 7 Jonathan J. Cook. Reverse execution of Java bytecode. *The Computer Journal*, 45(6):608–619, 2002.
- 8 Robert C. Daley and Jack B. Dennis. Virtual memory, processes, and sharing in MULTICS. *Communications of the ACM*, 11(5):306–312, 1968.
- 9 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*, pages 337–340, 2008.
- 10 Edsger W. Dijkstra. Program inversion. In *Program Construction*, volume 69, pages 54–57. Springer-Verlag, 1979.
- 11 Mikhail Dmitriev. Towards flexible and safe technology for runtime evolution of Java language applications. In *Proceedings of the Workshop on Engineering Complex Object-Oriented Systems for Evolution*, 2001.
- 12 Cristiano Giuffrida, Calin Iorgulescu, Anton Kuijsten, and Andrew S. Tanenbaum. Back to the future: Fault-tolerant live update with time-traveling state transfer. In *Proceedings of the 27th Large Installation System Administration Conference*, pages 89–104, 2013.
- 13 Cristiano Giuffrida, Anton Kuijsten, and Andrew S. Tanenbaum. Safe and automatic live update for operating systems. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 279–292, 2013.
- 14 Tianxiao Gu, Chun Cao, Chang Xu, Xiaoxing Ma, Linghao Zhang, and Jian Lu. Javelus: A low disruptive approach to dynamic software updates. In *Proceedings of 19th the Asia-Pacific Software Engineering Conference*, pages 527–536, 2012.
- 15 Tianxiao Gu, Chun Cao, Chang Xu, Xiaoxing Ma, Linghao Zhang, and Jian Lü. Low-disruptive dynamic updating of Java applications. *Information and Software Technology*, 56(9):1086–1098, 2014.
- 16 Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 317–330, 2011.
- 17 Deepak Gupta, Pankaj Jalote, and Gautam Barua. A formal framework for on-line software version change. *IEEE Transactions on Software Engineering*, 22(2):120–131, 1996.
- 18 William R. Harris and Sumit Gulwani. Spreadsheet table transformations from examples. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 317–328, 2011.

- 19 Christopher M. Hayden, Eric A. Hardisty, Michael Hicks, and Jeffrey S. Foster. Efficient systematic testing for dynamically updatable software. In *Proceedings of the 2nd International Workshop on Hot Topics in Software Upgrades*, pages 9:1–9:5, 2009.
- 20 Christopher M. Hayden, Stephen Magill, Michael Hicks, Nate Foster, and Jeffrey S. Foster. Specifying and verifying the correctness of dynamic software updates. In *Proceedings of the 4th International Conference on Verified Software: Theories, Tools, Experiments*, pages 278–293, 2012.
- 21 Christopher M. Hayden, Edward K. Smith, Michael Hicks, and Jeffrey S. Foster. State transfer for clear and efficient runtime updates. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering Workshops*, pages 179–184, 2011.
- 22 C.M. Hayden, E.K. Smith, E.A. Hardisty, M. Hicks, and J.S. Foster. Evaluating dynamic software update safety using systematic testing. *IEEE Transactions on Software Engineering*, 38(6):1340–1354, 2012.
- 23 Michael Hicks and Scott Nettles. Dynamic software updating. *ACM Transactions on Programming Languages and Systems*, 27(6):1049–1096, 2005.
- 24 Jevgeni Kabanov and Varmo Vene. A thousand years of productivity: the JRebel story. *Software: Practice and Experience*, 2012.
- 25 James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.
- 26 V.P. La Manna, J. Greenyer, C. Ghezzi, and C. Brenner. Formalizing correctness criteria of dynamic updates derived from specification changes. In *Proceedings of the 2013 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, pages 63–72, 2013.
- 27 Vu Le and Sumit Gulwani. FlashExtract: A framework for data extraction by examples. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 542–553, 2014.
- 28 Sheng Liang and Gilad Bracha. Dynamic class loading in the Java virtual machine. In *Proceedings of the ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 36–44, 1998.
- 29 Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java Virtual Machine Specification, Java SE 8 Edition*. Addison-Wesley Professional, 1st edition, 2014.
- 30 Xiaoxing Ma, Luciano Baresi, Carlo Ghezzi, Valerio Panzica La Manna, and Jian Lu. Version-consistent dynamic reconfiguration of component-based distributed systems. In *Proceedings of the ACM SIGSOFT Symposium and the European Conference on Foundations of Software Engineering*, pages 245–255, 2011.
- 31 Stephen Magill, Michael Hicks, Suriya Subramanian, and Kathryn S. McKinley. Automating object transformations for dynamic software updating. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, pages 265–280, 2012.
- 32 Kristis Makris and Rida A. Bazzi. Immediate multi-threaded dynamic software updates using stack reconstruction. In *Proceedings of the Conference on USENIX Annual Technical Conference*, 2009.
- 33 Iulian Neamtiu and Michael Hicks. Safe and timely updates to multi-threaded programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 13–24, 2009.
- 34 Iulian Neamtiu, Michael Hicks, Gareth Stoye, and Manuel Oriol. Practical dynamic software updating for c. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 72–83, 2006.
- 35 Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: A Formal Introduction*. Wiley, 1 edition, 1992.

- 36 Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *Proceedings of the 29th International Conference on Software Engineering*, pages 75–84, 2007.
- 37 Michael Paleczny, Christopher Vick, and Cliff Click. The Java HotSpot™ server compiler. In *Proceedings of the 2001 Symposium on Java™ Virtual Machine Research and Technology Symposium - Volume 1*, pages 1–12, 2001.
- 38 Luís Pina, Luís Veiga, and Michael Hicks. Rubah: DSU for Java on a stock JVM. In *Proceedings of the 2014 International Conference on Object Oriented Programming Systems Languages Applications*, pages 103–119, 2014.
- 39 Rishabh Singh and Sumit Gulwani. Synthesizing number transformations from input-output examples. In *Proceedings of the 24th International Conference on Computer Aided Verification*, pages 634–651, 2012.
- 40 Saurabh Srivastava, Sumit Gulwani, Swarat Chaudhuri, and Jeffrey S. Foster. Path-based inductive synthesis for program inversion. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 492–503, 2011.
- 41 Suriya Subramanian, Michael Hicks, and Kathryn S. McKinley. Dynamic software updates: A VM-centric approach. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, 2009.
- 42 Thomas Würthinger, Christian Wimmer, and Lukas Stadler. Dynamic code evolution for Java. In *Proceedings of the International Conference on the Principles and Practice of Programming in Java*, pages 10–19, 2010.
- 43 Xusheng Xiao, Sihan Li, Tao Xie, and Nikolai Tillmann. Characteristic studies of loop problems for structural test generation via symbolic execution. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, pages 246–256, 2013.
- 44 Cristian Zamfir and George Candea. Execution synthesis: A technique for automated software debugging. In *Proceedings of the 5th European Conference on Computer Systems*, pages 321–334, 2010.
- 45 Min Zhang, Kazuhiro Ogata, and Kokichi Futatsugi. Formalization and verification of behavioral correctness of dynamic software updates. *Electronic Notes in Theoretical Computer Science*, 294(0):12–23, 2013.

FHJ: A Formal Model for Hierarchical Dispatching and Overriding

Yanlin Wang¹

The University of Hong Kong, China
ylwang@cs.hku.hk

Haoyuan Zhang¹

The University of Hong Kong, China
hyzhang@cs.hku.hk

Bruno C. d. S. Oliveira¹

The University of Hong Kong, China
bruno@cs.hku.hk

Marco Servetto

Victoria University of Wellington, New Zealand
marco.servetto@ecs.vuw.ac.nz

Abstract

Multiple inheritance is a valuable feature for Object-Oriented Programming. However, it is also tricky to get right, as illustrated by the extensive literature on the topic. A key issue is the *ambiguity* arising from inheriting multiple parents, which can have conflicting methods. Numerous existing work provides solutions for conflicts which arise from *diamond inheritance*: i.e. conflicts that arise from implementations sharing a common ancestor. However, most mechanisms are inadequate to deal with *unintentional method conflicts*: conflicts which arise from two unrelated methods that happen to share the same name and signature.

This paper presents a new model called *Featherweight Hierarchical Java (FHJ)* that deals with unintentional method conflicts. In our new model, which is partly inspired by C++, conflicting methods arising from unrelated methods can coexist in the same class, and *hierarchical dispatching* supports unambiguous lookups in the presence of such conflicting methods. To avoid ambiguity, hierarchical information is employed in method dispatching, which uses a combination of static and dynamic type information to choose the implementation of a method at run-time. Furthermore, unlike all existing inheritance models, our model supports *hierarchical method overriding*: that is, methods can be *independently overridden* along the multiple inheritance hierarchy. We give illustrative examples of our language and features and formalize **FHJ** as a minimal Featherweight-Java style calculus.

2012 ACM Subject Classification Software and its engineering → Object oriented languages

Keywords and phrases multiple inheritance, hierarchical dispatching, OOP, language design

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2018.20

Acknowledgements We thank the anonymous reviewers for their valuable comments.

¹ Funded by Hong Kong Research Grant Council projects number 17210617 and 17258816



1 Introduction

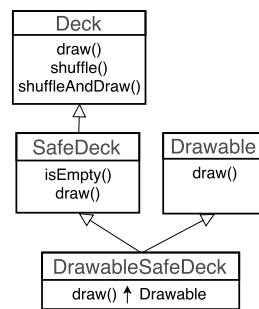
Inheritance in Object-Oriented Programming (OOP) offers a mechanism for code reuse. However many OOP languages are restricted to single inheritance, which is less expressive and flexible than multiple inheritance. Nevertheless, different flavours of multiple inheritance have been adopted in some popular OOP languages. C++ has had multiple inheritance from the start. Scala adapts the ideas from traits [28, 9, 16] and mixins [5, 11, 32, 2, 13] to offer a disciplined form of multiple inheritance. Java 8 offers a simple variant of traits, disguised as interfaces with default methods [12].

A reason why programming languages have resisted to multiple inheritance in the past is that, as Cook [7] puts it, “*multiple inheritance is good but there is no good way to do it*”. One of the most sensitive and critical issues is perhaps the ambiguity introduced by multiple inheritance. One case is the famous *diamond problem* [27, 29] (also known as the *fork-join inheritance* [27]). In the diamond problem, inheritance allows one feature to be inherited from multiple parent classes that share a common ancestor. Hence conflicts arise. The variety of strategies for resolving such conflicts urges the occurrence of different multiple inheritance models, including traits, mixins, CZ [17], and many others. Existing languages and research have addressed the issue of diamond inheritance extensively. Other issues including how multiple inheritance deals with state, have also been discussed quite extensively [33, 17, 31].

In contrast to diamond inheritance, the second case of ambiguity is *unintentional method conflicts* [28]. In this case, conflicting methods do not actually refer to the same feature. In a nominal system, methods can be designed for different functionality but happen to have the same names (and signatures). A simple example of this situation is two `draw` methods that are inherited from a deck of cards and a drawable widget, respectively. In such context, the two `draw` methods have very different meanings, but they happen to share the same name. When inheritance is used to compose these classes, a compilation error happens due to conflicts. However, unlike the diamond problem, the conflicting methods have very different meanings and do not share a common parent. We call such a case *fork inheritance*, in analogy to diamond inheritance.

When unintentional method conflicts happen, they can have severe effects in practice if no appropriate mechanisms to deal with them are available. In practice, existing languages only provide limited support for the issue. In most languages, the mechanisms available to deal with this problem are the same as the diamond inheritance. However, this is often inadequate and can lead to tricky problems in practice. This is especially the case when it is necessary to combine two large modules and their features, but the inheritance is simply prohibited by a small conflict. As a workaround from the diamond inheritance side, it is possible to define a new method in the child class to override those conflicting methods. However, using one method to fuse two unrelated features is clearly unsatisfactory. Therefore we need a better solution to keep both features separately during inheritance, so as not to break *independent extensibility* [36].

C++ and C# do allow for two unintentionally conflicting methods to coexist in a class. C# allows this by interface multiple inheritance and explicit method implementations. But since C# is a single inheritance language, it is only possible to *implement* multiple interfaces (but not multiple classes). C++ accepts fork inheritance and resolves the ambiguity by specifying the expected path by *upcasts*. However, neither the C# nor C++ approaches allow such conflicting methods to be further overridden. Some other workarounds or approaches include delegation and renaming/exclusion in the trait model. However, renaming/exclusion can break the subtyping relation between a subclass and its parent. This is not adequate for the class model commonly used in mainstream OOP languages, where the subclass is always expected to be a subtype of the parent class.



■ **Figure 1** `DrawableSafeDeck`: an illustration of hierarchical overriding.

This paper proposes two mechanisms to deal with unintentional method conflicts: *hierarchical dispatching* and *hierarchical overriding*. Hierarchical dispatching is inspired by the mechanisms in C++ and provides an approach to method dispatching, which combines static and dynamic information. Using hierarchical dispatching, the method binder will look at both the *static type* and the *dynamic type* of the receiver during runtime. When there are multiple branches that cause unintentional conflicts, the static type can specify one branch among them for unambiguity, and the dynamic type helps to find the most specific implementation. In that case, both unambiguity and extensibility are preserved. The main novelty over existing work is the formalization of the essence of a hierarchical dispatching algorithm, which (as far as we know) has not been formalized before.

Hierarchical overriding is a novel language mechanism that allows method overriding to be applied only to one branch of the class hierarchy. Hierarchical overriding adds expressive power that is not available in languages such as C++ or C#. In particular, it allows overriding to work for classes with multiple (conflicting) methods sharing the same names and signatures. An example is presented in Figure 1. In this example, there are 4 classes/interfaces. Two classes `Deck` and `Drawable` model a deck of cards and a drawable widget, respectively. The class `SafeDeck` adds functionality to check whether the deck is empty so as to prevent drawing a card from an empty deck. The interesting class is `DrawableSafeDeck`, which inherits from both `SafeDeck` and `Drawable`. Hierarchical overriding is used in `DrawableSafeDeck` to keep two separate `draw` methods for each parent, but override *only* the `draw` method coming from `Drawable`, in order to draw a widget with a deck of cards. Note that hierarchical overriding is denoted in the UML diagram with the notation `draw() ↑ Drawable`, expressing that the `draw` method from `Drawable` is overridden. Although in this example only one of the `draw` methods is overridden (and the other is simply inherited), hierarchical overriding supports multiple conflicting methods to be independently overridden as well.

To present hierarchical overriding and dispatching, we introduce a formalized model **FHJ** in Section 3 based on Featherweight Java [14], together with theorems and proofs for type soundness. We also have a prototype implementation of an **FHJ** interpreter written in Scala. The implementation validates all the examples presented in the paper. One nice feature of the implementation is that it can show the detailed step-by-step evaluation of the program, which is convenient for understanding and debugging programs & semantics.

In summary, our contributions are:

- **A formalization of the hierarchical dispatching algorithm** that integrates both the static type and dynamic type for method dispatch, and ensures unambiguity as well as extensibility in the presence of unintentional method conflicts.
- **Hierarchical overriding**: a novel notion that allows methods to override individual branches of the class hierarchy.

- **FHJ**: a formalized model based on Featherweight Java, supporting the above features. We provide the static and dynamic semantics and prove the type soundness of the model.
- **Prototype implementation**²: a simple implementation of **FHJ** interpreter in Scala. The implementation can type-check and run variants of all the examples shown in this paper.

2 A Running Example: Drawable Deck

This section illustrates the problem of unintentional method conflicts, together with the features of our model for addressing this issue, by a simple running example. In the following text, we will introduce three problems one by one and have a discussion on possible workarounds and our solutions. Problems 1 and 2 are related to hierarchical dispatching, and in C++ it is possible to have similar solutions to both problems. Hence it is important to emphasize that, with respect to hierarchical dispatching, our model is not a novel mechanism. Instead, inspired by the C++ solutions, our contribution is formalizing a minimal calculus of this feature together with a proof of type soundness. However, for the final problem, there is no satisfactory approach in existing languages, thus what we propose is a novel feature (hierarchical overriding) with the corresponding formalization of that feature.

In the rest of the paper, we use a Java-like syntax for programs. All types are defined with the keyword **interface**; the concept is closely related to Java 8 interfaces with default methods [4] and traits. In short, an interface in our model has the following characteristics:

- It allows multiple inheritance.
- Every method is either abstract or implemented with a body (like Java 8 default methods).
- The **new** keyword is used to instantiate an interface.
- It cannot have state.

2.1 Problem 1: Basic Unintentional Method Conflicts

Suppose that two components **Deck** and **Drawable** have been developed in a system. **Deck** represents a deck of cards and defines a method **draw** for drawing a card from the deck. **Drawable** is an interface for graphics that can be drawn and also includes a method called **draw** for visual display. For simple illustration, the default implementation of the **draw** in **Drawable** only creates a blank canvas on the screen, while the **draw** method in **Deck** simply prints out a message "Draw a card."

```
interface Deck {
  void draw() { // draws a card from the Deck
    println("Draw a card.");
  }
}
interface Drawable {
  void draw() { // create a blank canvas
    JFrame frame = new JFrame();
    frame.setVisible(true);
  }
}
```

In **Deck**, **draw** uses **println**, which is a library function. The two **draw** methods can have different return types, but for simplicity, the return types are both **void** here. Note that,

² The implementation is available at <https://github.com/YanlinWang/MIM/tree/master/Calculus>

similarly to Featherweight Java [14], `void` is unsupported in our formalization. We could have also defined an interface called `Void` and return an object of that type instead. To be concise, however, we use `void` in our examples. In interface `Drawable`, the `draw` method creates a blank canvas.

Now, suppose that a programmer is designing a card game with a GUI. He may want to draw a deck on the screen, so he first defines a drawable deck using multiple inheritance:

```
interface DrawableDeck extends Drawable, Deck {}
```

The point of using multiple inheritance is to compose the features from various components and to achieve code reuse, as supported by many mainstream OO languages. Nevertheless, at this point, languages like Java simply treat the two `draw` methods as the same, hence the compiler fails to compile the program and reports an error.

This case is an example of a so-called *unintentional method conflict*. It arises when two inherited methods happen to have the same name and parameter types, but they are designed for different functionalities with different semantics. Now one may quickly come up with a workaround, which is to manually merge the two methods by creating a new `draw` method in `DrawableDeck` to override the old ones. However, merging two methods with totally different functionalities does not make any sense. This non-solution would hide the old methods and break independent extensibility.

2.1.1 Problem and Possible Workarounds

The essential problem is how to resolve unintentional method conflicts and invoke the conflicting methods separately without ambiguity. To tackle this problem, there are several other workarounds that come to our mind. We briefly discuss those potential fixes and workarounds next:

- *I. Delegation.* As an alternative to multiple inheritance, delegation can be used by introducing two fields (or field methods) with the `Drawable` type and `Deck` type, respectively. Although it avoids method conflicts, it is known that using delegation makes it hard to correctly maintain self-references in an extensible system and also introduces a lot of boilerplate code.
- *II. Refactor `Drawable` and/or `Deck` to rename the methods.* If the source code for `Drawable` or `Deck` is available then it may be possible to rename one of the `draw` methods. However, this approach is non-modular, as it requires modifying existing code and becomes impossible if the code is unavailable.
- *III. Method exclusion/renaming.* Eiffel [18] and some trait models support method exclusion/renaming. Those features can eliminate conflicts, although most programming languages do not support them. In a traditional OO system, they can break the subtyping relationship. Moreover, in contrast with exclusion, renaming can indeed preserve both conflicting behaviours. However, it is cumbersome in practice, as introducing new names can affect other code blocks.

2.1.2 FHJ's solution

To solve this problem it is important to preserve both conflicting methods during inheritance instead of merging them into a single method. Therefore **FHJ** accepts the definition of `DrawableDeck`. To disambiguate method calls, we can use *upcasts* in **FHJ** to specify the “branch” in the inheritance hierarchy that should be called. The following code illustrates the use of upcasts for disambiguation:

```

interface Deck { void draw() {...} }
interface Drawable { void draw() {...} }
interface DrawableDeck extends Drawable, Deck {}
// main program
((Deck) new DrawableDeck()).draw() // calls Deck.draw
// new DrawableDeck().draw() // this call is ambiguous and rejected

```

In our language, a program consists of interfaces declarations and a main an expression which produces the final result. In the above main expression `((Deck) new DrawableDeck()).draw()`, the cast indicates that we expect to invoke the `draw` method from the branch `Deck`. Similarly, we could have used an upcast to `Drawable` to call the `draw` method from `Drawable`. Without the cast, the call would be ambiguous and **FHJ**'s type system would reject it.

This example illustrates the basic form of fork inheritance, where two unintentionally conflicting methods are accepted by multiple inheritance. Note that C++ supports this feature and also addresses the ambiguity by upcasts. The code for the above example in C++ is similar.

2.2 Problem 2: Dynamic Dispatching

Using explicit upcasts for disambiguation helps when making calls to classes with conflicting methods, but things become more complicated with dynamic dispatching. Dynamic dispatching is very common in OO programming for code reuse. Let us expand the previous example a bit, by redefining those interfaces with more features:

```

interface Deck {
  void draw() {...}
  void shuffle() {...}
  void shuffleAndDraw() { this.shuffle(); this.draw(); }
}

```

Here `shuffleAndDraw` invokes `draw` from its own enclosing type. In **FHJ**, this invocation is dynamically dispatched. This is important, because a programmer may define a subtype of `Deck` and override the method `draw`:

```

interface SafeDeck extends Deck {
  boolean isEmpty() {...}
  void draw() { // overriding
    if (isEmpty()) println("The deck is empty.");
    else println("Draw a card");
  }
}

```

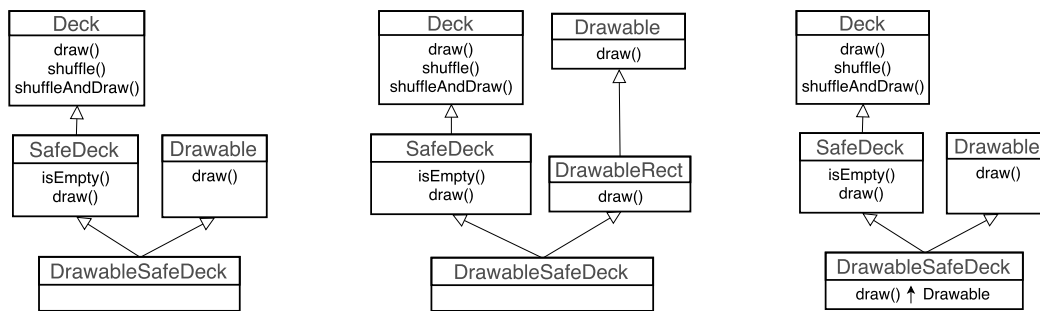
Without dynamic dispatching, we may have to copy the `shuffleAndDraw` code into `SafeDeck`, so that `shuffleAndDraw` calls the new `draw` defined in `SafeDeck`. Dynamic dispatching immediately saves us from the duplication work, since the method becomes automatically dispatched to the most specific one. Nevertheless, as seen before, dynamic dispatch would potentially introduce ambiguity. For instance, when we have the class hierarchy structure shown in Figure 2(left) with the following code:

```

interface DrawableSafeDeck extends Drawable, SafeDeck {}
new DrawableSafeDeck().shuffleAndDraw()

```

Indeed, using reduction steps following the reduction rules in FJ [14]-like languages, where no static types are tracked, the reduction steps would roughly be:



■ **Figure 2** UML diagrams for 3 variants of `DrawableSafeDeck`.

```

new DrawableSafeDeck().shuffleAndDraw()
-> new DrawableSafeDeck().shuffle(); new DrawableSafeDeck().draw()
-> ...
-> new DrawableSafeDeck().draw()
-> <<error: ambiguous call!!!>>

```

When the `DrawableSafeDeck` object calls `shuffleAndDraw`, the implementation in `Deck` is dispatched. But then `shuffleAndDraw` invokes “`this.draw()`”, and at this point, the receiver is replaced by the object `new DrawableSafeDeck()`. From the perspective of `DrawableSafeDeck`, the `draw` method seems to be ambiguous since `DrawableSafeDeck` inherits two `draw` methods from both `SafeDeck` and `Drawable`. But ideally we would like `shuffleAndDraw` to invoke `SafeDeck.draw` because they belong to the same class hierarchy branch.

2.2.1 FHJ’s solution

The essential problem is how to ensure that the correct method is invoked. To solve this problem, **FHJ** uses a variant of method dispatching that we call *hierarchical dispatching*. In hierarchical dispatching, both the static and dynamic type information are used to select the right method implementation. During runtime, a method call makes use of both the static type and the dynamic type of the receiver, so it is a combination of static and dynamic dispatching. Intuitively, the static type specifies one branch to avoid ambiguity, and the dynamic type finds the most specific implementation on that branch. To be specific, the following code is accepted by **FHJ**:

```

interface Deck {
  void draw() {...}
  void shuffle() {...}
  void shuffleAndDraw() { this.shuffle(); this.draw(); }
}
interface Drawable {...}
interface SafeDeck extends Deck {...}
interface DrawableSafeDeck extends Drawable, SafeDeck {}
new DrawableSafeDeck().shuffleAndDraw() // SafeDeck.draw is called

```

The computation performed in **FHJ** is as follows:

```

new DrawableSafeDeck().shuffleAndDraw()
-> ((DrawableSafeDeck) new DrawableSafeDeck()).shuffleAndDraw()
-> ((Deck) new
  DrawableSafeDeck()).shuffle(); ((Deck) new DrawableSafeDeck()).draw()
-> ...
-> ((Deck) new DrawableSafeDeck()).draw()
-> ... // SafeDeck.draw

```

Notably, we track the static types by adding upcasts during reduction. In contrast to FJ, where `new C()` is a value, in **FHJ** such an expression is not a value. Instead, an expression of the form `new C()` is reduced to `(C) new C()`, which is a value in **FHJ** and the cast denotes the static type of the expression. This rule is applied in the first reduction step. In the second reduction step, when `shuffleAndDraw` is dispatched, the receiver `(DrawableSafeDeck)new DrawableSafeDeck()` replaces the special variable `this` by `(Deck) new DrawableSafeDeck()`. Here, the static type used in the cast `(Deck)` denotes the origin of the `shuffleAndDraw` method, which is discovered during method lookup. Later, in the fourth step, `((Deck) new DrawableSafeDeck()).draw()` is an instance of *hierarchical invocation*, which can be read as “finding the most specific `draw` above `DrawableSafeDeck` and along path `Deck`”. The meaning of “above `DrawableSafeDeck`” implies its supertypes, and “along path `Deck`” specifies the branch. Finally, in the last reduction step, we find the most specific version of `draw` in `SafeDeck`. In this sequence of reduction steps, the cast that tracks the origin of `shuffleAndDraw` is crucial to unambiguously find the correct implementation of `draw`. The formal procedure will be introduced in Section 3 and Section 4.

2.3 Problem 3: Overriding on Individual Branches

Method overriding is common in Object-Oriented Programming. With diamond inheritance, where conflicting methods are intended to have the same semantics, method overriding is not a problem. If conflicting methods arise from multiple parents, we can override all those methods in a single unified (or merged) method in the subclass. Therefore further overriding is simple, because there is only one method that can be overridden.

With unintentional method conflicts, however, the situation is more complicated because different, separate, conflicting methods can coexist in one class. Ideally, we would like to support overriding for those methods too, in exactly the same way that overriding is available for other (non-conflicting) methods. However, we need to be able to override the individual conflicting methods, rather than overriding all conflicting methods into a single merged one.

We illustrate the problem and the need for a more refined overriding mechanism with an example. Suppose that the programmer defines a new interface `DrawableSafeDeck` (based on the code in Section 2.2 without the old `DrawableSafeDeck`), but he needs to override `Drawable.draw` and give a new implementation of drawing so that the deck can indeed be visualized on the canvas.

2.3.1 Potential solutions/workarounds in existing languages

Unfortunately in all languages we know of (including C++), the existing approaches are unsatisfactory. One direction is to simply avoid this issue, by putting overriding before inheritance. For example, as shown in Figure 2(middle), we define a new component `DrawableRect` that extends `Drawable`, which simply draws the deck as a rectangle, and modifies the hierarchy:

```
interface DrawableRect extends Drawable {
    void draw() {
        JFrame frame = new JFrame("Canvas");
        frame.setSize(600, 600);
        frame.getContentPane().setBackground(Color.red);
        frame.getContentPane().add(new Square(10,10,100,100)); ...
    }
}
interface DrawableSafeDeck extends DrawableRect, SafeDeck {}
```

This workaround seems to work, but there are severe issues:

- It changes the hierarchy and existing code, hence breaks the modularity.
- Separate overriding is required to come after the fork inheritance, especially when the implementation needs functionality from both parents. In the above code, we have assumed that the overriding is unrelated to `Deck`. But when the drawing relies on some information of the `Deck` object, we have to either introduce field methods for delegation or change the signature of `draw` to take a parameter. Either way introduces unnecessary complexity and affects extensibility.

There are more involved workarounds in C++ using templates and complex patterns, but such patterns are complex to use and there are still issues. A more detailed discussion of such an approach is presented in Section 6.2.

2.3.2 FHJ's solution

An additional feature of our model is *hierarchical overriding*. It allows conflicting methods to be overridden on individual branches, hence offers independent extensibility. The above example can be easily realized by:

```
interface DrawableSafeDeck extends Drawable, SafeDeck {
  void draw() override Drawable {
    JFrame frame = new JFrame("Canvas");
    frame.setSize(600, 600);
    frame.getContentPane().setBackground(Color.red);
    frame.getContentPane().add(new Square(10,10,100,100)); ...
  }
}
((Drawable)new DrawableSafeDeck()).draw(); //calls the draw in DrawableSafeDeck
```

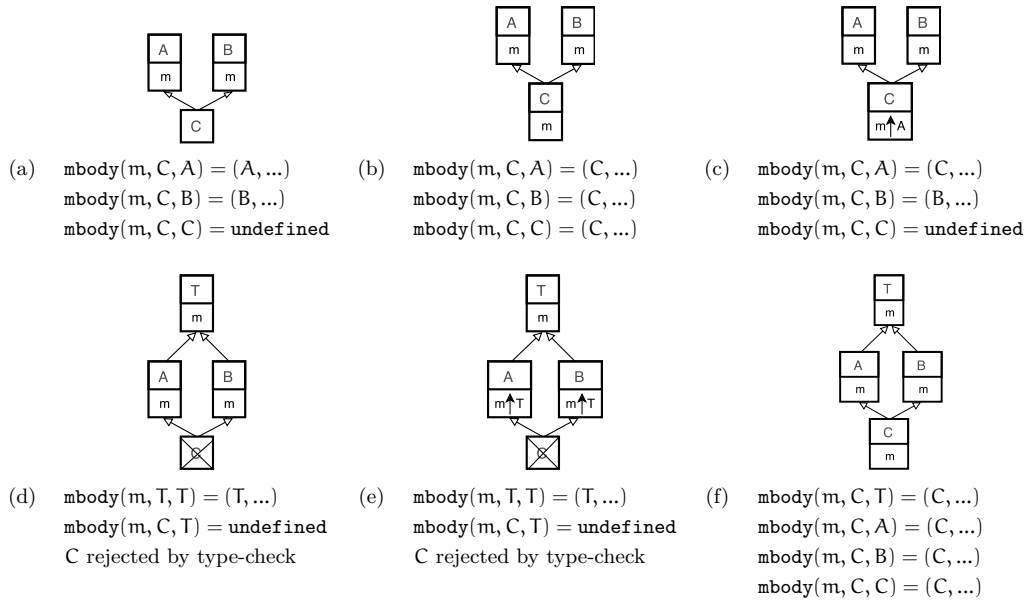
The UML graph is shown in Figure 2(right), where the up-arrow \uparrow is short for “**override**”. Here the idea is that *only* `Drawable.draw` is overridden. This is accomplished by specifying, in the method definition, that the method only overrides the `draw` from `Drawable`. The individual overriding allows us to make use of the methods from `SafeDeck` as well. In the formalization, the hierarchical overriding feature is an important feature, involved in the algorithm of hierarchical dispatch.

Note that, although the example here only shows one conflicting method being overridden, hierarchical overriding allows (as expected) multiple conflicting methods to be overridden in the same class.

2.3.3 Terminology

In `Drawable`, `Deck`, and `SafeDeck`, the `draw` methods are called *original methods* in this paper, because they are originally defined by the interfaces. In contrast, `DrawableSafeDeck` defines a *hierarchical overriding method*. The difference is that traditional method overriding overrides all branches by defining another original method, whereas hierarchical overriding only refines one branch.

A special rule for hierarchical overriding is: it can only refine *original* methods, and cannot jump over original methods with the same signature. For instance, writing “`void draw() override Deck {...}`” is disallowed in `DrawableSafeDeck`, because the existing two branches are `Drawable.draw` and `SafeDeck.draw`, while `Deck.draw` is already covered. It does not really make sense to refine the old branch `Deck`.



■ **Figure 3** Examples in **FHJ**. “ $m \uparrow A$ ” stands for hierarchical overriding “ m override A ”.

2.3.4 A peek at the hierarchical dispatching algorithm

In **FHJ**, fork inheritance allows several original methods (branches) to coexist, and hierarchical dispatch first finds the most specific original method (branch), then it finds the most specific hierarchical overriding on that branch.

Before the formalized algorithm, Figure 3 gives a peek at the behavior using a few examples. The UML diagrams present the hierarchy. In (d) and (e), a cross mark indicates that the interface fails to type-check. Generally, **FHJ** rejects the definition of an interface during compilation if it reaches a diamond with ambiguity. mbody is the method lookup function for hierarchical dispatch, formally defined in Section 4.1. In general, $\text{mbody}(m, X, Y) = (Z, \dots)$ reflects that the source code $((Y \text{ new } X()).m())$ calls $Z.m$ at runtime. It is undefined when method dispatch is ambiguous.

In Figure 3, (a) is the base case for unintentional conflicts, namely the fork inheritance. (b) uses overriding to explicitly merge the conflicting methods. (c) represents hierarchical overriding.

Furthermore, our model supports diamond inheritance and can deal with diamond problems. For example, (d) and (e) are two base cases of diamond inheritance in **FHJ** and the definition of each C is rejected because T is an ambiguous parent to C . One solution for diamond inheritance is to merge methods coming from different parents. (f) gives a common solution to the diamond as in Java or traits, which is to explicitly override $A.m$ and $B.m$ in C . And our calculus supports this kind of merging methods. In the last three examples, conflicting methods $A.m$ and $B.m$ should be viewed as intentional conflicts, as they come from the same source T .

3 Formalization

In this section, we present a formal model called **FHJ** (*Featherweight Hierarchical Java*), following a similar style as Featherweight Java [14]. **FHJ** is a minimal core calculus that formalizes the core concept of hierarchical dispatching and overriding. The syntax, typing rules and small-step semantics are presented.

3.1 Syntax

The abstract syntax of **FHJ** interface declarations, method declarations, and expressions is given in Figure 4. The multiple inheritance feature of **FHJ** is inspired by Java 8 interfaces, which supports method implementations via default methods. This feature is closely related to *traits*. To demonstrate how unintentional method conflicts are untangled in **FHJ**, we only focus on a small subset of the interface model. For example, all methods declared in an interface are either default methods or abstract methods. Default methods provide default implementations for methods. Abstract methods do not have a method body. Abstract methods can be overridden with future implementations.

3.1.1 Notations

The metavariables I, J, K range over interface names; x ranges over variables; m ranges over method names; e ranges over expressions; and M ranges over method declarations. Following Featherweight Java, we assume that the set of variables includes the special variable **this**, which cannot be used as the name of an argument to a method. We use the same conventions as FJ; we write \bar{I} as shorthand for a possibly empty sequence I_1, \dots, I_n , which may be indexed by I_i ; and write \bar{M} as shorthand for $M_1 \dots M_n$ (with no commas). We also abbreviate operations on pairs of sequences in an obvious way, writing $\bar{I} \bar{x}$ for $I_1 x_1, \dots, I_n x_n$, where n is the length of \bar{I} and \bar{x} .

3.1.2 Interfaces

In order to achieve multiple inheritance, an interface can have a set of parent interfaces, where such a set can be empty. Moreover, as usual in class-based languages, the extension relation over interfaces is acyclic. The interface declaration **interface** I **extends** \bar{I} $\{\bar{M}\}$ introduces an interface named I with parent interfaces \bar{I} and a suite of methods \bar{M} . The methods of I may either override methods that are already defined in \bar{I} or add new functionality special to I , we will illustrate this in more detail later.

3.1.3 Methods

Original methods and hierarchically overriding methods share the same syntax in our model for simplicity. The concrete method declaration $I \ m(\bar{I}_x \ \bar{x}) \ \mathbf{override} \ J \ \{\mathbf{return} \ e;\}$ introduces a method named m with result type I , parameters \bar{x} of type \bar{I}_x and the overriding target J . The body of the method simply includes the returned expression e . Notably, we have introduced the **override** keyword for two cases. Firstly, if the overridden interface is exactly the enclosing interface itself, then such a method is seen as *originally defined*. Note that the case of merging methods from different branches, also counts as originally defined. Secondly, for all other cases the method is considered a *hierarchical overriding method*. Note that in an interface J , $I \ m(\bar{I}_x \ \bar{x}) \ \{\mathbf{return} \ e;\}$ is syntactic sugar for $I \ m(\bar{I}_x \ \bar{x}) \ \mathbf{override} \ J \ \{\mathbf{return} \ e;\}$, which is the standard way to define methods in Java-like languages. The definition of abstract methods is written as $I \ m(\bar{I}_x \ \bar{x}) \ \mathbf{override} \ J \ ;$, which is similar to a concrete method but without the method body. For simplicity, overloading is not modelled for methods, which implies that we can uniquely identify a method by its name.

Interfaces	IL	$::=$	<code>interface I extends \bar{I} {\bar{M}}</code>
Methods	M	$::=$	<code>I m($\bar{I}_x \bar{x}$) override J {return e;} I m($\bar{I}_x \bar{x}$) override J ;</code>
Expressions	e	$::=$	<code>x e.m(\bar{e}) new I() (I)e</code>
Context	Γ	$::=$	<code>$\bar{x} : \bar{I}$</code>
Values	v	$::=$	<code>(I)new J()</code>

■ **Figure 4** Syntax of **FHJ**.

3.1.4 Expressions & Values

Expressions can be standard constructs such as variables, method invocation, object creation, together with cast expressions. Object creation is represented by `new I()`³. Fields and primitive types are not modelled in **FHJ**. The casts are merely safe upcasts, and in fact, they can be viewed as annotated expressions, where the annotation indicates its static type. The coexistence of static and dynamic types is the key to hierarchical dispatch. A value “(I)new J()” is the final result of multiple reduction steps for evaluating an expression.

For simplicity, **FHJ** does not formalize statements like assignments and so on because they are orthogonal features to the hierarchical dispatching and overriding feature. A program in **FHJ** consists of a list of interface declarations, plus a single expression.

3.2 Subtyping and Typing Rules

3.2.1 Subtyping

The subtyping of **FHJ** consists of only a few rules shown at the top of Figure 5. In short, subtyping relations are built from the inheritance in interface declarations. Subtyping is both reflexive and transitive.

3.2.2 Type-checking

Details of type-checking rules are displayed at the bottom of Figure 5, including expression typing, well-formedness of methods and interfaces. As a convention, an environment Γ is maintained to store the types of variables, together with the self-reference `this`.

(T-INVK) is the typing rule for method invocation. Naturally, the receiver and the arguments are required to be well-typed. `mbody` is our key function for method lookup that implements the hierarchical dispatching algorithm. The formal definition will be introduced in Section 4. Here `mbody(m, I0, I0)` finds the most specific `m` above `I0`. “Above `I0`” specifies the search space, namely the supertypes of `I0` including itself. For the general case, however, the hierarchical invocation `mbody(m, I, J)` finds “the most specific `m` above `I` and along path/branch `J`”. “Along path `J`” additionally requires the result to relate to `J`, that is to say, the most specific interface that has a subtyping relationship with `J`.

In (T-INVK), as the compilation should not be aware of the dynamic type, it only requires that invoking `m` is valid for the static type of the receiver. The result of `mbody` contains the interface that provides the most specific implementation, the parameters and the return type. We use underscore for the return expression, matching both implemented and abstract methods.

³ In Java the corresponding syntax is `new I(){}`.

$$\begin{array}{c}
\boxed{I <: J} \qquad \overline{I <: I} \\
\\
\frac{I <: J \quad J <: K}{I <: K} \qquad \frac{\text{interface } I \text{ extends } I_1, I_2, \dots, I_n \{ \overline{M} \}}{I <: I_1, I <: I_2, \dots, I <: I_n} \\
\\
\boxed{\Gamma \vdash e : I} \qquad (\text{T-VAR}) \Gamma \vdash x : \Gamma(x) \\
\\
(\text{T-INVK}) \frac{\Gamma \vdash e_0 : I_0 \quad \text{mbody}(m, I_0, I_0) = (K, \overline{J} \overline{x}, I _) \quad \Gamma \vdash \overline{e} : \overline{I} \quad \overline{I} <: \overline{J}}{\Gamma \vdash e_0.m(\overline{e}) : I} \\
\\
(\text{T-NEW}) \frac{\text{interface } I \text{ extends } \overline{I} \{ \overline{M} \} \quad \text{canInstantiate}(I)}{\Gamma \vdash \text{new } I() : I} \\
\\
(\text{T-ANNO}) \frac{\Gamma \vdash e : I \quad I <: J}{\Gamma \vdash (J)e : J} \\
\\
(\text{T-METHOD}) \frac{I <: J \quad \text{findOrigin}(m, I, J) = \{J\} \quad \text{mbody}(m, J, J) = (K, \overline{I_x} \overline{x}, I_e _) \quad \overline{x} : \overline{I_x}, \text{this} : I \vdash e_0 : I_0 \quad I_0 <: I_e}{I_e \text{ m}(\overline{I_x} \overline{x}) \text{ override } J \{ \text{return } e_0; \} \text{ OK IN } I} \\
\\
(\text{T-ABSMETHOD}) \frac{I <: J \quad \text{findOrigin}(m, I, J) = \{J\} \quad \text{mbody}(m, J, J) = (K, \overline{I_x} \overline{x}, I_e _)}{I_e \text{ m}(\overline{I_x} \overline{x}) \text{ override } J ; \text{ OK IN } I} \\
\\
(\text{T-INTF}) \frac{\overline{M} \text{ OK IN } I \quad \forall J :> I \text{ and } m, \text{mbody}(m, J, J) \text{ is defined} \Rightarrow \text{mbody}(m, I, J) \text{ is defined} \quad \forall J :> I \text{ and } m, I[m \text{ override } I] \text{ and } J[m \text{ override } J] \text{ defined} \Rightarrow \text{canOverride}(m, I, J)}{\text{interface } I \text{ extends } \overline{I} \{ \overline{M} \} \text{ OK}}
\end{array}$$

■ **Figure 5** Subtyping and Typing Rules of **FHJ**.

(T-NEW) is the typing rule for object creation `new I()`. The auxiliary function `canInstantiate(I)` (see definition in Section 4.4) checks whether an interface `I` can be instantiated or not. Since fork inheritance accepts conflicting branches to coexist, the check requires that the most specific method is concrete for each method on each branch.

(T-METHOD) is more interesting since a method can either be an original method or a hierarchical overriding, though they share the same syntax and method typing rule. `findOrigin(m, I, J)` is a fundamental function, used to find “the most specific interfaces that are above `I` and along path `J`, and originally defines `m`” (see Section 4 for full definition). By “most specific interfaces”, it implies that the inherited supertypes are excluded. Thus the condition `findOrigin(m, I, J) = {J}` indicates a characteristic of a hierarchical overriding: it must override an original method; the overriding is direct and there does not exist any other original method `m` in between. Then `mbody(m, J, J)` provides the type of the original method, so hierarchical overriding has to preserve the type. Finally the return expression is type-checked to be a subtype of the declared return type. For the definition of an original method, `I` equals `J` and the rule is straightforward. (T-ABSMETHOD) is a similar rule but works on abstract method declarations.

(T-INTF) defines the typing rule on interfaces. The first condition is obvious, namely, its methods need to be well checked. The third condition checks whether the overriding between original methods preserves typing. In this condition we again use some helper functions defined

in Section 4. $I[m \text{ override } I]$ is defined if I originally defines m , and $\text{canOverride}(m, I, J)$ checks whether $I.m$ has the same type as $J.m$. Generally the preservation of method type is required for any supertype J and any method m .

The second condition of (T-INTF) is more complex and is the key to type soundness. Unlike C++ which rejects on ambiguous calls, **FHJ** rejects on the definition of interfaces when they form a diamond. Consider the case when the second condition is broken: $\text{mbody}(m, J, J)$ is defined but $\text{mbody}(m, I, J)$ is undefined for some J and m . This indicates that m is available and unambiguous from the perspective of J , but is ambiguous to I on branch J . It means that there are multiple overriding paths of m from J to I , which form a diamond. Hence rejecting that case meets our expectation. Below is an example (Figure 3 (e)) that illustrates the reason why this condition is needed:

```
interface T           { T m() override T { return new T(); } }
interface A extends T { T m() override T { return new A(); } }
interface B extends T { T m() override T { return new B(); } }
interface C extends A, B {}
((T) new C()).m()
```

This program does not compile on interface C , because of the second condition in (T-INTF), where I equals C and J equals T . By the algorithm, $\text{mbody}(m, T, T)$ will refer to $T.m$, but $\text{mbody}(m, C, T)$ is undefined, since both $A.m$ and $B.m$ are most specific to C along path T , which forms a diamond. The expression $((T) \text{ new } C()).m()$ is one example of triggering ambiguity, but **FHJ** simply rejects the definition of C . To resolve the issue, the programmer needs to have an overriding method in C , to explicitly merge the conflicting ones.

Finally, rule (T-ANNO) is the typing rule for a cast expression. By the rule, only upcasts are valid.

3.3 Small-step Semantics and Propagation

Figure 6 defines the small-step semantics and propagation rules of **FHJ**. When evaluating an expression, they are invoked and produce a single value in the end.

3.3.1 Semantic Rules

(S-INVK) is the only computation rule we need for method invocation. As a small-step rule and by congruence, it assumes that the receiver and the arguments are already values. Specifically, the receiver $(J)\text{new } I()$ indicates the dynamic type I together with the static type J . Therefore $\text{mbody}(m, I, J)$ carries out hierarchical dispatching, acquires the types, the return expression e_0 and the interface I_0 which provides the most specific method. Here we use e_0 to imply that the return expression is forced to be non-empty because it requires a concrete implementation. Now the rule reduces method invocation to e_0 with substitution. Parameters are substituted with arguments, and the **this** reference is substituted with the receiver, and in the meanwhile the static types are recorded via annotations. Finally, the return type I_e is put in the front as an annotation.

3.3.2 Propagation Rules

(C-RECEIVER), (C-ARGS) and (C-FREDUCE) are natural propagation rules on receivers, arguments, and cast-expressions, respectively. (C-STATICTYPE) automatically adds an annotation I to the new object $\text{new } I()$. (C-ANNOREDUCE) merges nested upcasts into a single upcast with the outermost type.

4 Key Algorithms and Type-Soundness

In this section, we present the fundamental algorithms and auxiliary definitions used in our formalization and show that the resulting calculus is type sound. The functions presented in this section are the key components that implement our algorithm for method lookup.

4.1 The Method Lookup Algorithm in `mbody`

`mbody(m, Id, Is)` denotes the method body lookup function. We use `Id, Is`, since `mbody` is usually invoked by a receiver of a method `m`, with its dynamic type `Id` and static type `Is`. Such a function returns the most specific method implementation. More accurately, `mbody` returns $(J, \overline{I_x} \overline{x}, I_e e_0)$ where `J` is the found interface that contains the desired method; $\overline{I_x} \overline{x}$ are the parameters and its types, `e0` is the returned expression (empty for abstract methods). It considers both originally-defined methods and hierarchical overriding methods, so `findOrigin` and `findOverride` (see the definition in Section 4.2 and Section 4.3) are both invoked. The formal definition gives the expected results for the earlier examples in Figure 3.

▷ *Definition of `mbody(m, Id, Is)`:*

- `mbody(m, Id, Is) = (J, $\overline{I_x} \overline{x}, I_e e_0$)`
with: `findOrigin(m, Id, Is) = {I}`
`findOverride(m, Id, I) = {J}`
`J[m override I] = Ie m($\overline{I_x} \overline{x}$) override I {return e0};`
- `mbody(m, Id, Is) = (J, $\overline{I_x} \overline{x}, I_e \emptyset$)`
with: `findOrigin(m, Id, Is) = {I}`
`findOverride(m, Id, I) = {J}`
`J[m override I] = Ie m($\overline{I_x} \overline{x}$) override I;`

To calculate `mbody(m, Id, Is)`, the invocation of `findOrigin` looks for the most specific original methods and their interfaces, and expects a singleton set, so as to achieve unambiguity. Furthermore, the invocation of `findOverride` also expects a unique and most specific hierarchical override. And finally the target method is returned.

4.2 Finding the Most Specific Origin: `findOrigin`

We proceed to give the definitions of two core functions that support method lookup, namely, `findOrigin` and `findOverride`. Generally, `findOrigin(m, I, J)` finds the set of most specific interfaces where `m` is originally defined. Interfaces in this set should be above interface `I` and along path `J`. Finally with `prune` (defined in Section 4.4) the overridden interfaces will be filtered out.

▷ *Definition of `findOrigin(m, I, J)`:*

- `findOrigin(m, I, J) = prune(origins)`
with: `origins = {K | I <: K, and K <: J ∨ J <: K,`
and `K[m override K]` is defined}

By the definition, an interface belongs to `findOrigin(m, I, J)` if and only if:

$$\begin{array}{c}
\text{(S-INVK)} \frac{\text{mbody}(\text{m}, \text{I}, \text{J}) = (\text{I}_0, \overline{\text{I}_x} \overline{x}, \text{I}_e \text{e}_0)}{((\text{J})\text{new I}()) . \text{m}(\overline{v}) \rightarrow (\text{I}_e)[\overline{\text{I}_x} \overline{v} / \overline{x}, (\text{I}_0)\text{new I}() / \text{this}] \text{e}_0} \\
\\
\text{(C-RECEIVER)} \frac{\text{e}_0 \rightarrow \text{e}'_0}{\text{e}_0 . \text{m}(\overline{e}) \rightarrow \text{e}'_0 . \text{m}(\overline{e})} \quad \text{(C-ARGS)} \frac{\text{e} \rightarrow \text{e}'}{\text{e}_0 . \text{m}(\dots, \text{e}, \dots) \rightarrow \text{e}_0 . \text{m}(\dots, \text{e}', \dots)} \\
\\
\text{(C-STATIC TYPE)} \frac{}{\text{new I}() \rightarrow (\text{I})\text{new I}()} \\
\\
\text{(C-FREDUCE)} \frac{\text{e} \rightarrow \text{e}' \quad \text{e} \neq \text{new J}()}{(\text{I})\text{e} \rightarrow (\text{I})\text{e}'} \\
\\
\text{(C-ANNOREDUCE)} (\text{I})((\text{J})\text{new K}()) \rightarrow (\text{I})\text{new K}()
\end{array}$$

■ **Figure 6** Small-step semantics.

- It originally defines m ;
- It is a supertype of I (including I);
- It is either a supertype or a subtype of J (including J);
- Any subtype of it does not belong to the same result set because of `prune`.

4.3 Finding the Most Specific Overriding: `findOverride`

The `findOrigin` function only focuses on original method implementations, where all the hierarchical overriding methods are omitted during that step. On the other hand, `findOverride(m, I, J)` has the assumption that J defines an original m , and this function tries to find the interfaces with the most specific implementations that hierarchically overrides such an m . Formally,

▷ *Definition of `findOverride(m, I, J)`:*

- `findOverride(m, I, J) = prune(overrides)`

with: `overrides = {K | I <: K, K <: J and K[m override J] is defined}`

By the definition, an interface belongs to `findOverride(m, I, J)` if and only if:

- it is between I and J (including I , J);
- it hierarchically overrides J.m ;
- any subtype of it does not belong to the same set.

4.4 Other Auxiliaries

Below we give other minor definitions of the auxiliary functions that are used in previous sections.

▷ *Definition of $I[m \text{ override } J]$:*

- $I[m \text{ override } J] = I_e \ m(\overline{I_x} \ \overline{x}) \text{ override } J \ \{\text{return } e_0;\}$
with: `interface I extends \overline{I} { $I_e \ m(\overline{I_x} \ \overline{x}) \text{ override } J \ \{\text{return } e_0;\}$...}`
- $I[m \text{ override } J] = I_e \ m(\overline{I_x} \ \overline{x}) \text{ override } J$;
with: `interface I extends \overline{I} { $I_e \ m(\overline{I_x} \ \overline{x}) \text{ override } J$;...}`

Here $I[m \text{ override } J]$ is basically a direct lookup for method m in the body of I , where such a method overrides J (like static dispatch). The method can be either concrete or abstract, and the body of definition is returned. Notice that by our syntax, $I[m \text{ override } I]$ is looking for the originally-defined method m in I .

▷ *Definition of $\text{prune}(\text{set})$:*

- $\text{prune}(\text{set}) = \{I \in \text{set} \mid \nexists J \in \text{set} \setminus I, J <: I\}$

The `prune` function takes a set of types, and filters out those that have subtypes in the same set. In the returned set, none of them has subtyping relation to one another, since all supertypes have been removed.

▷ *Definition of $\text{canOverride}(m, I, J)$:*

- $\text{canOverride}(m, I, J)$ holds
iff: $I[m \text{ override } I] = I_e \ m(\overline{I_x} \ \overline{x}) \text{ override } I \dots$
 $J[m \text{ override } J] = I_e \ m(\overline{I_x} \ \overline{y}) \text{ override } J \dots$

`canOverride` just checks that two original m in I and J have the same type.

▷ *Definition of $\text{canInstantiate}(I)$:*

- $\text{canInstantiate}(I)$ holds
iff: $\forall m, \forall J \in \text{findOrigin}(m, I, I), \text{findOverride}(m, I, J) = \{K\}$,
and $K[m \text{ override } J] = I_e \ m(\overline{I_x} \ \overline{x}) \text{ override } J \ \{\text{return } e_0;\}$

`canInstantiate(I)` checks whether interface I can be instantiated by the keyword `new`. `findOrigin(m, I, I)` represents the set of branches that I inherits on method m . I can be instantiated if and only if for every branch, the most specific implementation is non-abstract.

4.5 Properties

We present the type soundness of the model by a few theorems below, following the standard technique of subject reduction and progress proposed by Wright and Felleisen [35]. The proof, together with some lemmas, is presented in Appendix. Type soundness states that if an expression is well-typed, then after many reduction steps it must reduce to a value, and its annotation is the same as the static type of the original expression.

► **Theorem 1** (Subject Reduction). *If $\Gamma \vdash e : I$ and $e \rightarrow e'$, then $\Gamma \vdash e' : I$.*

Proof. See Appendix A.1. ◀

► **Theorem 2 (Progress).** *Suppose e is a well-typed expression, if e includes $((J)\text{new } I()) \cdot m(\bar{v})$ as a sub-expression, then $mbody(m, I, J) = (I_0, \overline{I_x} \bar{x}, I_e e_0)$ and $\#(\bar{x}) = \#(\bar{v})$ for some $I_0, \overline{I_x}, \bar{x}, I_e$ and e_0 .*

Proof. See Appendix A.1. ◀

► **Theorem 3 (Type Soundness).** *If $\phi \vdash e : I$ and $e \rightarrow^* e'$ with e' a normal form, then e' is a value v with $\phi \vdash v : I$.*

Proof. Immediate from Theorem 1 and Theorem 2. ◀

Note that in Theorem 2, “ $\#(\bar{x})$ ” denotes the length of \bar{x} .

Our theorems are stricter than those of Featherweight Java [14]. In FJ, the subject reduction theorem states that after a step of reduction, the type of an expression may change to a subtype due to subtyping. However, in **FHJ**, the type remains unchanged because we keep track of the static types and use them for casting during reduction.

Finally we show that one-step evaluation is deterministic. This theorem is helpful to show that our model of multiple inheritance is not ambiguous (or non-deterministic).

► **Theorem 4 (Determinacy of One-Step Evaluation).** *If $t \rightarrow t'$ and $t \rightarrow t''$, then $t' = t''$.*

Proof. See Appendix A.1. ◀

5 Discussion

In this section, we will discuss the design space and reflect about some of the design decisions of our work. We relate our language to traits, Java interfaces as well as other languages. Furthermore, we discuss ways to improve our work.

5.1 Abstract Methods

Abstract methods are one of the key features in most general OO languages. For example, Java interfaces (prior to Java 8) were designed to include only method declarations, and those abstract methods can be implemented in a class body. The formal Featherweight Java model [14] does not include abstract methods because of the orthogonality to the core calculus. In traits, a similar idea is to use keywords like “**require**” for abstract method declarations [28]. Abstract methods provide a way to delay the implementations to future subtypes. Using overriding, they also help to “exclude” existing implementations.

In our formalized calculus, however, abstract methods are not a completely orthogonal feature. The `canInstantiate` function has to check whether an interface can be instantiated by looking at all the inherited branches and checking if each most specific method is concrete or not.

Our formalization has a simple form of abstract methods, which behave similarly to conventional methods with respect to conflicts. Other languages may behave differently. For instance, in Java 8 when putting two identical abstract methods together by multiple inheritance, there is no conflict error. In Figure 7, we use italic m to denote abstract methods. In both cases, the Java compiler accepts the definition of C and automatically merges the two inherited methods m into a single one. **FHJ** behaves differently from Java in both cases. In the fork inheritance case (left), C will have two distinct abstract methods corresponding to $A.m$ and $B.m$. In the diamond inheritance case, the definition of C is rejected. There are two reasons for this difference in behaviour. Firstly, our formalization just treats abstract



■ **Figure 7** Fork inheritance (left) and diamond inheritance (right) on abstract methods.

methods as concrete methods with an empty body, and that simplifies the rules and proofs a lot. Secondly, and more importantly, we distinguish and treat differently conflicting methods, since they may represent different operations, even if they are abstract. Thus our model adopts a very conservative behavior rather than automatically merging methods by default (as done in many languages). Arguably, the diamond case it is actually an intentional conflict due to the same source T . Therefore our model conservatively rejects this case. It is possible to change our model to account for other behaviors for abstract methods, but we view this as a mostly orthogonal change to our work, and should not affect the essence of the model presented here.

5.2 Orthogonal & Non-Orthogonal Extensions

Our model is designed as a minimal calculus that focuses on resolving unintentional conflicts. Therefore, we have omitted a number of common orthogonal features including primitive types, assignments, method overloading, covariant method return types, static dispatch, and so on. Those features can, in principle, be modularly added to the model without breaking type soundness. For example, we present the additional syntax, typing and semantic rules of static invocation below as an extension:

$$\begin{aligned} \text{Expressions } e & ::= \dots \mid e.J_0@J_1 :: m(\bar{e}) \\ \text{(T-STATICINVK)} & \frac{J_0[m \text{ override } J_1] = I \ m(\bar{J} \ \bar{x}) \ \text{override } J_1 \ \{\text{return } e;\} \quad \Gamma \vdash e_0 : I_0 \quad I_0 <: J_0 \quad \Gamma \vdash \bar{e} : \bar{I} \quad \bar{I} <: \bar{J}}{\Gamma \vdash e_0.J_0@J_1 :: m(\bar{e}) : I} \\ \text{(S-STATICINVK)} & \frac{J_0[m \text{ override } J_1] = I_e \ m(\bar{I}_x \ \bar{x}) \ \text{override } J_1 \ \{\text{return } e_0;\} \quad ((J) \text{new } I()) . J_0@J_1 :: m(\bar{v}) \rightarrow (I_e)[\bar{I}_x] \bar{v} / \bar{x}, (J_0) \text{new } I() / \text{this} e_0}{((J) \text{new } I()) . J_0@J_1 :: m(\bar{v}) \rightarrow (I_e)[\bar{I}_x] \bar{v} / \bar{x}, (J_0) \text{new } I() / \text{this} e_0} \end{aligned}$$

A static invocation $e.J_0@J_1 :: m(\bar{e})$ aims at finding the method m in J_0 that hierarchically overrides J_1 , thus $J_0[m \text{ override } J_1]$ is invoked. As shown in (S-STATICINVK), static dispatch needs a receiver for the substitution of the “this” reference, so as to provide the latest implementations. In fact, static dispatch is common in OO programming, as it provides a shortcut to the reuse of old implementation easily, and super calls can also rely on this feature. For convenience we just make it simple above, whereas in languages like C++ or Java, the static or super invocations are more flexible, as they can climb the class hierarchy.

One non-orthogonal extension to **FHJ** could be to generalize the model to allow multiple hierarchical method overriding, meaning that, we allow overriding methods to update multiple branches instead of only one branch. This feature offers a more fine-grained mechanism for merging and can be helpful to easily understand the structure of the hierarchy. Multiple overriding would be useful in the following situation, for example:

```

interface A { void m() {...} }
interface B { void m() {...} }
interface C { void m() {...} }
interface D extends A, B, C {
  void m() override A,B {...} // overrides branches A and B only
  void m() override C {...} // overrides branch C
}

```

Here D inherits from three interfaces A, B, C with conflicting methods m, but only merges two of those methods. While we can simulate D without multiple overriding in our calculus (by introducing an intermediate class), a better approach would be to support multiple overriding natively.

We present the modification of syntax, typing and semantic rules below (abstract methods omitted):

$$\begin{array}{c}
 \text{Methods } M ::= \dots \mid I \text{ m}(\overline{I_x} \overline{x}) \text{ override } \overline{J} \{ \text{return } e; \} \\
 \\
 \text{(T-MOMETHOD)} \frac{\forall J_i \in \overline{J}, I <: J_i \quad \text{findOrigin}(m, I, J_i) = \{J_i\} \\
 \text{mbody}(m, J_i, J_i) = (K, \overline{I_x} \overline{x}, I_e _) \quad \overline{x} : \overline{I_x}, \text{this} : I \vdash e_0 : I_0 \quad I_0 <: I_e}{I_e \text{ m}(\overline{I_x} \overline{x}) \text{ override } \overline{J} \{ \text{return } e_0; \} \text{ OK IN } I}
 \end{array}$$

Semantic rules themselves remain unchanged, however, we need to change slightly the definition of `findOverride` in `mbody`:

▷ *Definition of* `findOverride(m, I, J)` :

- `findOverride(m, I, J) = prune(overrides)`

with: `overrides = {K | I <: K, K <: J and K[m override \overline{J}]} where $J \in \overline{J}$`

With this approach, branches A and B are merged in the sense that they share the same code, which can be separately updated in future interfaces. Another approach would be to deeply merge the branches, with similar effect as introducing an intermediate interface **AB** to explicitly merge the two branches. However, this approach is problematic because there is no clear mechanism for identifying and further updating the merged branches. This could be an interesting future work to explore.

Other typical non-orthogonal extensions to **FHJ** could be to have fields. The design of **FHJ** can be viewed as a variant of Java 8 with default methods which allows for unintentional method conflicts. Like Java interfaces and traits, state is forbidden in **FHJ**. There are some inheritance models that also account for fields, such as C++ that uses virtual inheritance [10]. In our model, however, we can perhaps borrow the idea of *interface-based programming* [33], which models state with abstract state operations. This can be realized by extending our current model with static methods and anonymous classes from Java. However such an extension requires more thought, so we leave it to future work.

5.3 Loosening the Model: Reject Early or Reject Later?

FHJ rejects the following case of diamond inheritance:

```

interface A { void m() {...} }
interface B extends A { void m() {...} }
interface C extends A { void m() {...} }
interface D extends B, C {}

```

Here both $B.m$ and $C.m$ override $A.m$, and D inherits both conflicting methods without an explicit override. In this case, automatically merging the two methods (to achieve diamond inheritance) is not possible, which is why many models (like traits and Java 8) reject such programs. Moreover, keeping the two method implementations in D is problematic. In essence, hierarchical information is not helpful to disambiguate later method calls, since the two methods share the same origin ($A.m$). Our calculus rejects such conflicts by the (T-INTF) rule, where D is considered to be ill-formed. We believe that rejecting D follows the principle of models like traits and Java 8 interfaces, where the language/type-system is meant to alert the programmer for a possible conflict early.

Nonetheless, C++ accepts the definition of D , but forbids later upcasts from D to A because of ambiguity. Our language is more conservative on definitions of interfaces compared to C++, but on the upside, upcasts are not rejected. We could also loosen the model to accept definitions such as D , and perform ambiguity check on upcasts and other expressions. Then, we would need to handle more cases than C++ because of the complication caused by the hierarchical overriding feature.

6 Related Work

We describe related work in four parts. We first discuss mainstream popular multiple inheritance models and then some specific models (e.g., C++ and C#) which are closest to our work. Then we discuss related techniques used in **SELF**. Finally, we discuss the foundation and related work of our formalization.

6.1 Mainstream Multiple Inheritance Models

Multiple inheritance is a useful feature in object-oriented programming, although it is difficult to model and can cause various problems (e.g. the diamond problem [27, 29]). There are many existing languages/models that support multiple inheritance [10, 22, 5, 28, 17, 19, 20, 11, 2]. The mixin models [5, 11, 32, 2, 13] allow naming components that can be applied to various classes as reusable functionality units. However, the linearization (total ordering) of mixin inheritance cannot provide a satisfactory resolution in some cases and restricts the flexibility of mixin composition. Scala traits [22] are in fact linearized mixins and hence have the same problem as mixins.

Simplifying the mixins approach, traits [28, 9] draw a strong line between units of reuse and object factories. Traits act as units of reuse, containing functionality code. Classes, on the other hand, are assembled from traits and act as object factories. Java 8 interfaces are closely related to traits: concrete method implementations are allowed (via the **default** keyword) inside interfaces, thus allowing for a restricted form of multiple inheritance. There are also proposals such as FeatherTrait Java [16] for extending Java with traits. Extensions [25, 26] to the original trait model exists with various advanced features, such as *renaming*. As discussed in Section 2, the renaming feature gives a workaround to the unintentional method conflicts problem. However, it breaks structural subtyping.

Malayeri and Aldrich proposed a model CZ [17] which aims to do multiple inheritance without the diamond problem. Inheritance is divided into two concepts: inheritance dependency and implementation inheritance. Using a combination of **requires** and **extends**, a program with diamond inheritance is transformed to one without diamonds. Moreover, fields and multiple inheritance can coexist. However untangling inheritance also untangles the class structure. In CZ, not only the number of classes but also the class hierarchy complexity increases.

The above-mentioned models/languages support multiple inheritance, focusing on diamond inheritance. They handle method conflicts in the same way, by simply disallowing two methods with the same signature from two different units to coexist. In contrast, our work provides mechanisms that allow methods with the same signatures, but different parents to coexist in a class. Disambiguation is possible in many cases by using both static and dynamic type information during method dispatching. In the cases where real ambiguity exists, **FHJ**'s type system can reject interface definitions and/or method calls statically.

6.2 Resolving Unintentional Method Conflicts

A few language implementations have realized the problem of unintentional conflicts and provide some support for it.

C++ model. C++ supports a very flexible inheritance model. C++ allows the existence of unintentional conflicts and users may specify a hierarchical path via casts for disambiguation, as discussed in Section 2. With virtual methods, dynamic dispatch is used and the method lookup algorithm will find the most specific method definition. A contribution of our work is to provide a minimal formal model of hierarchical dispatching, whereas C++ can be viewed as a real-world implementation. There are several formalizations [34, 24, 23] in the literature modeling various C++ features. However, as far as we know, there is no formal model that captures this aspect of the C++ method dispatching model. Apart from this, as discussed in Section 5.3, **FHJ** conservatively rejects some interface/class definitions that C++ accepts, and upcasts are never rejected since the ambiguity is prevented beforehand.

Although C++ supports hierarchical dispatching, it does not support hierarchical overriding. However, there are some possible workarounds that can mimic hierarchical overriding, including the *MiddleMan* approach⁴, the *interface classes* pattern as described in Section 25.6 of [31], the *LotterySimulation* discussion in [30]. Since these workarounds share the same spirit, we will discuss in detail the *MiddleMan* approach, with the code shown in Figure 8. In this example, classes A and B are two classes that both define a method with the same name `m` unintentionally.

Class `MiddleMan`, as its name suggests, acts as a middleman between its class `C` and its parents `A`, `B`. `MiddleMan` defines a virtual method `m` that overrides a parent method `m` and delegates the implementation to another method `m_impl` that takes `this` as a parameter. C++ supports method overloading, so that multiple `m_impl` methods with different parameter types can coexist. When defining class `C`, we specify the parents to be `MiddleMan<A>`, `MiddleMan` instead of `A`, `B`. In this way, programmers may define new versions of `A.m` and `B.m` in class `C` by providing the corresponding `m_impl` methods. Then in the client code, the method call `((A*)c)->m()` will print out the string "MA2", as expected. Although this workaround can help us defining partial method overrides to a certain extent, the drawbacks are obvious. Firstly, the approach is complex and requires the programmer to fully understand this approach. Moreover, the lack of direct syntax support makes `MiddleMan` code cumbersome to write. Finally, the approach is ad-hoc, meaning that the class `MiddleMan` shown in Figure 8 is not general enough to be used in other cases: more middlemen are needed if partial method overrides happen in other classes; and it is even worse when return types differ.

⁴ <https://stackoverflow.com/questions/44632250/can-i-do-mimic-things-like-this-partial-override-in-c>


```

class A { public: virtual void m() {cout << "MA" << endl;}};
class B { public: virtual void m() {cout << "MB" << endl;}};
template<class C>
class MiddleMan : public C {
    void m() override final { m_impl(this); }
protected:
    virtual void m_impl(MiddleMan*) { return this->C::m(); }
};
class C : public MiddleMan<A>, public MiddleMan<B> {
private:
    void m_impl (MiddleMan<A>*) override {cout << "MA2" << endl;}
    void m_impl (MiddleMan<B>*) override {cout << "MB2" << endl;}
};
int main()
{
    C* c = new C();
    ((A*)c)->m();    //print "MA2"
    return 0;
}

```

■ **Figure 8** The *MiddleMan* approach.

C# explicit method implementations. Explicit method implementations is a special feature supported by C#. As described in C# documentation [19], a class that implements an interface can explicitly implement a member of that interface. When a member is explicitly implemented, it can only be accessed through an instance of the interface. Explicit interface implementations allow an interface to inherit multiple interfaces that share the same member names and give each interface member a separate implementation.

Explicit interface member implementations have two advantages. Firstly, they allow interface implementations to be excluded from the public interface of a class. This is particularly useful when a class implements an internal interface that is of no interest to a consumer of that class or struct. Secondly, they allow disambiguation of interface members with the same signature. However, there are two critical differences to **FHJ**: (1) default method implementations are not allowed in C# interfaces; (2) there is only one level of conflicting method implementations at the class that implements the multiple parent interfaces. Further overriding of those methods is not possible in subclasses.

Languages using hygienicity. In NextGen/MixGen [1], HygJava [15] and Magda [3], *hygienicity* is proposed to deal with unintentional method conflicts. The idea is to give a method a unique identifier by prefixing the name with an unambiguous path. As shown in Figure 9, the prefix `HelloWorld` in the method call (`new HelloWorld []`).`HelloWorld.MainMatter()` is mandatory. So writing programs in these languages is tedious if not supported by a specialized IDE, that aids filling prefix/method information. The advantage of this approach, compared to ours, is that it does not require any additional notion for method dispatching. Indeed the compilation strategy is simple, just by generating conventional code (say in Java or C++) with method names attached with prefixes. Unfortunately, the disadvantage is that some expressive power is lost. In particular *merging* methods arising from diamond inheritance is not possible because the methods have different prefixes. As shown in Figure 10, two methods `m` from different branches `A` and `B` cannot be overridden by the method `m` in `C` because they are regarded as unrelated methods, and `m` in `C` is just another new method that has nothing to do with `A.m` or `B.m`. The reason is that in these hygienic approaches, path names are used to

```

mixin HelloWorld of Object =
  new Object MainMatter()
  begin
    "Hello world".String.print();
  end;
end;
(new HelloWorld []).HelloWorld.MainMatter();

```

■ **Figure 9** Full-qualified name of method calls in Magda.

```

mixin A of Object =
  new String m()
  begin
    return "A";
  end;
end;
mixin B of Object =
  new String m()
  begin
    return "B";
  end;
end;
mixin C of A, B =
  new String m()
  begin
    return "C";
  end;
end;
end;

```

■ **Figure 10** Code in Magda.

distinguish different methods. In contrast, our model can deal with unintentional conflicts, as well as merged methods because our semantics is not simply based on prefixing. Instead, our model keeps the names of methods unchanged, and our direct operational semantics takes static and dynamic type information into account at runtime when doing method dispatching. Finally, the multiple inheritance model in Magda is based on Mixins, whereas **FHJ** is based on traits. Thus, Magda inherits all limitations of Mixins (such as the linearization problem, etc).

6.3 Hierarchical Dispatch in SELF

As we have discussed before, although the mix of static and dynamic dispatch is particularly useful under certain circumstances, it has received little research attention. In the prototype-based language **SELF** [6], inheritance is a basic feature. **SELF** does not include classes but instead allows individual objects to inherit from (or delegate to) other objects. Although it is different from class-based languages, the multiple inheritance model is somewhat similar. The **SELF** language supports multiple (object) inheritance in a clever way. It not only develops the new inheritance relation with *prioritized parents* but also adopts *sender path tiebreaker rule* for method lookup. In **SELF** “*if two slots with the same name are defined in equal-priority parents of the receiver, but only one of the parents is an ancestor or descendant of the object containing the method that is sending the message, then that parent’s slot takes precedence over the over parent’s slot*”. Similarly to our model, this sender path tiebreaker rule resolves ambiguities between unrelated slots. However, it is used in a prototype-based language setting and it does not support method hierarchical overriding as **FHJ** does.

6.4 Formalization Based on Featherweight Java

Featherweight Java (FJ) [14] is a minimal core calculus of the Java language, proposed by Igarashi et. al. There are many models built on Featherweight Java, including Feather-Trait [16], Featherweight defenders [12], Jx [21], Featherweight Scala [8], and so on. FJ provides the standard model of formalizing Java-like object-oriented languages and is easily extensible. In terms of formalization, the key novelty of our model is making use of various types (such as parameter types, method return types, etc) to track the static types as well as the dynamic types during reduction. As far as we know, this technique has not appeared in the literature before. This notion is of vital importance in our hierarchical dispatch algorithm, and it allows for a more precise subject-reduction theorem as discussed in Section 3.

7 Conclusion

This paper proposes **FHJ** as a formalized multiple inheritance model for unintentional method conflicts. Previous approaches either do not support unintentional method conflicts, thus have to compromise between code reuse and type safety, or do not fully support overriding in the presence of unintentional conflicts. To deal with unintentional method conflicts we introduce two key mechanisms: hierarchical dispatching and hierarchical overriding. Hierarchical dispatching is inspired by the mechanisms in C++. We provide a minimal formal model of hierarchical dispatching in **FHJ**. Such an algorithm makes use of both dynamic type information and static information from either upcasts or parameters' information. It not only offers code reuse and dynamic dispatch, but also ensures unambiguity using our hierarchical dispatching algorithm for method resolution. Additionally we introduce *hierarchical overriding* to allow conflicting methods in different branches to be individually overridden.

FHJ is formalized following the style of Featherweight Java and proved to be sound. A prototype interpreter is implemented in Scala. We believe that the formalization of hierarchical dispatching features is general and can be safely embedded in other OO models, so as to have support for the fork inheritance.

Our model can certainly be improved in some aspects. As discussed in Section 5, there are orthogonal and non-orthogonal features that can potentially be added to the design space. The future work relates to loosening the model without giving up its soundness, together with more exploration on supporting fields in the multiple inheritance setting.

References

- 1 Eric Allen, Jonathan Bannet, and Robert Cartwright. A first-class approach to genericity. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '03, pages 96–114, New York, NY, USA, 2003. ACM. doi:10.1145/949305.949316.
- 2 Davide Ancona, Giovanni Lagorio, and Elena Zucca. Jam—designing a java extension with mixins. *ACM Trans. Program. Lang. Syst.*, 25(5):641–712, 2003.
- 3 Viviana Bono, Jarek Kuśmierk, and Mauro Mulaturo. Magda: A new language for modularity. In *Proceedings of the 26th European Conference on Object-Oriented Programming*, ECOOP'12, pages 560–588, Berlin, Heidelberg, 2012. Springer-Verlag. doi:10.1007/978-3-642-31057-7_25.
- 4 Viviana Bono, Enrico Mensa, and Marco Naddeo. Trait-oriented programming in java 8. In *PPPJ '14*, 2014.
- 5 Gilad Bracha and William Cook. Mixin-based inheritance. In *OOPSLA/ECOOP '90*, 1990.

- 6 Craig Chambers, David Ungar, Bay-Wei Chang, and Urs Hölzle. Parents are shared parts of objects: Inheritance and encapsulation in self. *Lisp Symb. Comput.*, 4(3), 1991.
- 7 Steve Cook. Varieties of inheritance. In *OOPSLA '87 Panel P2*, 1987.
- 8 Vincent Cremet, François Garillot, Sergueï Lenglet, and Martin Odersky. A core calculus for scala type checking. In *MFCSS '06*, 2006.
- 9 Stéphane Ducasse, Oscar Nierstrasz, Nathanael Schärli, Roel Wuyts, and Andrew P. Black. Traits: A mechanism for fine-grained reuse. *ACM Trans. Program. Lang. Syst.*, 28(2):331–388, 2006.
- 10 Margaret A Ellis and Bjarne Stroustrup. *The annotated C++ reference manual*. Addison-Wesley, 1990.
- 11 Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *POPL '98*, 1998.
- 12 Brian Goetz and Robert Field. Featherweight defenders: A formal model for virtual extension methods in java. <http://cr.openjdk.java.net/~briangoetz/lambda/featherweight-defenders.pdf>, 2012.
- 13 James Hendler. Enhancement for multiple-inheritance. In *OOPWORK '86*, 1986.
- 14 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.
- 15 Jaroslaw DM Kusmieriek and Viviana Bono. Hygienic methods - introducing hygjava. *Journal of Object Technology*, 6(9):209–229, 2007.
- 16 Luigi Liquori and Arnaud Spiwack. Feathertrait: A modest extension of featherweight java. *ACM Trans. Program. Lang. Syst.*, 30(2):11, 2008.
- 17 Donna Malayeri and Jonathan Aldrich. Cz: Multiple inheritance without diamonds. In *OOPSLA '09*, 2009.
- 18 B Meyer. Eiffel: Programming for reusability and extendibility. *SIGPLAN Not.*, 22(2), 1987.
- 19 Microsoft. Csharp explicit interface member implementations document. [https://msdn.microsoft.com/en-us/library/aa664591\(v=vs.71\).aspx](https://msdn.microsoft.com/en-us/library/aa664591(v=vs.71).aspx), 2003.
- 20 David A. Moon. Object-oriented programming with flavors. In *OOPSLA '86*, 1986.
- 21 Nathaniel Nystrom, Stephen Chong, and Andrew C. Myers. Scalable extensibility via nested inheritance. In *OOPSLA '04*, 2004.
- 22 Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An overview of the scala programming language. Technical report, École Polytechnique Fédérale de Lausanne 1015 Lausanne, Switzerland, 2004.
- 23 G. Ramalingam and Harini Srinivasan. A member lookup algorithm for c++. In *PLDI '97*, 1997.
- 24 Tahina Ramananandro. *Mechanized Formal Semantics and Verified Compilation for C++ Objects*. PhD thesis, Université Paris-Diderot-Paris VII, 2012.
- 25 John Reppy and Aaron Turon. A foundation for trait-based metaprogramming. In *FOOL/WOOD '06*, 2006.
- 26 John Reppy and Aaron Turon. Metaprogramming with traits. In *ECOOP '07*, 2007.
- 27 Markku Sakkinen. Disciplined inheritance. In *ECOOP '89*, 1989.
- 28 Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P. Black. Traits: Composable units of behaviour. In *ECOOP '03*, 2003.
- 29 Ghan Bir Singh. Single versus multiple inheritance in object oriented programming. *SIGPLAN OOPS Mess.*, 1995.
- 30 Bjarne Stroustrup. *The design and evolution of C++*. Pearson Education India, 1994.
- 31 Bjarne Stroustrup. *The C++ programming language*. Pearson Education India, 1995.

- 32 Marc Van Limberghen and Tom Mens. Encapsulation and composition as orthogonal operators on mixins: A solution to multiple inheritance problems. *Object Oriented Systems*, 3(1):1–30, 1996.
- 33 Yanlin Wang, Haoyuan Zhang, Bruno C. d. S. Oliveira, and Marco Servetto. Classless java. In *GPCE '16*, 2016.
- 34 Daniel Wasserrab, Tobias Nipkow, Gregor Snelting, and Frank Tip. An operational semantics and type safety proof for multiple inheritance in c++. In *OOPSLA '06*, 2006.
- 35 A.K. Wright and M. Felleisen. A syntactic approach to type soundness. *Inf. Comput.*, 115(1):38–94, 1994.
- 36 Matthias Zenger and Martin Odersky. Independently extensible solutions to the expression problem. In *FOOL '05*, 2005.

A

 Appendix

A.1 Proofs

► **Lemma 5.** *If $\text{mbody}(m, I_d, I_s) = (J, \overline{I_x} \bar{x}, I_e e_0)$, then $\bar{x} : \overline{I_x}$, $\text{this} : J \vdash e_0 : I_0$ for some $I_0 <: I_e$.*

Proof. By the definition of `mbody`, the target method `m` is found in `J`. By the method typing rule (T-METHOD), there exists some $I_0 <: I_e$ such that $\bar{x} : \overline{I_x}$, $\text{this} : J \vdash e_0 : I_0$. ◀

► **Lemma 6 (Weakening).** *If $\Gamma \vdash e : I$, then $\Gamma, x : J \vdash e : I$.*

Proof. Straightforward induction. ◀

► **Lemma 7 (Method Type Preservation).** *If $\text{mbody}(m, J, J) = (K, \overline{I_x} _, I_e _)$, then for any $I <: J$, $\text{mbody}(m, I, J) = (K', \overline{I_x} _, I_e _)$.*

Proof. Since `mbody`(`m`, `J`, `J`) is defined, by (T-INTF) we derive that `mbody`(`m`, `I`, `J`) is also defined. Suppose that

$$\text{findOrigin}(m, J, J) = \{I_0\}$$

$$\text{findOverride}(m, J, I_0) = \{K\}$$

$$\text{findOrigin}(m, I, J) = \{I'_0\}$$

$$\text{findOverride}(m, I, I'_0) = \{K'\}$$

Below we use $I[m \uparrow J]$ to denote the type of method `m` defined in `I` that overrides `J`. We have to prove that $K'[m \uparrow I'_0] = K[m \uparrow I_0]$. Two facts:

- A. By (T-INTF), `canOverride` ensures that an override between any two original methods preserves the method type. Formally,

$$I_1 <: I_2 \Rightarrow I_1[m \uparrow I_1] = I_2[m \uparrow I_2]$$

- B. By (T-METHOD) and (T-ABSMETHOD), any partial override also preserves method type. Formally,

$$I_1 <: I_2 \Rightarrow I_1[m \uparrow I_2] = I_2[m \uparrow I_2]$$

By definition of `findOverride`, $K <: I_0, K' <: I'_0$. By Fact B,

$$K[m \uparrow I_0] = I_0[m \uparrow I_0] \quad K'[m \uparrow I'_0] = I'_0[m \uparrow I'_0]$$

Hence it suffices to prove that $I'_0[m \uparrow I'_0] = I_0[m \uparrow I_0]$. Actually when calculating `findOrigin(m, J, J)`, by the definition of `findOrigin` we know that $I_0 <: J$ and $I_0[m \text{ override } I_0]$ is defined. So when calculating `findOrigin(m, I, J)` with $I <: J$, I_0 should also appear in the set before pruned, since the conditions are again satisfied. But after pruning, only I'_0 is obtained, by definition of `prune` it implies $I'_0 <: I_0$. By Fact A, the proof is done. \blacktriangleleft

► **Lemma 8** (Term Substitution Preserves Typing). *If $\Gamma, \bar{x} : \bar{I}_x \vdash e : I$, and $\Gamma \vdash \bar{y} : \bar{I}_x$, then $\Gamma \vdash [\bar{y}/\bar{x}]e : I$.*

Proof. We prove by induction. The expression e has the following cases:

Case Var. Let $e = x$. If $x \notin \bar{x}$, then the substitution does not change anything. Otherwise, since \bar{y} have the same types as \bar{x} , it immediately finishes the case.

Case Invk. Let $e = e_0.m(\bar{e})$. By (T-INVK) we can suppose that

$$\Gamma, \bar{x} : \bar{I}_x \vdash e_0 : I_0 \quad \text{mbody}(m, I_0, I_0) = (_, \bar{J} _, I _)$$

$$\Gamma, \bar{x} : \bar{I}_x \vdash \bar{e} : \bar{I}_e \quad \bar{I}_e <: \bar{J} \quad \Gamma, \bar{x} : \bar{I}_x \vdash e : I$$

By induction hypothesis,

$$\Gamma \vdash [\bar{y}/\bar{x}]e_0 : I_0 \quad \Gamma \vdash [\bar{y}/\bar{x}]\bar{e} : \bar{I}_e$$

Again by (T-INVK), $\Gamma \vdash [\bar{y}/\bar{x}]e : I$.

Case New. Straightforward.

Case Anno. Straightforward by induction hypothesis and (T-ANNO). \blacktriangleleft

A.1.1 Proof for Theorem 1

Proof.

Case S-Invk. Let

$$e = ((J)\text{new } I()).m(\bar{v}) \quad \Gamma \vdash e : I_e$$

$$e' = (I_{e_0})[(\bar{I}_x)\bar{v}/\bar{x}, (I_0)\text{new } I()/\text{this}]e_0$$

$$\text{mbody}(m, I, J) = (I_0, \bar{I}_x \bar{x}, I_{e_0} e_0)$$

Since `mbody(m, I, J)` is defined, the definition of `mbody` ensures that $I <: J$. And since e is well-typed, by (T-INVK),

$$\Gamma \vdash \bar{v} : \bar{I}_v \quad \bar{I}_v <: \bar{I}_x$$

By the rules (T-ANNO) and (T-NEW),

$$\Gamma \vdash (\bar{I}_x)\bar{v} : \bar{I}_x \quad \Gamma \vdash (I_0)\text{new } I() : I_0$$

On the other hand, by Lemma 5,

$$\bar{x} : \bar{I}_x, \text{this} : I_0 \vdash e_0 : I'_{e_0} \quad I'_{e_0} <: I_{e_0}$$

By Lemma 6,

$$\Gamma, \bar{x} : \bar{I}_x, \text{this} : I_0 \vdash e_0 : I'_{e_0}$$

Hence by Lemma 8, the substitution preserves typing, thus

$$\Gamma \vdash [(\bar{I}_x)\bar{v}/\bar{x}, (I_0)\text{new } I()/\text{this}]e_0 : I'_{e_0}$$

Since $I'_{e_0} <: I_{e_0}$, the conditions of (T-ANNO) are satisfied, hence $\Gamma \vdash e' : I_{e_0}$. Now we only need to prove that $I_{e_0} = I_e$. Since I_{e_0} is from $\text{mbody}(m, I, J)$, whereas I_e is from $\text{mbody}(m, J, J)$, by the rule (T-INVK) on e . Since $I <: J$, by Lemma 7, $I_{e_0} = I_e$.

Case C-Receiver. Straightforward induction.

Case C-Args. Straightforward induction.

Case C-StaticType. Immediate by (T-ANNO).

Case C-FReduce. Immediate by (T-ANNO) and induction.

Case C-AnnoReduce. Immediate by (T-ANNO) and transitivity of $<:$. ◀

A.1.2 Proof for Theorem 2

Proof. Since e is well-typed, by (T-INVK) and (T-ANNO) we know that

$$I <: J, \text{ and } \text{mbody}(m, J, J) \text{ is defined}$$

By (T-INTF), $\text{mbody}(m, I, J)$ is also defined, and the type checker ensures the expected number of arguments.

On the other hand, since $I <: J$, by the definition of `findOrigin`,

$$\text{findOrigin}(m, I, J) \subseteq \text{findOrigin}(m, I, I)$$

By (T-NEW), `canInstantiate(I) = True`. By the definition of `canInstantiate`, any $J_0 \in \text{findOrigin}(m, I, I)$ satisfies that `findOverride(m, I, J0)` contains only one interface, in which the m that overrides J_0 is a concrete method. Therefore $\text{mbody}(m, I, J)$ also provides a concrete method, which finishes the proof. ◀

A.1.3 Proof for Theorem 4

Proof. The Proof is done by induction on a derivation of $t \rightarrow t'$, following the book *TAPL*.


- If the last rule used in the derivation of $t \rightarrow t'$ is (S-INVK), then we know that t has the form $((J)\text{new } I()).m(\bar{v})$ with I, J, m determined. Now it is obvious that the last rule in the derivation of $t \rightarrow t''$ should also be (S-INVK) with the same I, J, m . Since $\text{mbody}(m, I, J)$ is a *function* that given the same input will calculate the same result, we know the two induction results are the same, thus $t' = t''$ is immediately proved.
- If the last rule used in the derivation of $t \rightarrow t'$ is (C-RECEIVER), then t has the form $e_0.m(\bar{e})$ and $e_0 \rightarrow e'_0$. Since e_0 is not a value, the last rule used in $t \rightarrow t''$ has to be (C-RECEIVER) (other rules do not match) too. Assume in the reduction $t \rightarrow t''$, $e_0 \rightarrow e''_0$, thus $e'_0.m(\bar{e}) = e''_0.m(\bar{e})$. Thus, $t' = t''$ proved.
- If the last rule used in the derivation of $t \rightarrow t'$ is (C-STATICTYPE), then t is fixed to be `new I()`. The last rule used in $t \rightarrow t''$ has to be (C-STATICTYPE), and obviously, $t' = t'' = (I)\text{new } I()$.

- If the last rule used in the derivation of $t \rightarrow t'$ is (C-FREDUCE), then t has the form $(I)e$ and $e \rightarrow e'$. The last rule used in $t \rightarrow t''$ cannot be (C-STATICTYPE) because it requires t to be $\text{new } I()$; it can neither be (C-ANNOREDUCE) because it requires t to be $(I)((J)\text{new } K())$ where $(J)\text{new } K()$ is already a value. So the last rule used in $t \rightarrow t''$ can only be (C-FREDUCE) (other rules do not match). Assume in the reduction $t \rightarrow t''$, $e \rightarrow e''$, and $(I)e \rightarrow (I)e''$. By induction hypothesis, $e' = e''$, thus $t' = t''$ proved.
- If the last rule used in the derivation of $t \rightarrow t'$ is (C-ANNOREDUCE), then the form of t is fixed to be $(I)((J)\text{new } K())$. Since $(I)((J)\text{new } K())$ is not reducible, the rule (C-FREDUCE) does not apply. The only rule applies in $t \rightarrow t''$ is (C-ANNOREDUCE). Thus $t' = t'' = (I)\text{new } K()$ proved.
- If the last rule used in the derivation of $t \rightarrow t'$ is (C-ARGS), then t has the form $v.m(\dots, e, \dots)$ and $e \rightarrow e'$. The last rule used in $t \rightarrow t''$ cannot be (S-INVK) because it requires all arguments to be values. Thus only (C-ARGS) applies to $t \rightarrow t''$. Assume in the reduction $t \rightarrow t''$, $e \rightarrow e''$. By induction hypothesis, $e' = e''$, thus $v.m(\dots, e', \dots) = v.m(\dots, e'', \dots)$, thus $t' = t''$ proved. ◀

Modeling Infinite Behaviour by Corules


Davide Ancona¹

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

 <https://orcid.org/0000-0002-6297-2011>


Francesco Dagnino

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

 <https://orcid.org/0000-0003-3599-3535>

Elena Zucca

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

 <https://orcid.org/0000-0002-6833-6470>

Abstract

Generalized inference systems have been recently introduced, and used, among other applications, to define semantic judgments which uniformly model terminating computations and divergence. We show that the approach can be successfully extended to more sophisticated notions of infinite behaviour, that is, to express that a diverging computation produces some possibly infinite result. This also provides a motivation to smoothly extend the theory of generalized inference systems to include, besides *coaxioms*, also *corules*, a more general notion for which significant examples were missing until now. We first illustrate the approach on a λ -calculus with output effects, for which we also provide an alternative semantics based on standard notions, and a complete proof of the equivalence of the two semantics. Then, we consider a more involved example, that is, an imperative Java-like language with I/O primitives.

2012 ACM Subject Classification Theory of computation \rightarrow Operational semantics

Keywords and phrases Operational semantics, coinduction, trace semantics

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2018.21

1 Introduction

In semantic definitions of programming languages or systems, *finite* behaviour can be easily modeled by inductive techniques. For instance, both in small-step and big-step operational semantics, the fact that evaluation of an expression e terminates producing a final result r can be formally expressed as a judgment $e \Rightarrow r$ defined by an inference system², so that each valid judgment has a finite proof tree.

However, modeling *infinite* behaviour is not so easy. The simplest infinite behaviour we may want to model is divergence in itself, that is, the fact that evaluation of e does not terminate, as can be formally expressed by introducing a special result ∞ . Traditionally, this is modeled at the meta-level in small-step definitions: the inference system does *not* define the judgment $e \Rightarrow \infty$, but only single steps, and then we formalize divergence simply as “an

¹ Member of GNCS (Gruppo Nazionale per il Calcolo Scientifico), INdAM (Istituto Nazionale di Alta Matematica "F. Severi")

² In small-step style this judgment can be inductively defined on top of the one-step reduction relation.



infinite sequence of steps”. In big-step semantics the situation is even worse: non terminating and stuck computations are identified, since in both cases it is not possible to construct a finite proof tree showing that the computation returns a result.

A simple solution, in both approaches, is to define $e \Rightarrow \infty$ by a separate inference system, where inference rules are interpreted coinductively [8, 13]. However, in this way there are two stratified systems with overlapping rules, and there is no unique formal definition of the behaviour. The possibility of providing such unique definition has been investigated in previous work [13, 3], by interpreting coinductively the standard rules for the big-step operational semantics (*coevaluation*). Unfortunately, this approach is not satisfactory; for instance, coevaluation is non-deterministic in some cases: for a diverging term such as $\Omega = (\lambda x.x x)(\lambda x.x x)$, $\Omega \Rightarrow r$ can be derived for any r , instead of the unique valid judgment $\Omega \Rightarrow \infty$. Moreover, for some other diverging term, e.g., $\Omega (0 0)$, no judgment can be derived, since there is no valid judgment for the subterm $0 0$. This happens because in an application $e_1 e_2$, if we follow a left-to-right strategy, then divergence of e_1 should be *propagated* regardless of e_2 , but this cannot be properly modeled due to the presence of spurious judgments $\Omega \Rightarrow r$. In some recent work [6], it has been shown that both problems can be solved by defining the judgment $e \Rightarrow r$ by a *generalized inference system* [5]. This semantics can be seen as a *filtered coevaluation*. Indeed, infinite proof trees are allowed, but appropriate *coaxioms* are introduced to filter out spurious judgments $\Omega \Rightarrow r$, so that, for all terms expected to diverge, it is only possible to derive ∞ as result. In this way, it is also possible to provide rules for propagation of divergence, solving the second problem.

In the present paper, we face the problem of expressing more sophisticated notions of infinite behaviour. That is, we want to model not only that evaluation of an expression e could not terminate, but also that this diverging computation could produce some possibly infinite result. For instance, if computations can have output effects because of expression `out e`, then the result of a diverging computation could be a possibly infinite stream of output values. More in general, the behaviour of a non terminating program is a possibly infinite trace of observable events. As discussed above for pure divergence, this is modeled at the meta-level in small-step definitions: the inference system does *not* define the judgment $e \Rightarrow r$ where r is an infinite result, but only single labelled steps, and then we define divergent results as infinite sequences of labelled steps. On the other hand, again analogously to the pure divergence case, big-step rules interpreted simply coinductively allow derivation of spurious results. Hence, we investigate whether the approach of [6] could be effectively used also in this more general case to filter out such judgments.

Our conclusion, illustrated in the body of the paper, is that the approach based on filtered coevaluation works very well indeed for modeling trace semantics, or, more in general, infinite behaviour of divergent programs. Moreover, an interesting result is that, to filter out spurious judgments, we need to use not only coaxioms, but also *corules*. Corules were not considered in the original definition of generalized inference systems [5], even though formal definitions trivially extend to include them, simply for the lack of significant examples. Modeling infinite behaviour offers exactly such a significant example: notably, corules are needed to ensure that a diverging computation yields a possibly infinite result. For instance, if expression `out e` is reached infinitely many times during the non terminating evaluation of another expression, then the latter actually produces the effects of `out e`, providing that e converges. Similar corules are needed for other constructs having other kinds of observable behavior. Motivated by this important application, we present in this paper a slight variation of generalized inference systems [5] which also includes corules; we will use “generalized inference system” and “inference system with corules” as synonyms.

Summarizing the above discussion, the novel contribution of this paper w.r.t. previous work is not the general framework, which is the same of [5, 6] (except for the immediate generalization from coaxioms to corules), including, e.g., the fixpoint theory, not reported here. The novel, non trivial problem faced here is how to obtain, starting from an inductive judgment defining the result of finite computations, an extended one defining the (possibly infinite) result of infinite computations as well. We show that the problem can be solved by adding suitable corules and considering the resulting generalized inference system. This has been done in [6] only in the very special case where the only observable result is divergence. The general case is more challenging and requires more complex corules.

The paper is structured as follows. Sect. 2 introduces inference systems with corules, briefly recalling/extending the notions and results from [5]. In Sect. 3 we illustrate the approach on a lambda calculus enriched with output effects. That is, we show that, by using corules, we can directly define a unique judgment $e \Rightarrow r$ where r is either a finite result (final value and finite output stream) or an infinite result (∞ and possibly infinite output stream). In Sect. 4 we show that, by only using standard techniques, namely, labeled transition systems, coinduction, and observational equivalence, we can provide an alternative semantics which, however, requires much more work. In Sect. 5 we formally prove the equivalence of such two semantics. In Sect. 6 we consider a more involved example, that is, a simple imperative Java-like language with I/O primitives. Finally, the most relevant related work is surveyed in Sect. 7, and Sect. 8 concludes and outlines directions for further investigation. The Appendix contains an algebraic presentation of the observational equivalence, some of the proofs and additional examples.

2 Inference systems with corules

In this section, we provide a short introduction to inference systems with corules, needed to make the paper self-contained. The material is largely taken from [5], apart that we consider, besides coaxioms, *corules* with an analogous meaning. Here we focus on the proof-theoretic view, which is essential for our aims.

First of all we recall standard notions about inference systems [1, 13]. Assume a set \mathcal{U} called the *universe*, whose elements are called *judgments*. An *inference system* \mathcal{I} consists of a set of (*inference*) *rules*, which are pairs $\frac{Pr}{c}$, with $Pr \subseteq \mathcal{U}$ the set of *premises*, $c \in \mathcal{U}$ the *consequence* (a.k.a. *conclusion*). A rule with an empty set of premises is also called an *axiom*. A *proof tree* is a tree whose nodes are (labeled with) judgments in \mathcal{U} , and there is a node c with set of children Pr only if there exists a rule $\frac{Pr}{c}$.

The *inductive interpretation* of \mathcal{I} , denoted $\llbracket \mathcal{I} \rrbracket^{\text{ind}}$, is the set of judgments which are the root of a finite³ proof tree, whereas the *coinductive interpretation* of \mathcal{I} , denoted $\llbracket \mathcal{I} \rrbracket^{\text{coind}}$, is the set of judgments which are the root of an arbitrary (finite or infinite) proof tree.

Both interpretations can also be characterized set-theoretically as follows. We define the (*one step*) *inference operator* $F_{\mathcal{I}}: \wp(\mathcal{U}) \rightarrow \wp(\mathcal{U})$ by $F_{\mathcal{I}}(S) = \{c \mid Pr \subseteq S, \frac{Pr}{c} \in \mathcal{I}\}$. A set S is *closed* if $F_{\mathcal{I}}(S) \subseteq S$, and *consistent* if $S \subseteq F_{\mathcal{I}}(S)$. That is, no new judgments can be inferred from a closed set, and all judgments in a consistent set can be inferred from the set itself. Then, it can be proved that $\llbracket \mathcal{I} \rrbracket^{\text{ind}}$ is the smallest closed set, that is, the intersection of all closed sets, and $\llbracket \mathcal{I} \rrbracket^{\text{coind}}$ is the largest consistent set, that is, the union of all consistent sets.

³ Under the common assumption that sets of premises are finite, otherwise we should say a well-founded tree, that is, a tree with no infinite paths.

21:4 Modeling Infinite Behaviour by Corules

We describe now the notion of inference system with corules.

► **Definition 1** (Inference system with corules). An *inference system with corules*, or *generalized inference system*, is a pair $(\mathcal{I}, \mathcal{I}^{\text{co}})$ where \mathcal{I} and \mathcal{I}^{co} are inference systems, whose elements are called *rules* and *corules*, respectively.

Analogously to rules, the meaning of corules is to derive a consequence from the premises. However, they can only be used in a special way, described in the following.

Given two inference systems \mathcal{I} and \mathcal{I}^{co} , $\mathcal{I} \cup \mathcal{I}^{\text{co}}$ is the (standard) inference system whose rules are the union of those in \mathcal{I} and \mathcal{I}^{co} . Moreover, given a subset S of the universe, $\mathcal{I}|_S$ denotes the inference system obtained from \mathcal{I} by keeping only rules with consequence in S . Then, the interpretation of an inference system with corules $(\mathcal{I}, \mathcal{I}^{\text{co}})$ is defined as follows.

1. First, we consider the inference system $\mathcal{I} \cup \mathcal{I}^{\text{co}}$ where corules can be used as rules as well, and we take its inductive interpretation $\llbracket \mathcal{I} \cup \mathcal{I}^{\text{co}} \rrbracket^{\text{ind}}$.
2. Then, we take the coinductive interpretation of the inference system obtained from \mathcal{I} by keeping only rules with consequence in $\llbracket \mathcal{I} \cup \mathcal{I}^{\text{co}} \rrbracket^{\text{ind}}$.

Altogether, we get the following definition.

► **Definition 2** (Interpretation of a generalized inference system). Let $(\mathcal{I}, \mathcal{I}^{\text{co}})$ be a generalized inference system. Then, its *interpretation* $\llbracket \mathcal{I}, \mathcal{I}^{\text{co}} \rrbracket$ is defined by

$$\llbracket \mathcal{I}, \mathcal{I}^{\text{co}} \rrbracket = \llbracket \mathcal{I} |_{\llbracket \mathcal{I} \cup \mathcal{I}^{\text{co}} \rrbracket^{\text{ind}}} \rrbracket^{\text{coind}}$$

In proof-theoretic terms, $\llbracket \mathcal{I}, \mathcal{I}^{\text{co}} \rrbracket$ is the set of judgments which have an arbitrary (finite or infinite) proof tree in \mathcal{I} , whose nodes all have a finite proof tree in $\mathcal{I} \cup \mathcal{I}^{\text{co}}$. Note that a finite proof tree in \mathcal{I} is a finite proof tree in $\mathcal{I} \cup \mathcal{I}^{\text{co}}$ as well, hence the condition is only significant for nodes which are roots of an infinite path in the proof tree.

We report now some examples from [5] which illustrate the expressive power of generalized inference systems. Both examples only use corules with no premises (*coaxioms*). Examples which need corules with premises will be shown in the following section.

As usual, sets of rules can be expressed by a *metarule* with side conditions, and analogously sets of corules can be expressed by a *metacorule* with side conditions. In the examples,

(meta)corules will be written $\frac{Pr}{c}$, that is, with thicker lines, to be distinguished from (meta)rules.

The first example computes the judgment $n \xrightarrow{*} \mathcal{N}$ meaning that \mathcal{N} is the set of nodes reachable from a node n of a given graph. Let us represent a graph by its set of nodes V and a function adj which returns all the adjacents of a node. The judgment is defined by the generalized inference system $(\mathcal{I}, \mathcal{I}^{\text{co}})$ where \mathcal{I} and \mathcal{I}^{co} are all the instances of (ADJ) and (CO-EMPTY) below, respectively.

$$(\text{ADJ}) \frac{n_1 \xrightarrow{*} \mathcal{N}_1 \ \dots \ n_k \xrightarrow{*} \mathcal{N}_k}{n \xrightarrow{*} \{n\} \cup \mathcal{N}_1 \cup \dots \cup \mathcal{N}_k} \quad adj(n) = \{n_1, \dots, n_k\} \quad (\text{CO-EMPTY}) \frac{}{n \xrightarrow{*} \emptyset} \quad n \in V$$

Consider, for instance, a graph with nodes a, b, c , with an arc from a into b and conversely, and c isolated. To show the aim of corules, let us first consider what happens if we only consider metarule (ADJ) , disregarding the corules. We have in this way a (standard) inference system, which, as described above, can be interpreted either inductively or coinductively. However, neither interpretation provides the desired meaning. Indeed, if we interpret (ADJ) inductively, then we get only the judgment $c \xrightarrow{*} \{c\}$. On the other hand, if we interpret the rules coinductively, then we get the correct judgments $a \xrightarrow{*} \{a, b\}$ and $b \xrightarrow{*} \{a, b\}$, but we also

get the wrong judgments $a \xrightarrow{*}\{a, b, c\}$ and $b \xrightarrow{*}\{a, b, c\}$. For instance, the judgment $a \xrightarrow{*}\{a, b, c\}$ has the infinite proof tree shown below.

$$\frac{\frac{\frac{\vdots}{(ADJ)}{a \xrightarrow{*}\{a, b, c\}}}{(ADJ)}{b \xrightarrow{*}\{a, b, c\}}}{(ADJ)}{a \xrightarrow{*}\{a, b, c\}}$$

If we take into account corules, instead, then the interpretation of the resulting generalized inference system provides the desired meaning. For instance, in the example, the judgment $a \xrightarrow{*}\{a, b\}$ has an infinite proof tree in \mathcal{I} where each node has a finite proof tree in $\mathcal{I} \cup \mathcal{I}^{\text{co}}$, as shown below:

$$\frac{\frac{\frac{\vdots}{(ADJ)}{a \xrightarrow{*}\{a, b\}}}{(ADJ)}{b \xrightarrow{*}\{a, b\}}}{(ADJ)}{a \xrightarrow{*}\{a, b\}} \quad \frac{\frac{\text{(CO-EMPTY)}}{(ADJ)}{a \xrightarrow{*}\emptyset}}{(ADJ)}{b \xrightarrow{*}\{b\}}}{(ADJ)}{a \xrightarrow{*}\{a, b\}} \quad \frac{\frac{\text{(CO-EMPTY)}}{(ADJ)}{b \xrightarrow{*}\emptyset}}{(ADJ)}{a \xrightarrow{*}\{a\}}}{(ADJ)}{b \xrightarrow{*}\{a, b\}}$$

whereas this is not the case for the judgment $a \xrightarrow{*}\{a, b, c\}$. In other words, corules filter out undesired infinite proof trees.

Note that the inductive and coinductive interpretation of \mathcal{I} are special cases, notably:

- the inductive interpretation of \mathcal{I} is the interpretation of (\mathcal{I}, \emptyset)
- the coinductive interpretation of \mathcal{I} is the interpretation of $(\mathcal{I}, \{\frac{\emptyset}{c} \mid c \in \mathcal{U}\})$.

In [5] it is shown that this corresponds to taking a fixed point of $F_{\mathcal{I}}$ which is, in general, neither the least, nor the greatest.

We show now how the recursive definition of a function can be expressed as an inference system with corules. Let \mathbb{Z} denote the set of integers, \mathbb{L} the set of (finite and infinite) lists of integers, Λ the empty list and $x:l$ the list with head x and tail l .

The function which returns the greatest element contained in a (non empty) list is expressed by judgments of shape $\text{max}(l, x)$, with $l \in \mathbb{L}$ and $x \in \mathbb{Z}$.

$$\frac{\text{max}(x:\Lambda, x)}{\text{max}(x:\Lambda, x)} \quad \frac{\text{max}(l, y)}{\text{max}(x:l, z)} z = \text{max}(x, y) \quad \frac{\text{max}(x:l, x)}{\text{max}(x:l, x)}$$

Without coaxioms the coinductive interpretation fails to be a function (for instance, for l the infinite list of 1s, any judgment $\text{max}(l, x)$ with $x \geq 1$ can be derived), and the coaxioms “filter out” the wrong results. We refer to related work [5, 6] for other examples.

Several proof techniques have been proposed for generalized inference systems with coaxioms [5]. For the aim of this paper, we only need the *bounded coinduction principle* reported below, which is a generalization of the standard coinduction principle.

Let $(\mathcal{I}, \mathcal{I}^{\text{co}})$ be a generalized inference system, and \mathcal{S} (for “specification”) an intended set of judgments, called *valid* in the following. *Completeness*, that is, the property that each valid judgment can be derived ($\mathcal{S} \subseteq \llbracket \mathcal{I}, \mathcal{I}^{\text{co}} \rrbracket$), can be proved as follows:

► **Theorem 3** (Bounded coinduction principle). *If the following two conditions hold:*

1. $\mathcal{S} \subseteq \llbracket \mathcal{I} \cup \mathcal{I}^{\text{co}} \rrbracket^{\text{ind}}$, that is, each valid judgment has a finite proof tree in $\mathcal{I} \cup \mathcal{I}^{\text{co}}$;
2. $\mathcal{S} \subseteq F_{\mathcal{I}}(\mathcal{S})$, that is, each valid judgment is the consequence of an inference rule in \mathcal{I} where all premises are in \mathcal{S}

then $\mathcal{S} \subseteq \llbracket \mathcal{I}, \mathcal{I}^{\text{co}} \rrbracket$.

$$\begin{array}{ll}
 e ::= v \mid x \mid e_1 \ e_2 \mid \text{out } e & \text{expression} \\
 v ::= i \mid \lambda x. e & \text{value} \\
 \ell ::= v \mid \tau & \text{label} \\
 \mathcal{E}[\] ::= \square \mid \mathcal{E}[\] \ e \mid v \ \mathcal{E}[\] \mid \text{out } \mathcal{E}[\] & \text{evaluation context}
 \end{array}$$

$$\begin{array}{c}
 (\beta) \frac{}{(\lambda x. e) \ v \xrightarrow{\tau} e[x \leftarrow v]} \quad
 (\text{out}) \frac{}{\text{out } v \xrightarrow{v} v} \quad
 (\text{ctx}) \frac{e \xrightarrow{\ell} e'}{\mathcal{E}[e] \xrightarrow{\ell} \mathcal{E}[e']} \quad \mathcal{E}[\] \neq \square
 \end{array}$$

■ **Figure 1** λ -calculus with output effects: syntax and labeled transition system.

The standard coinduction principle can be obtained when $\mathcal{I}^{\text{co}} = \{\frac{\emptyset}{c} \mid c \in \mathcal{U}\}$; for this particular case the first condition trivially holds.

3 Infinite behaviour by corules

As mentioned in the Introduction, in this paper we are interested in modeling infinite behaviour. That is, in addition to explicitly model divergent computations, we would like to also model their possibly infinite result. This infinite result can be thought of as what an external observer can see during the infinite computation. In this section we show how corules can be used to define such a semantics.

We illustrate the technique by an extension of the call-by-value lambda calculus with output effects. The syntax is given in Fig. 1, together with a labeled transition system meant to provide the reader with a simple formal account of the intended meaning.

We assume infinite sets of *variables* x and *integer constants* i . Values are defined in the standard way, as either integer literals or lambda abstractions. Beyond standard constructs, we added expressions of shape $\text{out } e$, which output the result of the evaluation of e .

As it is common practice, the reduction relation is decorated by a *label* ℓ representing the observable effect of a single step. A label v represents an output effect, and can be generated by rule (out), while a label τ represents the absence of observable output and can be generated by the β -rule. The rule (ctx) is the usual contextual closure and defines the standard (call-by-value and left-to-right) evaluation strategy.

As discussed in the Introduction, the labelled transition system in Fig. 1 models a single evaluation step, and infinite behaviour is only obtained at the meta-level by considering infinite sequences of labelled steps.

We show now a generalized inference system defining a judgment $e \Rightarrow r$ which directly models both finite and infinite behaviour of expressions.

The top section of Fig. 2 defines *results* r of (finite and infinite) computations. If the evaluation of an expression terminates, then the result of the computation is of shape (v, o) where v is the final value, and o is the (necessarily finite) output stream produced during the evaluation. If the evaluation of an expression diverges, hence there is no final value, then the result is of shape (∞, o_∞) where o_∞ is the (possibly infinite) output stream produced during the evaluation. Output streams are sequences of values delimited by square brackets for readability. Streams grow left-to-right; hence, the rightmost element in a finite stream corresponds to the most recent output value. Concatenation of two streams $o \cdot o_\infty$ is defined in the usual way, under the assumption that the left-hand side operand must be finite.

The second section of Fig. 2 contains the generalized inference system defining the judgment $e \Rightarrow r$, meaning that r is the (finite or infinite) result of the evaluation of e .

$$\begin{array}{ll}
r ::= (v, o) \mid (\infty, o_\infty) & \text{result} \\
o ::= [v_1 \dots v_n] & \text{finite output} \\
o_\infty ::= o \mid [v_1 \dots v_n \dots] & \text{finite/infinite output} \\
\mathcal{D}[\] ::= \square e \mid \square \square \mid \text{out } \square & \text{(divergence) propagation context}
\end{array}$$

$$\begin{array}{c}
(\text{VAL}) \frac{}{v \Rightarrow (v, [])} \quad (\text{APP}) \frac{e_1 \Rightarrow (\lambda x. e, o_1) \quad e_2 \Rightarrow (v, o_2) \quad e[x \leftarrow v] \Rightarrow r}{e_1 e_2 \Rightarrow o_1 \cdot o_2 \cdot r} \\
(\text{OUT}) \frac{e \Rightarrow (v, o)}{\text{out } e \Rightarrow (v, o \cdot [v])} \quad (\text{DIV}) \frac{\forall i = 1..n. e_i \Rightarrow (v_i, o_i) \quad e \Rightarrow (\infty, o_\infty)}{\mathcal{D}[\bar{e}^n, e] \Rightarrow (\infty, o_1 \cdot \dots \cdot o_n \cdot o_\infty)} \\
(\text{CO-EMPTY}) \frac{}{e \Rightarrow (\infty, [])} \quad (\text{CO-OUT}) \frac{e \Rightarrow (v, o)}{\text{out } e \Rightarrow (\infty, o \cdot [v] \cdot o_\infty)}
\end{array}$$

■ **Figure 2** λ -calculus with output effects: inference system with corules.

Propagation contexts can have one or two holes (all at fixed depth 1) and allow a more concise treatment of divergence propagation with a single rule, see comments below for (DIV).

We recall that in rules thicker lines distinguish corules from rules.

Rule (VAL) is straightforward: the evaluation of a value always converges to itself and produces no output.

Rule (APP) is an extension of the usual big-step rule for application. First, the two subexpressions are evaluated; if they both converge, then the body of the function is evaluated after the formal parameter has been substituted with its argument. As usual, $e[x \leftarrow v]$ denotes capture-avoiding substitution modulo α -renaming. The evaluation of the body returns a general result, meaning that the evaluation may either converge or diverge. The final result is obtained by suitably concatenating the output streams generated by the evaluations of e_1 and e_2 with that generated by the evaluation of the body. If $r = (v_\infty, o_\infty)$, then $o \cdot r$ denotes $(v_\infty, o \cdot o_\infty)$, where v_∞ is either a value v or ∞ .

In rule (OUT), if the evaluation of e converges to a value v , then the whole expression converges to the same value; moreover, the value v is added to the output stream o generated by the evaluation of e .

The remaining rule (DIV) deals with propagation of non termination and composition of the corresponding output streams. Evaluation of the subexpressions in the holes of the context proceeds from left to right and the subexpression corresponding to the rightmost hole is the first one which diverges. In this simple language (divergence) propagation contexts have no more than two holes, therefore the number n of subexpressions that converge can be either 0 or 1. More in detail, the context $\square e$ corresponds to the case where application diverges because the left-hand side expression does not terminate, whereas the context $\square \square$ to the case where the left-hand side expression terminates, whereas the right-hand side does not. Finally, $\text{out } \square$ propagates non termination of the unique subexpression of $\text{out } e$.

As explained in the previous section, corules are used to filter out spurious results of divergent computations.

Corule (CO-EMPTY) deals with divergent computations which produce a finite output stream, hence do not produce any output (we will also say that the infinite computation is “non-productive”) from a certain point. In this case, as shown in the first two examples below, (CO-EMPTY) prevents derivation of judgments with arbitrary output streams, similarly to the examples shown in the previous section. In all such cases the only correct option is non-termination with the empty output stream $[]$.

$$\begin{array}{c}
 \begin{array}{c}
 \text{---} \\
 \vdots \\
 \text{---}
 \end{array} \\
 \frac{\text{(VAL)} \frac{\lambda x.x \ x \Rightarrow (\lambda x.x \ x, [])}{\lambda x.x \ x \Rightarrow (\lambda x.x \ x, [])} \quad \text{(VAL)} \frac{\lambda x.x \ x \Rightarrow (\lambda x.x \ x, [])}{\lambda x.x \ x \Rightarrow (\lambda x.x \ x, [])} \quad \text{(APP)} \frac{\Omega_1 \equiv (x \ x)[x \leftarrow \lambda x.x \ x] \Rightarrow r}{\Omega_1 \equiv (x \ x)[x \leftarrow \lambda x.x \ x] \Rightarrow r}}{\nabla_1 = \text{(APP)} \frac{\Omega_1 \Rightarrow [] \cdot [] \cdot r \equiv r}{\Omega_1 \Rightarrow [] \cdot [] \cdot r \equiv r}} \\
 \frac{\nabla_1}{\text{(OUT)} \frac{\Omega_1 \Rightarrow (v, o)}{\text{out } \Omega_1 \Rightarrow (v, o \cdot [v])}} \quad \frac{\nabla_1}{\text{(DIV)} \frac{\Omega_1 \Rightarrow (\infty, o_\infty)}{\text{out } \Omega_1 \Rightarrow (\infty, o_\infty)}}
 \end{array}$$

■ **Figure 3** Infinite proof trees for Example 1.

Corule (co-out) deals with non terminating terms which produce an infinite number of values; this happens if expressions of shape $\text{out } e$ are evaluated an infinite number of times. The premise requires the subexpression e to converge to a value v , ensuring that the output expression is actually evaluated, adding v to the output stream. In this way we guarantee productivity, that may not hold if e diverges (see the difference between examples 2 and 3 below).

Example 1. As a first example, we consider the term $\text{out } \Omega_1$, where $\Omega_1 = (\lambda x.x \ x) (\lambda x.x \ x)$; the only derivable judgment is $\text{out } \Omega_1 \Rightarrow (\infty, [])$, corresponding to the expected semantics: the evaluation of $\text{out } \Omega_1$ diverges and generates an empty output stream.

Indeed, a judgment $\text{out } \Omega_1 \Rightarrow r$ is derivable if:

- it has a possibly infinite proof tree with no corules
- such proof tree is valid according to the corules, that is, each node has a finite proof tree with corules.

The first condition is illustrated in Fig. 3. The top part of the figure shows the infinite proof tree for the judgment $\Omega_1 \Rightarrow r$, for all possible r , where the vertical dots indicate that the proof continues with the same repeated pattern. Here and in the following examples, we add to the proof tree some comments (with a grey background) showing an equivalent expression, as a help for the reader. No other finite or infinite proof trees can be built for such judgment. In such a simple case the proof tree is regular; however, there exist examples of divergent computations with non regular proof trees.

For the evaluation of $\text{out } \Omega_1$ we can apply two different rules, depending on the shape of r . If r is a converged result (v, o) , then the only applicable rule is (out) , and we derive the judgment $\text{out } \Omega_1 \Rightarrow (v, o \cdot [v])$; otherwise, r is a diverged result (∞, o_∞) , and the judgment $\text{out } \Omega_1 \Rightarrow (\infty, o_\infty)$ can be derived by rule (div) .

Among all judgments derivable with an infinite tree built with only the rules as shown above, the only one that is valid according to the corules is $\text{out } \Omega_1 \Rightarrow (\infty, [])$; in this case it suffices to exhibit a finite proof tree for $\Omega_1 \Rightarrow (\infty, [])$ which can be built by applying also the corules. Such a tree is trivial, thanks to $\text{coaxiom } (\text{co-empty})$:

$$\text{(CO-EMPTY)} \frac{}{\Omega_1 \Rightarrow (\infty, [])}$$

By rule (div) , and from the trivial finite tree above, it is possible to derive a finite tree also for the judgment $\text{out } \Omega_1 \Rightarrow (\infty, [])$.

It is easy to see that instead, for $r \neq (\infty, [])$, it is not possible to derive a finite tree, built by also the corules, for the judgment $\Omega_1 \Rightarrow r$.

$$\nabla_2 = \frac{\frac{\frac{\text{(VAL)} \overline{\omega_2 \Rightarrow (\omega_2, [])}}{\omega_2 \Rightarrow (\omega_2, [])} \quad \frac{\text{(VAL)} \overline{\omega_2 \Rightarrow (\omega_2, [])}}{\omega_2 \Rightarrow (\omega_2, [])} \quad \frac{\text{(DIV)} \overline{\frac{\text{(APP)} \overline{\frac{\vdots}{\Omega_2 \Rightarrow (\infty, o_\infty)}}{\text{out } \Omega_2 \equiv (\text{out } (x \ x))[x \leftarrow \omega_2] \Rightarrow (\infty, o_\infty)}}}}{\text{out } \Omega_2 \equiv (\text{out } (x \ x))[x \leftarrow \omega_2] \Rightarrow (\infty, o_\infty)}}}{\Omega_2 \Rightarrow [] \cdot [] \cdot (\infty, o_\infty)} \equiv (\infty, o_\infty)}{\text{(APP)} \overline{\Omega_2 \Rightarrow (\infty, o_\infty)}}$$

■ **Figure 4** Infinite proof trees for Example 2.

$$\nabla_3 = \frac{\frac{\frac{\text{(VAL)} \overline{\omega_3 \Rightarrow (\omega_3, [])}}{\omega_3 \Rightarrow (\omega_3, [])} \quad \frac{\text{(OUT)} \overline{\frac{\text{(VAL)} \overline{\omega_3 \Rightarrow (\omega_3, [])}}{\omega_3 \Rightarrow (\omega_3, [])}}}{\text{out } \omega_3 \Rightarrow (\omega_3, [\omega_3])} \quad \frac{\text{(APP)} \overline{\frac{\text{(VAL)} \overline{\frac{\vdots}{\omega_3 \text{ (out } \omega_3) \equiv (x \text{ (out } x))[x \leftarrow \omega_3] \Rightarrow (\infty, o_\infty)}}}}{\omega_3 \text{ (out } \omega_3) \equiv (x \text{ (out } x))[x \leftarrow \omega_3] \Rightarrow (\infty, o_\infty)}}}}{\omega_3 \text{ (out } \omega_3) \equiv (x \text{ (out } x))[x \leftarrow \omega_3] \Rightarrow [] \cdot [\omega_3] \cdot (\infty, o_\infty)}}}{\frac{\frac{\text{(VAL)} \overline{\omega_3 \Rightarrow (\omega_3, [])}}{\omega_3 \Rightarrow (\omega_3, [])} \quad \frac{\text{(VAL)} \overline{\omega_3 \Rightarrow (\omega_3, [])}}{\omega_3 \Rightarrow (\omega_3, [])} \quad \nabla_3}{\Omega_3 \Rightarrow [] \cdot [] \cdot (\infty, o_\infty)} \equiv (\infty, o_\infty)}}{\text{(APP)} \overline{\Omega_3 \Rightarrow (\infty, o_\infty)}}$$

■ **Figure 5** Infinite proof tree for Example 3.

Example 2. As a second example, we consider the term $\Omega_2 = \omega_2 \ \omega_2 = (\lambda x. \text{out } (x \ x)) (\lambda x. \text{out } (x \ x))$ and show that, as in the previous case, the only derivable judgment is $\Omega_2 \Rightarrow (\infty, [])$, as expected: the evaluation of Ω_2 does not terminate and does not output any value.

By applying only the rules, the infinite proof tree shown in Fig. 4 can be built for the judgment $\Omega_2 \Rightarrow (\infty, o_\infty)$, for any possible o_∞ . No judgments of shape $\Omega_2 \Rightarrow (v, o)$ can be derived, because this could be achieved only with an infinite proof tree containing infinite applications of rule (OUT) for the judgment $\text{out } \Omega_2 \Rightarrow (v, o)$. This is not possible because such a rule is applicable only for finite output streams, whereas the infinite applications of (OUT) would force o to be infinite. Among all judgments derivable with an infinite tree built with the rules as shown above, the only one that is valid according to the corules is $\text{out } \Omega_2 \Rightarrow (\infty, [])$; in this case it is sufficient to exhibit a finite proof tree which uses also the corules for the judgment $\text{out } \Omega_2 \Rightarrow (\infty, [])$. Again, this can be simply obtained by corule (CO-EMPTY):

$$\frac{\text{(CO-EMPTY)}}{\text{out } \Omega_2 \Rightarrow (\infty, [])}$$

By rule (APP), and from the simple finite tree above, it is possible to derive a finite tree for the judgment $\Omega_2 \Rightarrow (\infty, [])$ as well.

No finite proof tree can be derived for $\text{out } \Omega_2 \Rightarrow (\infty, o_\infty)$ for any $o_\infty \neq []$, because corule (CO-OUT) can be used only if $\Omega_2 \Rightarrow (v, o)$ can be derived (by using also the corules) with a finite tree, but this is not possible, because the only possible derivable trees are infinite, as shown above.

Example 3. As a final example, we show that the only judgment derivable for $\Omega_3 = \omega_3 \ \omega_3 = (\lambda x. (x \ (\text{out } x))) (\lambda x. (x \ (\text{out } x)))$ is $\Omega_3 \Rightarrow (\infty, [\omega_3 \dots \omega_3 \dots])$, corresponding to the expected semantics: the evaluation of Ω_3 diverges and generates the output stream consisting of infinite occurrences of the value ω_3 .

In the system without corules it is possible to build only one proof tree, shown in Fig. 5, which is infinite and forces the equation $o_\infty = [\omega_3] \cdot o_\infty$, which admits a unique solution; hence, the judgment $\Omega_3 \Rightarrow (\infty, o_\infty)$ is derivable only for $o_\infty = [\omega_3 \dots \omega_3 \dots]$. Then, we have to show that by considering also the corules we can derive finite proof trees for all judgments involved in the proof tree for $\Omega_3 \Rightarrow (\infty, o_\infty)$; to this aim, it is sufficient to exhibit a finite derivation just for the judgment $(x \text{ (out } x)) [x \leftarrow \omega_3] \Rightarrow (\infty, o_\infty)$ at the root of proof tree ∇_3 .

$$\frac{\frac{\text{(VAL)}}{\omega_3 \Rightarrow (\omega_3, [\])} \quad \frac{\text{(CO-OUT)}}{\text{out } (\omega_3) \Rightarrow (\infty, o_\infty)}}{\text{(DIV)} \quad \frac{\omega_3 \Rightarrow (\omega_3, [\]) \quad \text{out } (\omega_3) \Rightarrow (\infty, o_\infty)}{(x \text{ (out } x)) [x \leftarrow \omega_3] \Rightarrow [\] \cdot (\infty, o_\infty)} \equiv (\infty, o_\infty)}$$

By rule (APP) and from the finite tree above, it is possible to derive a finite tree as well for $\Omega_3 \Rightarrow (\infty, o_\infty)$.

We conclude this section by proving a *conservativity* property [6] which we always expect to hold for an operational semantics modeling also divergence, given through corules. Namely, we require that the introduction of divergence does not affect standard convergent computations, as formally stated in the following theorem. This property is important since it implies that, for convergent judgments, we can reason by standard inductive techniques.

► **Theorem 4** (Conservativity of $e \Rightarrow r$). *If $e \Rightarrow r$ holds, then $r = (v, o)$ if and only if the judgment has a finite proof tree.*

Proof. Let us denote by $(\mathcal{I}, \mathcal{I}^{\text{co}})$ the generalized inference system defining the judgment $e \Rightarrow r$ in Fig. 2, and by $\mathcal{I} \cup \mathcal{I}^{\text{co}}$ the (standard) inference system that is the union of \mathcal{I} and \mathcal{I}^{co} . If $e \Rightarrow r$ is derivable in $(\mathcal{I}, \mathcal{I}^{\text{co}})$, then by definition all judgments in the proof tree have a finite derivation in $\mathcal{I} \cup \mathcal{I}^{\text{co}}$, including the judgment itself. Noting that a rule's conclusion is a divergent judgment iff at least one of the premises is also divergent (see rules (DIV) and (APP)), it can be shown by induction on the depth of the tree that $r = (\infty, o_\infty)$ iff we use either (CO-EMPTY) or (CO-OUT) in the tree. It follows that $r = (v, o)$ iff we do not use (CO-EMPTY) and (CO-OUT) in the finite proof tree in $\mathcal{I} \cup \mathcal{I}^{\text{co}}$, hence it is a finite proof tree in \mathcal{I} as needed. ◀

4 Infinite behaviour by standard techniques

In this section we show that, by only using standard techniques, namely, labeled transition systems (LTSs from now on), coinduction, and observational equivalence, we can provide an alternative semantics for lambda calculus with output effects, which, however, is much more involved than the direct definition provided in the previous section.

The starting point is the LTS introduced in Fig. 1. This (inductive) inference system provides a very simple and intuitive model, which, however, is not abstract enough for reasoning about the observable behaviour of programs. Namely, two expressions having different sequences of labeled steps could be equivalent in terms of their observable behaviour.

To obtain, starting from the LTS, the same level of abstraction of the semantics defined in the previous section, some additional work is needed.

First of all, as discussed in the Introduction, the overall result of a computation is only modeled at the meta-level in the labeled transition system: that is, this inference system only defines single steps, and then we say that there is “a possibly infinite sequence of labeled steps”. To express this statement, instead, in a formal way, we can define, of top of the LTS, another judgment $e \rightsquigarrow \tilde{r}$ which associates with each expression e its result, consisting of the final value, if any, or ∞ if diverging, and the possibly infinite stream of labels produced during the computation. This judgment can be defined by standard coinduction, as detailed below.

$$\begin{aligned}
\tilde{r} &::= (v, \tilde{\sigma}) \mid (\infty, \tilde{\sigma}_\infty) && \text{(rough) result} \\
\tilde{\sigma}, \tilde{s} &::= [\ell_1 \dots \ell_n] && \text{finite stream} \\
\tilde{\sigma}_\infty, \tilde{s}_\infty &::= \tilde{\sigma} \mid [\ell_1 \dots \ell_n \dots] && \text{finite/infinite stream}
\end{aligned}$$

$$\frac{}{v \rightsquigarrow (v, [])} \quad \frac{e' \rightsquigarrow \tilde{r}}{e \rightsquigarrow [\ell] \cdot \tilde{r}} \quad e \xrightarrow{\ell} e' \quad \frac{}{e \rightsquigarrow \tilde{r}}$$

■ **Figure 6** λ -calculus with output effects: coinductive inference system.

However, not even this judgment is abstract enough, since, as mentioned above, different sequences of labels could be equivalent in terms of observable behaviour, that is, roughly, τ steps should be not relevant. In other words, results \tilde{r} obtained in this judgment need a further abstraction step, consisting in the definition of an appropriate observational equivalence: in the following, we will call them *rough* results, and we will call *observable* results those obtained in the judgment defined in the previous section. We describe now in details the two steps.

Computing rough results. The definition of the judgment $e \rightsquigarrow \tilde{r}$ is given in Fig. 6.

The top section of the figure defines (rough) results \tilde{r} of (finite and infinite) computations. The definition is analogous to that of (observable) results r in Fig. 2. However, here the second component of a result is, rather than an output stream (a stream of values), a stream of labels (called simply a stream from now on), corresponding to the fact that a computation step can be silent (no output). Concatenation of two streams $\tilde{\sigma} \cdot \tilde{\sigma}_\infty$ is also defined analogously to that of output streams, hence requires that $\tilde{\sigma}$ is finite.

The second section of the figure contains the coinductive inference system defining the judgment $e \rightsquigarrow \tilde{r}$, meaning that \tilde{r} is the (rough) finite or infinite result of the evaluation of e .

Again analogously to output streams and (observable) results, we can extend concatenation of streams to (rough) results by setting $\tilde{\sigma} \cdot (v_\infty, \tilde{\sigma}_\infty) = (v_\infty, \tilde{\sigma} \cdot \tilde{\sigma}_\infty)$, where v_∞ is either a value v or ∞ .

A very important point to be noted and discussed is that the definition of the judgment $e \rightsquigarrow \tilde{r}$ is *purely coinductive*. Indeed, in Fig. 6, we used the notation of generalized inference systems for uniformity, but, as mentioned on page 5, a generalized inference system with a unique (meta)coaxiom where the consequence is any judgment exactly corresponds to the inference system consisting of only the rules, interpreted coinductively.

In this case it is possible to give a purely coinductive definition, without incurring in the problem of spurious judgment discussed in the Introduction for coevaluation, since the definition is *productive*. That is, each time we apply the inference rule, we add a label ℓ to the previously computed result, thus also infinite derivations admit a unique result. In order to guarantee productivity, the presence of labels τ is essential, even though they are not interesting semantically, since they represent non-observable steps. This motivates the fact that, as mentioned above, by using standard techniques we need two additional steps on top of the labeled transition system: indeed, to be able to use pure coinduction we need rough results, and then we need to identify rough results which are observationally equivalent. By using corules, instead, as shown in the previous section, we are able to directly define the (observable) behaviour by a unique judgment.

$$\begin{array}{c}
\text{(TAU)} \frac{}{\tau^\alpha \approx \tau^\beta} \quad \alpha, \beta \in \mathbb{N} \cup \{\omega\} \quad \text{(VAL)} \frac{\tilde{o}_\infty \approx \tilde{o}'_\infty}{\tau^n \cdot [v] \cdot \tilde{o}_\infty \approx \tau^m \cdot [v] \cdot \tilde{o}'_\infty} \quad n, m \in \mathbb{N} \quad \text{(CO)} \frac{}{\tilde{o}_\infty \approx \tilde{o}'_\infty}
\end{array}$$

■ **Figure 7** Observational equivalence: coinductive inference system.

Also for the judgment $e \rightsquigarrow \tilde{r}$ we can state a conservativity result analogous to Theorem 4, implying that, for convergent judgments, we can reason by standard inductive techniques. In this case, we have a one-to-one correspondence between finite proof trees and convergent computations, and between infinite proof trees and divergent computations.

► **Theorem 5** (Conservativity of $e \rightsquigarrow \tilde{r}$). *If $e \rightsquigarrow \tilde{r}$ holds, then $\tilde{r} = (v, \tilde{o})$ if and only if the judgment has a finite proof tree.*

Proof. A finite proof tree for $e \rightsquigarrow \tilde{r}$ needs to start from the axiom $v \rightsquigarrow (v, [])$, hence $\tilde{r} = (v, \tilde{o})$. On the other hand, an infinite proof tree for $e \rightsquigarrow \tilde{r}$, since it adds a label to the result at each level, requires the output stream in the result to be infinite, therefore, thanks to the way we defined results, we get $\tilde{r} = (\infty, \tilde{o}_\infty)$. ◀

Observational equivalence. We discuss now how to define observational equivalence on streams. Intuitively, since τ labels represent non-observable actions, they should be ignored when comparing two streams: for instance, streams $[v_1 \tau v_2]$ and $[v_1 v_2]$ should be equivalent. Therefore we need to relax equality to another equivalence relation, denoted by \approx , coinductively defined by the inference system shown in Fig. 7, where τ^n is the stream of length n made of only τ s and τ^ω is the infinite stream of τ s.

The intuition behind this definition is that sequences of τ of arbitrary length are equivalent to $[]$ (as stated in rule (TAU)), hence they can be removed or added without changing the observable view (non- τ elements) of the stream. Note that the two rules are disjoint, since the consequence of the second rule requires an element of the stream to be different from τ on both sides, hence it may not happen that a stream made of only τ s is equivalent to a stream that contains non- τ elements. As in Fig. 6, the unique (meta)coaxiom where the consequence is any judgment corresponds to take the coinductive interpretation of the two rules.

This relation is indeed an equivalence, as formally stated below (the proof is in the Appendix).

► **Proposition 6.** *The relation \approx is an equivalence relation over (finite or infinite) streams.*

A more abstract view of the relation \approx can be given in categorical terms, namely as a bisimulation on a coalgebra structure carried by the set of streams. For the interested reader, this alternative definition is reported, and shown to be equivalent to the previous one, in the Appendix, Sect. A. To follow the next section, it is important to know that, following this definition, two streams are identified if and only if they are mapped to the same output stream by a function ε_τ that removes τ s from a stream, which can be described by the following equations:

$$\begin{cases} \varepsilon_\tau(\tau^\alpha) & = [] \\ \varepsilon_\tau(\tau^n \cdot [v] \cdot x) & = [v] \cdot \varepsilon_\tau(x) \end{cases}$$

Function ε_τ is employed in the next section to prove Theorem 7.

Now we have to extend this relation to results. Intuitively, two results are equivalent if they represent either both convergence with the same value, or both divergence, and the observable output streams are equivalent. More formally, we set

$$(v_\infty, \tilde{o}_\infty) \approx (v'_\infty, \tilde{o}'_\infty) \iff v_\infty = v'_\infty \text{ and } \tilde{o}_\infty \approx \tilde{o}'_\infty$$

It is immediate to check that this relation is also an equivalence. Note also that this extension can be defined by an inference system (interpreted coinductively) obtained from the definition of \approx on streams, by decorating each stream with the same (extended) value v_∞ , hence to prove $\tilde{r} \approx \tilde{r}'$ we can reason by coinduction.

Having introduced all this machinery, we can define that two expressions e and e' are *observationally equivalent* if $e \rightsquigarrow \tilde{r}$ implies $e' \rightsquigarrow \tilde{r}'$, for some $\tilde{r}' \approx \tilde{r}$, and conversely.

Consider, for instance, the value $id = \lambda x.x$ and the expressions $e_1 = \text{out}(id\ id)$ and $e_2 = (\text{out}\ id)\ id$. It is easy to see that the judgments $e_1 \rightsquigarrow (id, [\tau\ id])$ and $e_2 \rightsquigarrow (id, [id\ \tau])$ hold, and $[\tau\ id] \approx [id\ \tau]$, hence e_1 and e_2 are observationally equivalent.

Note that, considering the judgment $e \Rightarrow r$ defined in the previous section by using corules, the definition of observational equivalence reduces to semantic equivalence, that is, expressions e and e' are equivalent iff $e \Rightarrow r$ implies $e' \Rightarrow r$, and conversely. In other words, the judgment $e \Rightarrow r$ directly models the observable behaviour of programs. For instance, again it is easy to see that $e_1 \Rightarrow (id, [id])$ and $e_2 \Rightarrow (id, [id])$, hence e_1 and e_2 are equivalent also with respect to this semantics. Actually, as we will see in the next section, the two semantics can be shown to be equivalent.

5 Equivalence between the two semantics

In this section we will provide a complete⁴ proof of the equivalence between the semantics defined in Fig. 2 by using corules and that defined in Fig. 6 on top of the LTS in Fig. 1. We will briefly call them semantics by corules and LTS semantics, respectively. The main theorem is stated below.

► **Theorem 7 (Equivalence).**

1. If $e \Rightarrow r$, then there exists \tilde{r} such that $e \rightsquigarrow \tilde{r}$ and $r \approx \tilde{r}$.
2. If $e \rightsquigarrow \tilde{r}$, then there exists r such that $e \Rightarrow r$ and $\tilde{r} \approx r$.

We will prove separately the two parts of the theorem. Before providing such proofs we need to consider some properties relating the two semantics.

► **Lemma 8 (Progress).** If $e \Rightarrow r$ and e is not a value, then there exists e' and ℓ such that $e \xrightarrow{\ell} e'$.

Proof. Straightforward induction on the definition of e . ◀

► **Lemma 9 (Subject reduction).** If $e \Rightarrow r$ and $e \xrightarrow{\ell} e'$, then $e' \Rightarrow r'$ and $r \approx [\ell] \cdot r'$.

Proof. Straightforward induction on the rules defining $e \xrightarrow{\ell} e'$. ◀

The following lemma states that, if an expression converges in the semantics by corules and produces a (rough) result in the LTS semantics, then the LTS semantics converges as well with the same value.

⁴ Proofs of some auxiliary results are in the Appendix.

► **Lemma 10.** *If $e \Rightarrow (v, o)$ and $e \rightsquigarrow \tilde{r}$, then $\tilde{r} = (v, \tilde{o})$.*

Proof. Thanks to Theorem 4, we can reason by induction on the rules defining $e \Rightarrow (v, o)$, hence the proof is routine. ◀

In the next theorem, starting from a judgment $e_0 \Rightarrow r_0$ in the semantics by corules, we construct a sequence of reduction steps in the LTS, which can be either finite or infinite:

$$\begin{aligned} e_0 &\xrightarrow{\ell_0} e_1 \xrightarrow{\ell_1} \dots \xrightarrow{\ell_{n-1}} e_n \equiv v \\ e_0 &\xrightarrow{\ell_0} e_1 \xrightarrow{\ell_1} \dots e_n \xrightarrow{\ell_n} e_{n+1} \xrightarrow{\ell_{n+1}} \dots \end{aligned}$$

(for uniformity, a finite reduction sequence is represented by a reduction sequence until $e_n \equiv v$ and then an infinite sequence of v). For each n , we also construct a result r_n and a rough result \tilde{r}_n such that $e_n \Rightarrow r_n$, $e_n \rightsquigarrow \tilde{r}_n$, and the first component (value or divergence) in r_n and \tilde{r}_n is the same. Thus, in particular, there exists \tilde{r}_0 so that $e_0 \rightsquigarrow \tilde{r}_0$ and the first component (value or divergence) in r_0 and \tilde{r}_0 is the same.

► **Theorem 11.** *If $e_0 \Rightarrow r_0$, then there exist sequences $(e_n)_{n \in \mathbb{N}}$ of expressions, $(r_n)_{n \in \mathbb{N}}$ of results, and $(\tilde{r}_n)_{n \in \mathbb{N}}$ of rough results, such that, for all $n \in \mathbb{N}$*

1. $e_n \Rightarrow r_n$ and
 - if $e_n = v$, then $e_{n+1} = v$ and $r_n = r_{n+1} = \tilde{r}_n = \tilde{r}_{n+1} = (v, [])$,
 - otherwise $e_n \xrightarrow{\ell} e_{n+1}$ and $r_n \approx [\ell] \cdot r_{n+1}$ and $\tilde{r}_n = [\ell] \cdot \tilde{r}_{n+1}$
2. $e_n \rightsquigarrow \tilde{r}_n$
3. if $r_n = (v_\infty, o_\infty^n)$, then $\tilde{r}_n = (v_\infty, \tilde{o}_\infty^n)$.

Note that, by combining point 1 and point 3, we get that all the results in both $(r_n)_{n \in \mathbb{N}}$ and $(\tilde{r}_n)_{n \in \mathbb{N}}$ have the same first component (final value or divergence), since they are related by an equivalence that on the first component is the equality.

In order to state the next lemma, we introduce the auxiliary judgment $e \xrightarrow{\tilde{o}}_\star e'$, stating that e reduces to e' in finitely many steps producing the (finite) stream \tilde{o} . The judgment is inductively defined as follows:

$$\frac{}{e \xrightarrow{[]}_\star e} \quad \frac{e' \xrightarrow{\tilde{o}}_\star e''}{e \xrightarrow{[\ell] \cdot \tilde{o}}_\star e''} \quad e \xrightarrow{\ell} e'$$

It is easy to check that $e \rightsquigarrow \tilde{o} \cdot \tilde{r}$ holds if and only if $e \xrightarrow{\tilde{o}}_\star e'$ and $e' \rightsquigarrow \tilde{r}$ for some e' .

► **Lemma 12.** *If $e \Rightarrow [v] \cdot r$, then there is a finite stream \tilde{o} such that $e \xrightarrow{\tilde{o}}_\star \mathcal{E}[\text{out } v]$.*

Intuitively, the above lemma ensures that, if in the semantics by corules we produce an output, then in the LTS semantics we will reduce in finitely many steps to (an expression which contains) a corresponding output expression.

We are now ready to prove the first part of Theorem 7.

Proof. (Theorem 7 (1))

By Theorem 11 (points 1 and 2) we know that there are sequences $(e_n)_{n \in \mathbb{N}}$, $(r_n)_{n \in \mathbb{N}}$ and $(\tilde{r}_n)_{n \in \mathbb{N}}$ such that $e_0 = e$, $r_0 = r$ and $e_0 \rightsquigarrow \tilde{r}_0$. Therefore we have only to prove that $r_0 \approx \tilde{r}_0$. To this aim, we prove by coinduction that, for all $n \in \mathbb{N}$, $r_n \approx \tilde{r}_n$. By Theorem 11 (point 3) we already know that on the first component \tilde{r}_n and r_n are equal, hence we have only to reason on the stream part. Again by Theorem 11 (point 1) we also know that for all $n \in \mathbb{N}$, either $r_n \approx [\ell_n] \cdot r_{n+1}$ and $\tilde{r}_n = [\ell_n] \cdot \tilde{r}_{n+1}$, or $r_n = \tilde{r}_n = (v, [])$. So, considering $r_n = (v_\infty, o_\infty^n)$ and $\tilde{r}_n = (v_\infty, \tilde{o}_\infty^n)$, we have three cases.

- If $\tilde{r}_n = (v, [])$, then $e_n = v$, hence $\tilde{r}_n = r_n = (v, [])$ by Theorem 11 (1), thus they are equivalent by axiom (TAU) in Fig. 7.
- If $\tilde{r}_n = (v_\infty, \tau^\alpha)$ for $\alpha \in \mathbb{N} \cup \{\omega\}$ with $\alpha \neq 0$, then for all $n \leq k < n + \alpha$ we have that ℓ_k exists (by Theorem 11 (1) we have $e_k \xrightarrow{\ell_k} e_{k+1}$) and $\ell_k = \tau$. We have to prove that $r_n = (v_\infty, [])$. Let us assume that $r_n = [v] \cdot r'$, hence, since by Theorem 11 (1) $e_n \Rightarrow r_n$ holds, by Lemma 12 we get that there is a finite stream \tilde{o} such that $e_n \xrightarrow{\tilde{o}} \mathcal{E}[\text{out } v]$ and by the determinism of $\xrightarrow{\ell}$ we get that there is $m \geq n$ such that $e_m = \mathcal{E}[\text{out } v]$. Therefore we get that $e_m \xrightarrow{v} e_{m+1} = \mathcal{E}[v]$ (rules (OUT) and (CTX) in Fig. 1), hence $\ell_m = v \neq \tau$ that is a contradiction; thus $r_n = (v_\infty, [])$. Therefore we get the thesis by the first axiom.
- If $\tilde{r}_n = \tau^k \cdot [v] \cdot \tilde{r}_{n+k+1}$, then $\ell_{n+k} = v$ and for all $n \leq l < n + k$ we have $\ell_l = \tau$. Therefore we get by Theorem 11 (1) that $r_n \approx \tau^k \cdot [v] \cdot r_{n+k+1}$, hence $r_n = [v] \cdot r_{n+k+1}$. Finally by coinductive hypothesis we know that $r_{n+k+1} \approx \tilde{r}_{n+k+1}$ and thus we get the thesis by the following rule:

$$\frac{r_{n+k+1} \approx \tilde{r}_{n+k+1}}{r_n \equiv [v] \cdot r_{n+k+1} \approx \tau^k \cdot [v] \cdot \tilde{r}_{n+k+1} \equiv \tilde{r}_n} \quad \blacktriangleleft$$

In order to prove the point 2 of Theorem 7 we treat separately the cases of convergence and divergence. To prove the theorem we will show that if $e \rightsquigarrow \tilde{r}$ holds, then $e \Rightarrow \varepsilon_\tau(\tilde{r})$ holds too, where $\varepsilon_\tau(v_\infty, \tilde{o}_\infty) = (v_\infty, \varepsilon_\tau(\tilde{o}_\infty))$. This immediately proves the theorem, since $\varepsilon_\tau(\tilde{r})$ is the witness we need to prove the existential quantification, and by definition of \approx we have $\tilde{r} \approx \varepsilon_\tau(\tilde{r})$.

Denoting by $(\mathcal{I}, \mathcal{I}^{\text{co}})$ the generalized inference system defining the judgment $e \Rightarrow r$ in Fig. 2, in the following we will call “extended system” the inference system $\mathcal{I} \cup \mathcal{I}^{\text{co}}$, that is, the (standard) inference system that is the union of rules and corules.

First of all, recall from the previous section conservativity for $e \rightsquigarrow \tilde{r}$ (Theorem 5), that is, if $e \rightsquigarrow \tilde{r}$ holds, then $\tilde{r} = (v, \tilde{o})$ if and only if $e \rightsquigarrow \tilde{r}$ has a finite proof tree. This result allows us to reason by induction on rules defining convergence.

► **Lemma 13** (Subject expansion). *If $e \xrightarrow{\ell} e'$ and $e' \Rightarrow r'$ has a finite derivation in the extended system, then $e \Rightarrow r$ has a finite derivation in the extended system and $r \approx [\ell] \cdot r'$.*

Proof. Straightforward induction on rules defining $\xrightarrow{\ell}$. ◀

► **Theorem 14.** *If $e \rightsquigarrow (v, \tilde{o})$, then $e \Rightarrow (v, \varepsilon_\tau(\tilde{o}))$.*

Proof. By induction on the rules defining $e \rightsquigarrow \tilde{r}$.

- If $v \rightsquigarrow (v, [])$, then the thesis follows from rule (VAL) .
- If $e \rightsquigarrow [\ell] \cdot \tilde{r}$, $e \xrightarrow{\ell} e'$ and $e' \rightsquigarrow \tilde{r}$, then by inductive hypothesis we get $e' \Rightarrow \varepsilon_\tau(\tilde{r})$, hence by Lemma 13 we get $e \Rightarrow r$ with $r \approx [\ell] \cdot \varepsilon_\tau(\tilde{r})$ and this implies, by definition of \approx , that $r = \varepsilon_\tau([\ell] \cdot \tilde{r})$. ◀

Now let us consider the case of divergence.

► **Lemma 15.** $\mathcal{E}[\text{out } v] \Rightarrow (\infty, [v] \cdot o_\infty)$ has a finite derivation in the extended system.

Proof. By induction on the definition of contexts.

□ We get the thesis by rules (VAL) and (CO-OUT) .

out $\mathcal{E}[\text{out } v]$ We get the thesis by inductive hypothesis and rule (DIV) .

$\mathcal{E}[\text{out } v]$ e We get the thesis by inductive hypothesis and rule (DIV) .

v' $\mathcal{E}[\text{out } v]$ We get the thesis by rule (VAL) , inductive hypothesis and rule (DIV) . ◀

► **Theorem 16.** *If $e \rightsquigarrow (\infty, \tilde{d}_\infty)$, then $e \Rightarrow (\infty, \varepsilon_\tau(\tilde{d}_\infty))$ has a finite derivation in the extended system.*

Proof. We have two cases:

- if $\tilde{d}_\infty = \tau^\alpha$ with $\alpha \in \mathbb{N} \cup \{\omega\}$, then $\varepsilon_\tau(\tilde{d}_\infty) = []$, hence the thesis follows from the rule (CO-EMPTY);
- otherwise, we have $\tilde{d}_\infty = \tau^n \cdot [v] \cdot \tilde{d}'_\infty$, hence by definition of $e \rightsquigarrow \tilde{r}$, we get that $e \xrightarrow{\tau^n}_* \mathcal{E}[\text{out } v]$ and $\mathcal{E}[\text{out } v] \rightsquigarrow (\infty, [v] \cdot \tilde{d}'_\infty)$; therefore, by Lemma 15 we get that $\mathcal{E}[\text{out } v] \Rightarrow (\infty, [v] \cdot \varepsilon_\tau(\tilde{d}'_\infty))$ is derivable by a finite proof tree in the extended system, then by a transitive closure of Lemma 13 we get the thesis, since $\varepsilon_\tau(\tilde{d}_\infty) = [v] \cdot \varepsilon_\tau(\tilde{d}'_\infty)$. ◀

Proof. (Theorem 7 (2)) We show by bounded coinduction (Theorem 3) that if $e \rightsquigarrow \tilde{r}$, then $e \Rightarrow \varepsilon_\tau(\tilde{r})$. The boundedness condition immediately follows from Theorem 14, Theorem 4 and Theorem 16. Let us proceed with the coinductive step: if $\tilde{r} = (v, \tilde{d})$, then the thesis follows from Theorem 14, hence we assume that $\tilde{r} = (\infty, \tilde{d}_\infty)$ and proceed by case analysis on e .

v Empty case.

x Empty case.

out e Since **out** $e \rightsquigarrow (\infty, \tilde{d}_\infty)$ holds, we get that $e \rightsquigarrow (\infty, \tilde{d}_\infty)$ holds, hence by coinductive hypothesis $e \Rightarrow (\infty, \varepsilon_\tau(\tilde{d}_\infty))$ and by rule (DIV) we get the thesis.

e_1 e_2 We have three cases:

- If $e_1 \rightsquigarrow (\infty, \tilde{d}_\infty)$, then by coinductive hypothesis we get $e_1 \Rightarrow (\infty, \varepsilon_\tau(\tilde{d}_\infty))$, hence we get the thesis by rule (DIV).
- If $e_1 \rightsquigarrow (v, \tilde{d})$ and $e_2 \rightsquigarrow (\infty, \tilde{d}'_\infty)$, then $\tilde{d}_\infty = \tilde{d} \cdot \tilde{d}'_\infty$, by Theorem 14 and coinductive hypothesis we get that $e_1 \Rightarrow (v, \varepsilon_\tau(\tilde{d}))$ and $e_2 \Rightarrow (\infty, \varepsilon_\tau(\tilde{d}'_\infty))$, hence we get the thesis by rule (DIV) since $\varepsilon_\tau(\tilde{d}_\infty) = \varepsilon_\tau(\tilde{d}) \cdot \varepsilon_\tau(\tilde{d}'_\infty)$.
- If $e_1 \rightsquigarrow (\lambda x.e, \tilde{d}_1)$ and $e_2 \rightsquigarrow (v_2, \tilde{d}_2)$, then we have that $e_1 \xrightarrow{\tilde{d}_1}_* \lambda x.e$ and $e_2 \xrightarrow{\tilde{d}_2}_* v_2$, hence $e_1 \xrightarrow{\tilde{d}_1}_* \lambda x.e \xrightarrow{\tilde{d}_2}_* \lambda x.e \cdot v_2 \xrightarrow{\tau}_* e[x \leftarrow v_2] = e'$ with $e' \rightsquigarrow (\infty, \tilde{d}'_\infty)$, and $\tilde{d}_\infty = \tilde{d}_1 \cdot \tilde{d}_2 \cdot \tilde{d}'_\infty$. Therefore by Theorem 14 and coinductive hypothesis we get $e_1 \Rightarrow (\lambda x.e, \varepsilon_\tau(\tilde{d}_1))$, $e_2 \Rightarrow (v_2, \varepsilon_\tau(\tilde{d}_2))$ and $e' \Rightarrow (\infty, \varepsilon_\tau(\tilde{d}'_\infty))$, hence we get the thesis by rule (APP) since we have $\varepsilon_\tau(\tilde{d}_\infty) = \varepsilon_\tau(\tilde{d}_1) \cdot \varepsilon_\tau(\tilde{d}_2) \cdot \varepsilon_\tau(\tilde{d}'_\infty)$. ◀

6 A simple imperative Java-like language

In this section we provide the semantics by corules of an imperative Java-like language with **in** and **out** statements for reading and writing values, respectively. Our motivations for studying this language are the following.

- To check the approach on a (slightly) more realistic example than in Sect. 3. Notably, the language considered in this section is imperative and allows update of both local variables (in this simple calculus, just method parameters) and object fields, hence, stack frames and the heap need to be modeled. Moreover, since object values are heap references, they are read and written in a serialized format, and the serialization function needs to be coinductively defined.
- To investigate input as well. Managing both input and output requires deeper insights, since several approaches are possible; the one we propose, where I/O operations are treated as “events”, seems to be simpler and is easily extensible to other kinds of significant interactions of the program with the outside, as acquisition and release of resources.

$$\begin{aligned}
p &::= \overline{cd} \ e \\
cd &::= \text{class } c_1 \text{ extends } c_2 \{ \overline{fd} \ \overline{md} \} \\
fd &::= f; \\
md &::= m(\overline{x}) \{e\} \\
e &::= \text{new } c(\overline{e}) \mid x \mid \text{false} \mid \text{true} \mid e.f \mid e_0.m(\overline{e}) \mid x=e \mid e_1.f=e_2 \mid \text{in} \mid \text{out } e \\
&\quad \mid \text{if } (e) \ e_1 \text{ else } e_2
\end{aligned}$$

■ **Figure 8** Java-like language: syntax.

$$\begin{aligned}
v, u &::= \text{false} \mid \text{true} \mid \iota && \text{(internal) value} \\
v &::= \text{false} \mid \text{true} \mid \text{obj}(c, \overline{f}^n \mapsto \overline{v}^n) && \text{serialized value} \\
e &::= \text{in } v \mid \text{out } v && \text{event} \\
\theta &::= [e_1 \dots e_n] \mid [e_1 \dots e_n \dots] && \text{event trace} \\
r &::= (v, \theta); \Pi; \mathcal{H} \mid (\infty, \theta) && \text{result} \\
\mathcal{D}[\] &::= \text{new } c(\overline{\square}^n, \overline{e}) \mid \square.f \mid \square.m(\overline{\square}^k, \overline{e}) \quad (n > 0, k \geq 0) && \text{propagation context} \\
&\quad \mid x=\square \mid \square.f=e \mid \square.f=\square \mid \text{out } \square \mid \text{if } (\square) \ e_1 \text{ else } e_2
\end{aligned}$$

■ **Figure 9** Java-like language: values, results and propagation contexts.

The syntax of the language is defined in Fig. 8. We write \overline{cd}^n , or simply \overline{cd} , for the sequence $cd_1 \dots cd_n$, and analogously for other sequences. With abused notation $\overline{f}^n \mapsto \overline{v}^n$ stands for $f_1 \mapsto v_1, \dots, f_n \mapsto v_n$. We assume sets of *variables (parameters)* x , including **this**, *class names* c , including **Object**, *field names* f , and *method names* m .

The calculus is an imperative untyped Java-like language, where variables and fields can be updated; since the work is focused on the dynamic semantics, we have simplified the language by omitting type annotations.

For simplicity, statements are expressions with side effects; assignments to variables and fields return the value of the right-hand side expression, the **in** statement returns the deserialized value that has been acquired from the input, while the **out** e statement returns the value obtained from the evaluation of e , which is also output.

We assume standard syntactic restrictions: inheritance is acyclic, names are distinct in class, method, field, and parameter declarations, **Object** cannot be declared, class names other than **Object** that are referred in expressions must be declared.

A program consists of a sequence of class declarations and a main expression; class declarations contain field and method definitions, whereas a single constructor is implicitly defined, with parameters corresponding to the inherited and defined fields, with precedence to the inherited ones. In a method body the target object is accessed via the implicit read-only parameter **this**, which is assumed to differ from all explicitly declared parameters. Statement expressions include instance creation, variable, boolean literals, field selection, method invocation, variable and field update, input/output operations, and conditional expression.

We define now the semantics by corules of the language; as for the λ -calculus in Sect. 3, we provide a semantics with the fully deterministic left-to-right call-by-value evaluation strategy. Internal and serialized values, events, event traces, results and (divergence) propagation contexts are defined in Fig. 9. As for the λ -calculus, propagation contexts can contain many holes, all at fixed depth 1, and allow a more compact definition of propagation of diverging results (see rule (DIV) in Fig. 10). We assume an infinite set of *object references* ι . Internal

values are either boolean constants or object references; for boolean values the serialized form is the same, whereas object references are serialized to values of shape $obj(c, \bar{f}^n \mapsto \bar{v}^n)$, with c the class of the object, and \bar{f}^n its fields associated with their corresponding serialized values \bar{v}^n ; serialized objects are allowed to be regular terms, to allow serialization of cyclic objects.

Events that are tracked by the semantics are input/output operations, hence, they have shape $\text{in } v$ (input of serialized value v) and $\text{out } v$ (output of serialized value v). Traces are finite or infinite sequences of tracked events.

Results of converging computations are triples $(v, \theta); \Pi; \mathcal{H}$ where the first component is a pair (v, θ) consisting of a returned value, and a produced finite event trace, while the second and third ones are the stack frame Π and the heap \mathcal{H} yielded by the computation, respectively. Results model also diverging computations with pairs of shape (∞, θ) , where the event trace θ is allowed to be infinite; in case of diverging computations, neither returned value is defined, nor stack frame and heap are yielded. The stack frame of the method under execution is a finite partial map Π from variables (`this` and the explicit parameters of the method) to values. The heap \mathcal{H} is a finite partial map from references to objects. Objects are pairs $obj(c, \rho)$, where c is the object's class and ρ is a finite partial map from field names to values. We use the usual set-theoretic representation for finite maps, in particular for heaps \mathcal{H} , maps ρ from fields to values, and maps from fields to serialized values in serialized objects; the operator \uplus denotes union of maps with disjoint domains. We assume that heaps cannot contain dangling references: for all references ι , if $\mathcal{H}(\iota) = obj(c, \rho)$, then for all $f \in \text{dom}(\rho)$, $\rho(f) \in \text{dom}(\mathcal{H}) \uplus \{\text{false}, \text{true}\}$; indeed, in the language object references cannot be explicitly deallocated.

The semantics is formalized by the judgment $\Pi; \mathcal{H}; e \Rightarrow r$, where r is either $(v, \theta); \Pi'; \mathcal{H}'$ or (∞, θ) . In the former case, the judgment means that, in stack frame Π and heap \mathcal{H} , expression e converges to value v , with finitely generated event trace θ , and yielded stack frame Π' and heap \mathcal{H}' ; in the latter, expression e diverges with possibly infinite event trace θ .

The semantic rules are defined in Fig. 10; for brevity, we leave implicit the index of the judgment that should consist of the class declarations contained in the program.

The definitions of all auxiliary functions used in the side conditions of the rules can be found in the Appendix.

As in Fig. 2, rule (DIV) propagates diverging computations: if the evaluation of a subexpression diverges before all remaining subexpressions are evaluated, then the control flow of the program has to be modified; this happens in all situations captured by the propagation contexts defined in Fig. 9.

Rules (VAR) and (BOOL) are straightforward: the computation always converges and returns an empty trace.

Rules (NEW) and (FLD) are standard; in (NEW) the generated trace is obtained by concatenating in the same order the traces returned by the evaluation of the argument expressions; the disjoint union \uplus ensures that ι is a fresh reference in the current heap \mathcal{H}_n , the side condition requires that the arguments match all fields inherited and declared by class c . In (FLD) the generated trace corresponds to that obtained from the evaluation of the subexpression denoting the target object; the side condition ensures that such an evaluation returns an object reference defined in the heap and containing the accessed field.

Similarly to rule (APP) for function application in Fig. 2, rule (INV) deals with method invocation when the evaluation of the target and all arguments converges; the auxiliary function *restore* (see the comments to Fig. 12 in the Appendix) is used for restoring the current stack frame Π_{n+1} of the caller yielded after the evaluation of the target and the

$$\begin{array}{c}
\text{(DIV)} \frac{\forall i = 1..n. \Pi_{i-1}; \mathcal{H}_{i-1}; e_i \Rightarrow (v_i, \theta_i); \Pi_i; \mathcal{H}_i \quad \Pi_n; \mathcal{H}_n; e \Rightarrow (\infty, \theta)}{\Pi_0; \mathcal{H}_0; \mathcal{D}[\bar{e}^n, e] \Rightarrow (\infty, \theta_1 \cdot \dots \cdot \theta_n \cdot \theta)} \\
\\
\text{(VAR)} \frac{}{\Pi; \mathcal{H}; x \Rightarrow (v, [\]); \Pi; \mathcal{H}} \quad \text{(BOOL)} \frac{}{\Pi; \mathcal{H}; e \Rightarrow (e, [\]); \Pi; \mathcal{H}} \quad e \in \{\mathbf{false}, \mathbf{true}\} \\
\\
\text{(NEW)} \frac{\forall i = 1..n. \Pi_{i-1}; \mathcal{H}_{i-1}; e_i \Rightarrow (v_i, \theta_i); \Pi_i; \mathcal{H}_i}{\Pi_0; \mathcal{H}_0; \mathbf{new} \ c(\bar{e}^n) \Rightarrow (\iota, \theta_1 \cdot \dots \cdot \theta_n); \Pi_n; \mathcal{H}_n \uplus \{\iota \mapsto \mathit{obj}(c, \bar{f}^n \mapsto \bar{v}^n)\}} \quad \mathit{fields}(c) = \bar{f}^n \\
\\
\text{(FLD)} \frac{\Pi_0; \mathcal{H}_0; e \Rightarrow (\iota, \theta); \Pi_1; \mathcal{H}_1}{\Pi_0; \mathcal{H}_0; e.f \Rightarrow (v, \theta); \Pi_1; \mathcal{H}_1} \quad \mathcal{H}_1(\iota) = \mathit{obj}(c, \rho \uplus \{f \mapsto v\}) \\
\\
\text{(INV)} \frac{\forall i = 0..n. \Pi_i; \mathcal{H}_i; e_i \Rightarrow (v_i, \theta_i); \Pi_{i+1}; \mathcal{H}_{i+1} \quad \{\mathbf{this} \mapsto v_0, \bar{x}^n \mapsto \bar{v}^n\}; \mathcal{H}_{n+1}; e \Rightarrow r}{\Pi_0; \mathcal{H}_0; e_0.m(\bar{e}) \Rightarrow r'} \quad \begin{array}{l} \mathcal{H}_1(v_0) = \mathit{obj}(c, \rho) \\ \mathit{meth}(c, m) = \bar{x}^n.e \\ r' = \mathit{restore}(\Pi_{n+1}, (\theta_0 \cdot \dots \cdot \theta_n) \cdot r) \end{array} \\
\\
\text{(VAS)} \frac{\Pi_0; \mathcal{H}_0; e \Rightarrow (v, \theta); \Pi_1 \uplus \{x \mapsto u\}; \mathcal{H}_1}{\Pi_0; \mathcal{H}_0; x = e \Rightarrow (v, \theta); \Pi_1 \uplus \{x \mapsto v\}; \mathcal{H}_1} \\
\\
\text{(FAS)} \frac{\Pi_0; \mathcal{H}_0; e_1 \Rightarrow (\iota, \theta_1); \Pi_1; \mathcal{H}_1 \quad \Pi_1; \mathcal{H}_1; e_2 \Rightarrow (v, \theta_2); \Pi_2; \mathcal{H} \uplus \{\iota \mapsto \mathit{obj}(c, \rho \uplus \{f \mapsto u\})\}}{\Pi_0; \mathcal{H}_0; e_1.f = e_2 \Rightarrow (v, \theta_1 \cdot \theta_2); \Pi_2; \mathcal{H} \uplus \{\iota \mapsto \mathit{obj}(c, \rho \uplus \{f \mapsto v\})\}} \\
\\
\text{(IN)} \frac{}{\Pi; \mathcal{H}_0; \mathbf{in} \Rightarrow (v, [\mathbf{in} v]); \Pi; \mathcal{H}_1} \quad \mathit{deser}(\mathcal{H}_0, v) = (\mathcal{H}_1, v) \\
\\
\text{(OUT)} \frac{\Pi_0; \mathcal{H}_0; e \Rightarrow (v, \theta); \Pi_1; \mathcal{H}_1}{\Pi_0; \mathcal{H}_0; \mathbf{out} \ e \Rightarrow (v, \theta \cdot [\mathbf{out} v]); \Pi_1; \mathcal{H}_1} \quad \mathit{ser}(\mathcal{H}_1, v) = v \\
\\
\text{(IF)} \frac{\Pi_0; \mathcal{H}_0; e \Rightarrow (v, \theta); \Pi_1; \mathcal{H}_1 \quad \Pi_1; \mathcal{H}_1; e_i \Rightarrow r}{\Pi_0; \mathcal{H}_0; \mathbf{if} \ (e) \ e_1 \ \mathbf{else} \ e_2 \Rightarrow \theta \cdot r} \quad v = \mathbf{true} \wedge i = 1 \vee v = \mathbf{false} \wedge i = 2 \\
\\
\text{(CO-EMPTY)} \frac{}{\Pi; \mathcal{H}; e \Rightarrow (\infty, [\])} \quad \text{(CO-IN)} \frac{}{\Pi; \mathcal{H}; \mathbf{in} \Rightarrow (\infty, [\mathbf{in} v] \cdot \theta)} \\
\\
\text{(CO-OUT)} \frac{\Pi; \mathcal{H}; e \Rightarrow (v, \theta'); \Pi'; \mathcal{H}'}{\Pi; \mathcal{H}; \mathbf{out} \ e \Rightarrow (\infty, \theta' \cdot [\mathbf{out} v] \cdot \theta)} \quad v = \mathit{ser}(\mathcal{H}', v)
\end{array}$$

■ **Figure 10** Java-like language: inference system with corules.

arguments of the method invocation. The body of the method is evaluated in the new stack frame $\{\mathbf{this} \mapsto v_0, \bar{x}^n \mapsto \bar{v}^n\}$ defining \mathbf{this} , and the formal parameters; its result can correspond to either a converging or a diverging computation. In the latter case, r has shape (∞, θ) , with θ possibly infinite, the stack frame is not restored, but divergence is simply propagated to the conclusion of the rule, after concatenating to the left of θ , in the same evaluation order, the finite traces obtained from the evaluation of the target and the arguments; hence $\mathit{restore}(\Pi_{n+1}, (\theta_0 \cdot \dots \cdot \theta_n) \cdot (\infty, \theta)) = (\infty, \theta_0 \cdot \dots \cdot \theta_n \cdot \theta)$. The other side conditions are standard and ensure that the target of the invocation is an object whose class has method m with n parameters.

Rules (VAS) and (FAS) manage variable and field updates, respectively; the former changes⁵ the stack frame, the latter the heap. Rule (VAS) is applicable only when the variable to update is defined in the current stack frame; the trace generated from the assignment corresponds to that obtained from the evaluation of the right-hand side subexpression. Rule (FAS) is applicable only when the target evaluates to a reference to an object containing the field to be updated; the trace generated from the assignment corresponds to the concatenation, in the same order, of the traces obtained from the evaluation of the target and right-hand side subexpression.

In rule (IN) a trace with the single event `in v` is generated, where v is allowed to be any possible serialized value that can be obtained from the input. The corresponding returned inner value v is the deserialization of v (see the comments to Fig. 12 in the Appendix); such an operation can extend the heap if v corresponds to an object. In this case a new internal object has to be allocated.

In rule (OUT) the trace is obtained by concatenating that returned by the evaluation of the subexpression with the singleton trace containing the event `out v`, where v is the serialization of the value v (see the comments to Fig. 12 in the Appendix) to be output.

Rule (IF) is standard; it is applicable only if the evaluation of the condition converges to a boolean value. However, the overall result is allowed to diverge; this happens when the evaluation of the selected branch does not terminate. In any case, the generated trace is obtained by concatenating the traces, in the same evaluation order, yielded by the evaluation of the condition and the selected branch.

As for the semantics of the λ -calculus in Sect. 3, corules filter out undesired behavior in case of non termination: results can only have shape (∞, θ) , with θ possibly infinite; diverging computations which do not involve input/output operations give rise to proof trees where the definition of the yielded trace is non productive, hence corule (CO-EMPTY) forces the returned trace to be empty. The remaining corules (IN) and (OUT) relax the constraint of corule (CO-EMPTY) by allowing traces of arbitrary length (including infinity) when infinitely many input or output operations are performed, respectively; indeed, in both corules no constraint is imposed on the rest of the yielded trace, represented by the metavariable θ . As in Fig. 2, the premise of corule (CO-OUT) ensures that the evaluation of the subexpression denoting the value to output converges, to ensure that the corresponding serialized value is actually output.

We consider now an example program (other two examples are provided in the Appendix).

Example 1

Let us consider the program consisting of the following class declaration

```
class C extends Object { m(x) { this.m(out in) } }
```

and the main expression $e = \text{new } C().m(\text{true})$; such a program diverges and produces an infinite trace with alternating input and output events s.t. each event `in v` is immediately followed by the event `out v`; this is formalized by the derivable judgment $\emptyset; \emptyset; e \Rightarrow (\infty, \theta_0)$, where $\theta_0 = [\text{in } v_0 \text{ out } v_0 \text{ in } v_1 \text{ out } v_1 \dots]$.

Fig. 11 shows how an infinite tree can be derived with the standard rules of Fig. 10; for simplicity the figure considers only the cases where input values are just primitive boolean

⁵ Of course, stack frame and heap can also be indirectly changed by the evaluation of the subexpressions of the statements.

$$\nabla_i = \frac{\frac{\text{(VAR)} \overline{\Pi_i; \mathcal{H}; \text{this} \Rightarrow (\iota, []); \Pi_i; \mathcal{H}} \quad \text{(OUT)} \overline{\Pi_i; \mathcal{H}; \text{out in} \Rightarrow (v_i, [\text{in } v_i \text{ out } v_i]); \Pi_i; \mathcal{H}} \quad \nabla_{i+1}}{\Pi_i; \mathcal{H}; \text{this.m}(\text{out in}) \Rightarrow (\infty, [\text{in } v_i \text{ out } v_i] \cdot \theta_{i+1})} \quad \text{(IN)} \overline{\Pi_i; \mathcal{H}; \text{in} \Rightarrow (v_i, [\text{in } v_i]); \Pi_i; \mathcal{H}}$$

$$\frac{\frac{\text{(NEW)} \overline{\emptyset; \emptyset; \text{new } C() \Rightarrow (\iota, []); \emptyset; \mathcal{H}} \quad \text{(BOOL)} \overline{\emptyset; \mathcal{H}; \text{true} \Rightarrow (\text{true}, []); \emptyset; \mathcal{H}} \quad \nabla_0}{\emptyset; \emptyset; \text{new } C().\text{m}(\text{true}) \Rightarrow (\infty, \theta_0)} \quad \text{(INV)}$$

where $\forall i \in \mathbb{N}. \theta_i = [\text{in } v_i \text{ out } v_i] \cdot \theta_{i+1}$, $\mathcal{H} = \{\iota \mapsto \text{obj}(C, \emptyset)\}$, $\Pi_0 = \{\text{this} \mapsto \iota, \mathbf{x} \mapsto \text{true}\}$,
 $\Pi_{i+1} = \{\text{this} \mapsto \iota, \mathbf{x} \mapsto v_i\}$, $v_i \in \{\text{false}, \text{true}\}$

■ **Figure 11** Infinite derivation for $\emptyset; \emptyset; \text{new } C().\text{m}(\text{true}) \Rightarrow (\infty, \theta_0)$ in Example 1.

values and, hence, no heap memory is allocated when deserialization occurs every time a new value is read from the input through the **in** instruction. Hence, after the allocation triggered by the execution of **new** $C()$, the heap \mathcal{H} remains unchanged; in case infinite serialized objects are read from the input, the heaps in the derived judgments of the proof tree grow indefinitely with the depth of the tree.

Although in Fig. 11 only primitive input values are considered, in general the derived infinite proof tree is not regular and consists of an infinite set of subtrees $\nabla_0, \nabla_1, \dots$, each one depending from the particular input values v_0, v_1, \dots ; since at each call a value is input and then output, the definition for the trace θ_0 in the finally derived judgment is fully productive and no trace other than $[\text{in } v_0 \text{ out } v_0 \text{ in } v_1 \text{ out } v_1 \dots]$ can be derived; since such a trace is infinite, it is not possible to derive judgments of shape $\emptyset; \emptyset; \text{new } C().\text{m}(\text{true}) \Rightarrow (v, \theta_0); \emptyset; \mathcal{H}'$.

For each occurrence of rule (INV) in the infinite proof tree, only finite trees can be built for the evaluation of the target and argument of the method invocation, therefore for the corresponding premises only judgments for converging computations can be derived, and rule (DIV) cannot be applied in place of rule (INV).

Finally, to show that the proof tree in Fig. 11 is valid, we need to prove that by using also the corules we can derive finite proof trees for all judgments of the proof. To this aim, we can prove that for all judgments $\Pi_i; \mathcal{H}; \text{this.m}(\text{out in}) \Rightarrow (\infty, [\text{in } v_i \text{ out } v_i] \cdot \theta_{i+1})$ derived with ∇_i , it is possible to build the following finite tree:

$$\frac{\frac{\text{(VAR)} \overline{\Pi_i; \mathcal{H}; \text{this} \Rightarrow (\iota, []); \Pi_i; \mathcal{H}} \quad \text{(CO-OUT)} \overline{\Pi_i; \mathcal{H}; \text{out in} \Rightarrow (\infty, [\text{in } v_i \text{ out } v_i] \cdot \theta_{i+1})}}{\Pi_i; \mathcal{H}; \text{this.m}(\text{out in}) \Rightarrow (\infty, [\text{in } v_i \text{ out } v_i] \cdot \theta_{i+1})} \quad \text{(DIV)} \quad \text{(IN)} \overline{\Pi_i; \mathcal{H}; \text{in} \Rightarrow (v_i, [\text{in } v_i]); \Pi_i; \mathcal{H}}$$

Additional examples can be found in the Appendix.

We illustrate now by a simple example how to use the bounded coinduction technique (Theorem 3) to reason about concrete programs. Consider the following Java-like program:

```
class T extends Object{hasNext(){true}}
class F extends Object{hasNext(){false}}
class Main extends Object {
  loop(){if(in.hasNext()) this.loop() else false}
}
```

and abbreviate by **F** and **T**, respectively, the two input events $\text{in } \text{obj}(\text{F}, \emptyset)$ and $\text{in } \text{obj}(\text{T}, \emptyset)$.

Intuitively, there are two possible valid results of the program $e = \text{new } \text{Main}().\text{loop}()$. If the input provides infinitely many **T**'s, e loops forever, producing the corresponding infinite trace $[\text{T} \dots \text{T} \dots]$, abbreviated T^∞ in the following. If, after n **T**'s, an **F** is eventually

read, then the program terminates returning **false**, and producing a finite trace $[T \dots T F]$, abbreviated T^n in the following. Note that, if any other kind of (serialized) value is read, then the program execution is stuck, that is, there is no (either infinite or finite) result. In our formalization, differently from what happens in standard big-step semantics, this case is nicely distinguished from divergence, since in the former case there is no proof tree.

More precisely, valid judgments for e have one of the following shapes:

1. (a) $\emptyset; \emptyset; e \Rightarrow (\infty, T^\infty)$
 (b) $\emptyset; \emptyset; e \Rightarrow (\mathbf{false}, T^n); \emptyset; \{\iota \mapsto \mathit{obj}(\mathbf{Main}, \emptyset)\} \uplus \mathcal{H}^n$
 with $\mathcal{H}^n = \{\iota_i \mapsto \mathit{obj}(T, \emptyset) \mid i \in 1..n\} \uplus \{\iota' \mapsto \mathit{obj}(F, \emptyset)\}$.

In order to formally prove that such judgments are derivable, as it is customary in (co)induction proofs, we have to extend the set of valid judgments, including all those which are needed for the coinductive hypothesis:

2. $\emptyset; \emptyset; \mathbf{new\ Main}() \Rightarrow (\iota, [\]); \emptyset; \{\iota \mapsto \mathit{obj}(\mathbf{Main}, \emptyset)\}$
3. (a) $\Pi; \mathcal{H}; \mathbf{if}(\mathbf{in.hasNext}()) \mathbf{this.loop}() \mathbf{else\ false} \Rightarrow (\infty, T^\infty)$
 (b) $\Pi; \mathcal{H}; \mathbf{if}(\mathbf{in.hasNext}()) \mathbf{this.loop}() \mathbf{else\ false} \Rightarrow (\mathbf{false}, T^n); \Pi; \mathcal{H} \uplus \mathcal{H}^n$
4. (a) $\Pi; \mathcal{H}; \mathbf{in.hasNext}() \Rightarrow (\mathbf{true}, [T]); \Pi; \mathcal{H} \uplus \{\iota \mapsto \mathit{obj}(T, \emptyset)\}$
 (b) $\Pi; \mathcal{H}; \mathbf{in.hasNext}() \Rightarrow (\mathbf{false}, [F]); \Pi; \mathcal{H} \uplus \{\iota \mapsto \mathit{obj}(F, \emptyset)\}$
5. (a) $\Pi; \mathcal{H}; \mathbf{in} \Rightarrow (\iota, [T]); \Pi; \mathcal{H} \uplus \{\iota \mapsto \mathit{obj}(T, \emptyset)\}$
 (b) $\Pi; \mathcal{H}; \mathbf{in} \Rightarrow (\iota, [F]); \Pi; \mathcal{H} \uplus \{\iota \mapsto \mathit{obj}(F, \emptyset)\}$
6. (a) $\Pi; \mathcal{H}; \mathbf{true} \Rightarrow (\mathbf{true}, [\]); \Pi; \mathcal{H}$ (b) $\Pi; \mathcal{H}; \mathbf{false} \Rightarrow (\mathbf{false}, [\]); \Pi; \mathcal{H}$
7. $\Pi; \mathcal{H}; \mathbf{this} \Rightarrow \iota; \Pi; \mathcal{H}$
8. (a) $\Pi; \mathcal{H}; \mathbf{this.loop}() \Rightarrow (\infty, T^\infty)$
 (b) $\Pi; \mathcal{H}; \mathbf{this.loop}() \Rightarrow (\mathbf{false}, T^n); \Pi; \mathcal{H} \uplus \mathcal{H}^n$

where $\Pi = \{\mathbf{this} \mapsto \iota\}$, $\mathcal{H}(\iota) = \mathit{obj}(\mathbf{Main}, \emptyset)$.

Set \mathcal{S} the set of judgments of shape 1-8, $(\mathcal{I}, \mathcal{I}^{\text{co}})$ the generalized inference system defining the judgment $\Pi; \mathcal{H}; e \Rightarrow r$ in Fig. 10, and $\mathcal{I} \cup \mathcal{I}^{\text{co}}$ the (standard) inference system that is the union of \mathcal{I} and \mathcal{I}^{co} . To prove by bounded coinduction (Theorem 3) that each judgment in \mathcal{S} can be derived, we have to show:

Boundedness. Each judgment in \mathcal{S} has a finite proof tree in $\mathcal{I} \cup \mathcal{I}^{\text{co}}$.

For convergent judgments (all cases except 1a, 3a, 8a) it is easy to show that they have a finite proof tree in \mathcal{I} , hence in $\mathcal{I} \cup \mathcal{I}^{\text{co}}$ as well. In particular, for cases 3b, 8b this finite proof tree can be constructed by arithmetic induction on n .

For cases 1a, 3a, 8a it is enough to show that a finite proof tree for $\Pi; \mathcal{H}; \mathbf{in} \Rightarrow (\infty, T^\infty)$ in $\mathcal{I} \cup \mathcal{I}^{\text{co}}$ can be obtained by corule (co-IN), analogously to what we have shown for Example 3 in Sect. 3.

Coinductive step. Each judgment in \mathcal{S} is the consequence of an inference rule in \mathcal{I} where all premises are in \mathcal{S} .

The proof can be done by cases. For instance, for case 3b, the judgment is the consequence of rule (IF) with first premise of shape 4 and second premise of shape either 8b or 6b.

We conclude this section with a comment on the proof technique outlined above. Here we have added in the set \mathcal{S} of valid judgments exactly those strictly needed for the coinductive hypothesis. This is a minimal choice. A different approach, which we used in [6] under the name *divergence consistency principle*, is to add to \mathcal{S} all the judgments which are derivable in \mathcal{I} . With this approach, it is enough to prove conservativity (the analogous of Theorem 4), and that each valid diverging judgment (cases 1a, 3a, 8a in our example) is the consequence of a rule where all diverging premises are valid as well, and all converging premises are derivable. This technique makes the proof schema simpler, but here we preferred the explicit approach for illustrating better the specific example.

7 Related work

One of the first approaches to model divergence with operational semantic rules was proposed by Cousot and Cousot [8] for the call-by-value λ -calculus by means of two different judgments defined in a stratified way: the former with inductive rules to model termination, the latter with coinductive rules and depending from the former, to capture divergence. In a followup work [9] they proposed a more sophisticated framework based on bi-inductive definitions, an order-theoretic approach to inductive definitions which allows the simultaneous definition of finite and infinite behaviors in operational semantics; however, such a solution has been adopted only for the standard call-by-value λ -calculus, and no uses of it can be found in literature for expressing the semantics of other languages.

Leroy and Grall [13] have investigated several operational semantics of the call-by-value λ -calculus for capturing divergence, their equivalence, and suitability to formally prove type soundness and compiler correctness; they are all defined in terms of inductive and coinductive judgments defined in a stratified way.

Coinductive big-step semantics has been proposed by Ancona to proof soundness for a Java-like language [3].

An operational semantics modeling divergence with an ad hoc coinductive judgment has been investigated also by Chargueraud with the notion [7] of pretty-big step semantics.

Flag-based big-step semantics [18] is a recent approach able to capture divergence by interpreting the same semantic rules both inductively and coinductively; flags represent termination or divergence and are part of the result of the computation. In case of non termination, values (or other semantic details exclusively used for terminating computations) are non deterministically returned.

We are planning to investigate the relationship between corules and conditional coinduction [10] employed by Danielsson to combine induction and coinduction in definitions of total parser combinators. Followup work [11], inspired by the paper of Leroy and Grall [13], shows how the use the coinductive partiality monad allow the definition of big/small-step semantics for lambda-calculi and virtual machines as total, computable functions able to capture divergence.

Several approaches to divergence with operational semantics [17, 2, 4] have been inspired by the notion of definitional interpreter [19]. All semantic definitions depend on a counter which limits the number of steps a computation can take; if the value of the counter is not sufficient to complete the computation, then a timeout is returned. In this way divergence can be modeled by induction.

Owens et al. [17] investigate functional big-step semantics for proving by induction compiler correction, including divergence preservation. Amin and Rompf [2] explore inductive proof strategies for type soundness properties for the polymorphic type systems $F_{<}$, and equivalence with small-step semantics. Ancona [4] proposes an inductive proof of type soundness for the big-step semantics of a Java-like language.

More recently, Ancona et al. [6] have shown that with coaxioms it is possible to mix together induction and coinduction in a single inference system to define a big-step operational semantics able to capture divergence with just one judgment. The call-by-value λ -calculus is considered, and a proof of equivalence with the standard small-step semantics is provided. Then the semantics of an imperative Java-like language is defined, and a corresponding type soundness result is proved.

All papers surveyed so far limit their investigation to semantics which capture divergence, but, differently to the contribution of this paper, do not provide any support for reasoning

on the behavior of non terminating programs, because in case of non termination the only information that is conveyed is divergence.

Nakata and Uustalu [14, 15, 16] have investigated on coinductive trace semantics in big-step style; they started with the semantics of an imperative While language with no I/O [14] where traces are possibly infinite sequences of states; semantic rules are all coinductive and define two mutually dependent judgments. Based on such a semantics, they define a Hoare logic [15]; differently to our approach, weak trace equivalence is required for proving that programs exhibit equivalent observable behaviors. A constructive theory and metatheory and a Coq formalization are provided.

The semantics has been subsequently extended with interactive I/O [16], by exploiting the notion of resumption: a tree representing possible runs of a program to model its non-deterministic behavior due to input values. Also in this case a big-step trace semantics is defined with two mutually recursive coinductive judgments, and weak bisimilarity is needed; however, the definition of the observational equivalence is more involved, since it requires nesting inductive definitions in coinductive ones.

In both papers [14, 15] equivalence of the big-step and small-step semantics is proved; also, in the considered languages [14, 15] expressions and statements are distinct, and expressions cannot diverge (divergence is not obtained through infinite recursion, but rather through infinite while loops). This is a significant difference with the languages we have considered in this paper; under the assumption that the evaluation of e cannot diverge, the semantics of $\text{out } e$ becomes simpler, indeed, the corresponding corule could be turned into a coaxiom.

8 Conclusion

We have shown how generalized inference systems can be employed for formalizing in a convenient way non trivial operational semantics suitable for reasoning on the behavior of possibly diverging programs.

The two examples of semantics we have provided suggest that by using corules with a similar pattern, other notions of interesting infinite behaviour can be modeled to reason on properties of diverging programs, to prove, for instance, that a server program uses a finite amount of system resources, even when expected to never stop.

As a byproduct, we have extended the theory of generalized inference systems with the more general notion of corule, for which significant examples were missing until now.

We briefly discuss now advantages, drawbacks, and limitations of the approach. While we do not claim that our approach is *easier* than the standard formulation using labeled small-step semantics, an important advantage is that the whole system is directly based on a unique judgment, thus allowing more direct reasoning (proofs).

A difficulty in adopting the approach could be how to define the right corules, which requires new expertise in comparison with the well established inductive reasoning (essentially based on case analysis). For this reason, we are currently working on the definition of “canonical” corule patterns. Notably, the two examples in this paper should become instances of a general transformation from an inductive inference system modeling finite behaviour to an inference system with corules modeling infinite behaviour as well.

Of course, inference systems with corules are not a silver bullet for expressing any kind of recursive definition. The main limitation we have found until now is in modeling, roughly, recursive functions where the result can be an infinite set. For instance, the inference system with corules for the graph example at page 4 is not complete when the set of nodes is infinite. A similar example (the infinite carrier of a list) is mentioned in [5]. We plan to investigate and possibly address this limitation.

For the first simpler example of semantics we have fully proved that our approach is equivalent to a semantics based on the standard notion of LTS and observational equivalence; we leave for future work a proof of equivalence for the imperative Java-like language. Although in this case the technical details are more complex, we do not expect any surprise in the proofs of equivalence.

Besides the already mentioned directions, it would be of great interest to try to test our approach with the support of a proof assistant.

References

- 1 Peter Aczel. An introduction to inductive definitions. In Jon Barwise, editor, *Handbook of Mathematical logic*, pages 739–782. North Holland, 1977.
- 2 Nada Amin and Tiark Rompf. Type soundness proofs with definitional interpreters. In Giuseppe Castagna and Andrew D. Gordon, editors, *ACM Symp. on Principles of Programming Languages 2017*, pages 666–679. ACM Press, 2017. doi:10.1145/3009837.
- 3 Davide Ancona. Soundness of object-oriented languages with coinductive big-step semantics. In James Noble, editor, *ECOOP’12 - Object-Oriented Programming*, volume 7313 of *Lecture Notes in Computer Science*, pages 459–483. Springer, 2012. doi:10.1007/978-3-642-31057-7_21.
- 4 Davide Ancona. How to prove type soundness of Java-like languages without forgoing big-step semantics. In David J. Pearce, editor, *FTfJP’14 - Formal Techniques for Java-like Programs*, pages 1:1–1:6. ACM Press, 2014. doi:10.1145/2635631.2635846.
- 5 Davide Ancona, Francesco Dagnino, and Elena Zucca. Generalizing inference systems by coaxioms. In Hongseok Yang, editor, *ESOP 2017 - European Symposium on Programming*, volume 10201 of *Lecture Notes in Computer Science*, pages 29–55. Springer, 2017. doi:10.1007/978-3-662-54434-1_2.
- 6 Davide Ancona, Francesco Dagnino, and Elena Zucca. Reasoning on divergent computations with coaxioms. *PACMPL*, 1(OOPSLA):81:1–81:26, 2017. doi:10.1145/3133905.
- 7 Arthur Charguéraud. Pretty-big-step semantics. In Matthias Felleisen and Philippa Gardner, editors, *ESOP 2013 - European Symposium on Programming*, volume 7792 of *Lecture Notes in Computer Science*, pages 41–60. Springer, 2013. doi:10.1007/978-3-642-37036-6_3.
- 8 Patrick Cousot and Radhia Cousot. Inductive definitions, semantics and abstract interpretations. In Ravi Sethi, editor, *ACM Symp. on Principles of Programming Languages 1992*, pages 83–94. ACM Press, 1992. doi:10.1145/143165.143184.
- 9 Patrick Cousot and Radhia Cousot. Bi-inductive structural semantics. *Information and Computation*, 207(2):258–283, 2009. doi:10.1016/j.ic.2008.03.025.
- 10 Nils Anders Danielsson. Total parser combinators. In *Intl. Conf. on Functional Programming 2010*, pages 285–296. ACM Press, 2010.
- 11 Nils Anders Danielsson. Operational semantics using the partiality monad. In Peter Thiemann and Robby Bruce Findler, editors, *Intl. Conf. on Functional Programming 2012*, pages 127–138. ACM Press, 2012. doi:10.1145/2364527.2364546.
- 12 Bart Jacobs. *Introduction to Coalgebra: Towards Mathematics of States and Observation*, volume 59 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2016. doi:10.1017/CB09781316823187.
- 13 Xavier Leroy and Hervé Grall. Coinductive big-step operational semantics. *Information and Computation*, 207(2):284–304, 2009. doi:10.1016/j.ic.2007.12.004.
- 14 Keiko Nakata and Tarmo Uustalu. Trace-based coinductive operational semantics for while. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *The-*

- orem Proving in Higher Order Logics - TPHOLs 2009*, volume 5674 of *Lecture Notes in Computer Science*, pages 375–390. Springer, 2009. doi:10.1007/978-3-642-03359-9_26.
- 15 Keiko Nakata and Tarmo Uustalu. A Hoare logic for the coinductive trace-based big-step semantics of while. In Andrew D. Gordon, editor, *ESOP 2010 - European Symposium on Programming*, volume 6012 of *Lecture Notes in Computer Science*, pages 488–506. Springer, 2010. doi:10.1007/978-3-642-11957-6_26.
 - 16 Keiko Nakata and Tarmo Uustalu. Resumptions, weak bisimilarity and big-step semantics for while with interactive I/O: an exercise in mixed induction-coinduction. In Luca Aceto and Pawel Sobocinski, editors, *SOS'10 - Structural Operational Semantics*, volume 32 of *Electronic Proceedings in Theoretical Computer Science*, pages 57–75, 2010. doi:10.4204/EPTCS.32.5.
 - 17 Scott Owens, Magnus O. Myreen, Ramana Kumar, and Yong Kiam Tan. Functional big-step semantics. In Peter Thiemann, editor, *ESOP 2016 - European Symposium on Programming*, volume 9632 of *Lecture Notes in Computer Science*, pages 589–615. Springer, 2016. doi:10.1007/978-3-662-49498-1_23.
 - 18 C. B. Poulsen and P. D. Mosses. Flag-based big-step semantics. *Journal of Logical and Algebraic Methods in Programming*, 2016.
 - 19 John C. Reynolds. Definitional interpreters for higher-order programming languages. In *ACM '72, Proceedings of the ACM annual conference*, volume 2, pages 717—740. ACM Press, 1972.
 - 20 Jan J. M. M. Rutten. Universal coalgebra: a theory of systems. *Theoretical Computer Science*, 249(1):3–80, 2000. doi:10.1016/S0304-3975(00)00056-6.

A A coalgebraic view

We recall some basic notions about coalgebras, see for instance [12, 20]. Given a category \mathcal{C} and an endofunctor $F: \mathcal{C} \rightarrow \mathcal{C}$, an F -coalgebra is a pair (C, γ) where C is an object of \mathcal{C} and $\gamma: C \rightarrow FC$ is an arrow in \mathcal{C} . An F -coalgebra homomorphism between two F -coalgebras (C, γ) and (C', γ') is an arrow $f: C \rightarrow C'$ in \mathcal{C} such that $\gamma' \cdot f = Ff \cdot \gamma$, where \cdot denotes the arrow composition in \mathcal{C} . It is easy to check that the composition of coalgebra homomorphisms is again an homomorphism and that the identity arrow on the carrier of an F -coalgebra is an homomorphism, hence F -coalgebras and homomorphisms form a category and the terminal object in this category, if any, is named *terminal F -coalgebra*.

We now fix $\mathcal{C} = \text{Set}$, that is, we work in the category of sets and functions. Given two F -coalgebras (C, γ) and (C', γ') , a *bisimulation* between them is a relation $R \subseteq C \times C'$ that carries an F -coalgebra structure such that the canonical projections $\pi_1: R \rightarrow C$ and $\pi_2: R \rightarrow C'$ are F -coalgebra homomorphisms. In other words, a bisimulation is a relation that agrees with the coalgebraic structure of its components.

Let us now introduce some notations. If A is a set, A^∞ is the set of finite and infinite streams over A . We denote by \mathcal{V} the set of values, and by \mathcal{V}_τ the set $\mathcal{V} + \{\tau\}$, where $+$ denotes the coproduct in the category Set of sets and functions.

It is well-known that \mathcal{V}^∞ is the carrier of the terminal coalgebra of $F: \text{Set} \rightarrow \text{Set}$, the functor defined by $FX = \mathbf{1} + \mathcal{V} \times X$, with the map $\zeta: \mathcal{V}^\infty \rightarrow F\mathcal{V}^\infty$ defined by

$$\zeta(x) = \begin{cases} (v, x') & \text{if } x = [v] \cdot x' \\ \star & \text{otherwise } (x = []) \end{cases}$$

where $\star \in \mathbf{1}$ is the unique element of the terminal object $\mathbf{1}$ in Set . However, we can give an

F -coalgebra structure also to \mathcal{V}_τ^∞ , considering the function $\gamma_\tau: \mathcal{V}_\tau^\infty \rightarrow F\mathcal{V}_\tau^\infty$ defined by

$$\gamma_\tau(x) = \begin{cases} (v, x') & \text{if } x = \tau^n \cdot [v] \cdot x' \\ \star & \text{otherwise (} x = \tau^\alpha \text{)} \end{cases}$$

Since $(\mathcal{V}^\infty, \zeta)$ is terminal, there is a unique F -coalgebra homomorphism $\varepsilon_\tau: \mathcal{V}_\tau^\infty \rightarrow \mathcal{V}^\infty$, in other words ε_τ is the unique map making the following diagram commute:

$$\begin{array}{ccc} \mathcal{V}_\tau^\infty & \xrightarrow{\varepsilon_\tau} & \mathcal{V}^\infty \\ \downarrow \gamma_\tau & & \downarrow \zeta \\ F\mathcal{V}_\tau^\infty & \xrightarrow{F\varepsilon_\tau} & F\mathcal{V}^\infty \end{array}$$

Intuitively, this diagram forces ε_τ to satisfy the equations mentioned in the beginning, hence to be the function that removes τ s from a stream.

In order to construct \approx , we consider the set R_\approx defined as the pullback in \mathbf{Set} of the pair of functions $(\varepsilon_\tau, \varepsilon_\tau)$, hence, since ε_τ is an F -coalgebra homomorphism, we get the following commutative diagram:

$$\begin{array}{ccccc} R_\approx & \xrightarrow{\pi_2} & \mathcal{V}_\tau^\infty & & \\ \downarrow \pi_1 \lrcorner & & \downarrow \varepsilon_\tau & \searrow \gamma_\tau & \\ \mathcal{V}_\tau^\infty & \xrightarrow{\varepsilon_\tau} & \mathcal{V}^\infty & & F\mathcal{V}_\tau^\infty \\ & \searrow \gamma_\tau & \downarrow \zeta & \searrow F\varepsilon_\tau & \\ & & F\mathcal{V}_\tau^\infty & \xrightarrow{F\varepsilon_\tau} & F\mathcal{V}^\infty \end{array}$$

More explicitly, R_\approx is the set $\{(x, y) \in \mathcal{V}_\tau^\infty \times \mathcal{V}_\tau^\infty \mid \varepsilon_\tau(x) = \varepsilon_\tau(y)\}$, thus it is an *equivalence relation* on \mathcal{V}_τ^∞ .

It is easy to check that F preserves pullbacks in \mathbf{Set} , hence we get the following commutative diagram:

$$\begin{array}{ccccc} R_\approx & & & & \\ \downarrow \gamma_\approx & \searrow \gamma_\tau \cdot \pi_2 & & & \\ FR_\approx & \xrightarrow{F\pi_2} & F\mathcal{V}_\tau^\infty & & \\ \downarrow F\pi_1 \lrcorner & & \downarrow F\varepsilon_\tau & & \\ F\mathcal{V}_\tau^\infty & \xrightarrow{F\varepsilon_\tau} & F\mathcal{V}^\infty & & \end{array}$$

Therefore $(R_\approx, \gamma_\approx)$ is an F -coalgebra, π_1 and π_2 are F -coalgebra homomorphisms and hence R_\approx is a *bisimulation equivalence* on $(\mathcal{V}_\tau^\infty, \gamma_\tau)$.

We now show that \approx and R_\approx are indeed the same relation.

► **Proposition 17.** $\tilde{o}_\infty \approx \tilde{o}'_\infty$ if and only if $(\tilde{o}_\infty, \tilde{o}'_\infty) \in R_\approx$.

B Proofs

Proof of Prop. 6 (\approx is an equivalence) We show that \approx is reflexive, symmetric and transitive.

Reflexivity We have to prove that, for each stream \tilde{o}_∞ , $\tilde{o}_\infty \approx \tilde{o}_\infty$ holds; we proceed by coinduction. We distinguish two cases:

- if $\tilde{o}_\infty = \tau^\alpha$ for some $\alpha \in \mathbb{N} \cup \{\omega\}$, then the thesis follows by the first rule, taking β equal to α ;

- otherwise, there is a value v such that $\tilde{d}_\infty = \tau^n \cdot [v] \cdot \tilde{d}'_\infty$, hence by coinductive hypothesis we get $\tilde{d}'_\infty \approx \tilde{d}_\infty$, and the thesis follows from the second rule, taking m equal to n .

Symmetry Assume $\tilde{d}_\infty \approx \tilde{d}'_\infty$, we have to prove that $\tilde{d}'_\infty \approx \tilde{d}_\infty$; we proceed by coinduction.

We distinguish two cases:

- if both \tilde{d}_∞ and \tilde{d}'_∞ are made of only τ s, then the thesis follows from the first rule (it is enough to swap the two exponents);
- otherwise, we have $\tilde{d}_\infty = \tau^n \cdot [v] \cdot \tilde{s}_\infty$ and $\tilde{d}'_\infty = \tau^m \cdot [v] \cdot \tilde{s}'_\infty$ with $\tilde{s}_\infty \approx \tilde{s}'_\infty$; hence by coinductive hypothesis we get $\tilde{s}'_\infty \approx \tilde{s}_\infty$, and the thesis follows by applying the rule

$$\frac{\tilde{s}'_\infty \approx \tilde{s}_\infty}{\tilde{d}'_\infty = \tau^m \cdot [v] \cdot \tilde{s}'_\infty \approx \tau^n \cdot [v] \cdot \tilde{s}_\infty = \tilde{d}_\infty}$$

Transitivity Let us assume $\tilde{d}_\infty \approx \tilde{d}'_\infty$ and $\tilde{d}'_\infty \approx \tilde{d}''_\infty$. We proceed by coinduction. We distinguish two cases:

- if $\tilde{d}'_\infty = \tau^\alpha$ for some $\alpha \in \mathbb{N} \cup \{\omega\}$, then also \tilde{d}_∞ and \tilde{d}''_∞ are made of only τ s, hence the thesis follows from the first rule;
- otherwise, we have $\tilde{d}_\infty = \tau^p \cdot [v] \cdot \tilde{s}_\infty$, $\tilde{d}'_\infty = \tau^q \cdot [v] \cdot \tilde{s}'_\infty$ and $\tilde{d}''_\infty = \tau^r \cdot [v] \cdot \tilde{s}''_\infty$, with $\tilde{s}_\infty \approx \tilde{s}'_\infty$ and $\tilde{s}'_\infty \approx \tilde{s}''_\infty$. Therefore by coinductive hypothesis we get $\tilde{s}_\infty \approx \tilde{s}''_\infty$, and the thesis follows by applying the rule

$$\frac{\tilde{s}_\infty \approx \tilde{s}''_\infty}{\tilde{d}_\infty = \tau^p \cdot [v] \cdot \tilde{s}_\infty \approx \tau^r \cdot [v] \cdot \tilde{s}''_\infty = \tilde{d}''_\infty}$$

Proof of Prop. 17 (\approx and R_\approx coincide) For the implication \Leftarrow we proceed by coinduction, considering two cases:

- if $\varepsilon_\tau(\tilde{d}_\infty) = \varepsilon_\tau(\tilde{d}'_\infty) = []$, then $\tilde{d}_\infty = \tau^\alpha$ and $\tilde{d}'_\infty = \tau^\beta$, hence we get the thesis by the first rule;
- otherwise we have $\tilde{d}_\infty = \tau^n \cdot [v] \cdot \tilde{s}_\infty$, $\tilde{d}'_\infty = \tau^m \cdot [v] \cdot \tilde{s}'_\infty$ and $\varepsilon_\tau(\tilde{s}_\infty) = \varepsilon_\tau(\tilde{s}'_\infty)$. Hence by coinductive hypothesis we get $\tilde{s}_\infty \approx \tilde{s}'_\infty$ and then we get the thesis by the second rule.

To show the other implication, we have to prove that the set $E = \{(\varepsilon_\tau(\tilde{d}_\infty), \varepsilon_\tau(\tilde{d}'_\infty)) \mid \tilde{d}_\infty \approx \tilde{d}'_\infty\}$ is included in the diagonal relation on \mathcal{V}^∞ . To this aim, thanks to the (coalgebraic) coinduction principle, it is enough to show that E is a bisimulation, that is, that E is an F -coalgebra. Consider the following function defined on E :

$$\begin{cases} \gamma_E([], []) & = \star \\ \gamma_E([v] \cdot \varepsilon_\tau(\tilde{s}_\infty), [v] \cdot \varepsilon_\tau(\tilde{s}'_\infty)) & = (v, (\varepsilon_\tau(\tilde{s}_\infty), \varepsilon_\tau(\tilde{s}'_\infty))) \end{cases}$$

and note that by definition of \approx we have $\tilde{s}_\infty \approx \tilde{s}'_\infty$, hence $(\varepsilon_\tau(\tilde{s}_\infty), \varepsilon_\tau(\tilde{s}'_\infty)) \in E$, and so $\gamma_E: E \rightarrow FE$ is well-defined, and (E, γ_E) is an F -coalgebra, and this concludes the proof.

Proof of Lemma 12 We relax the hypothesis, only requiring that $e \Rightarrow [v] \cdot r$ is derivable by a finite proof tree using also corules. Indeed, by definition, if $e \Rightarrow [v] \cdot r$ is derivable, then it has a finite proof tree using also corules. We proceed by induction on the definition.

(co-empty) Empty case.

(co-out) We know that $\text{out } e \Rightarrow [v] \cdot o_\infty$ and $e \Rightarrow (v', o)$. We have two cases:

- if $o = [v] \cdot o'$, then, by inductive hypothesis, $e \xrightarrow{\tilde{o}}_\star \mathcal{E}[\text{out } v]$, hence $\text{out } e \xrightarrow{\tilde{o}}_\star \text{out } \mathcal{E}[\text{out } v]$;
- if $o = []$, then $v' = v$ and by Lemma 10 we get that $e \rightsquigarrow (v, \tilde{o})$, hence $e \xrightarrow{\tilde{o}}_\star v$; hence we get that $\text{out } e \xrightarrow{\tilde{o}}_\star \text{out } v$ as needed.

(val) Empty case.

(out) Analogous to case (co-out).

(app) We know that $e_1 \ e_2 \Rightarrow [v] \cdot r$, $e_1 \Rightarrow (\lambda x.e, o_1)$, $e_2 \Rightarrow (v_2, o_2)$ and $e[x \leftarrow v_2] \Rightarrow r'$. We distinguish three cases:

- if $o_1 = [v] \cdot o'_1$, then by inductive hypothesis we get that $e_1 \xrightarrow{\tilde{o}}_{\star} \mathcal{E}[\text{out } v]$, hence $e_1 \ e_2 \xrightarrow{\tilde{o}}_{\star} \mathcal{E}[\text{out } v]$ as needed;
- if $o_1 = []$ and $o_2 = [v] \cdot o'_2$, then by Lemma 10 $e_1 \rightsquigarrow (\lambda x.e, \tilde{o}_1)$, hence $e_1 \xrightarrow{\tilde{o}_1}_{\star} \lambda x.e$, and by inductive hypothesis $e_2 \xrightarrow{\tilde{o}}_{\star} \mathcal{E}[\text{out } v]$, therefore we get that $e_1 \ e_2 \xrightarrow{\tilde{o}_1}_{\star} \lambda x.e \ e_2 \xrightarrow{\tilde{o}}_{\star} \lambda x.e \ \mathcal{E}[\text{out } v]$ as needed;
- if $o_1 = o_2 = []$, then $r' = [v] \cdot r$, hence by Lemma 10 we get that $e_1 \rightsquigarrow (\lambda x.e, \tilde{o}_1)$ and $e_2 \rightsquigarrow (v_2, \tilde{o}_2)$, hence $e_1 \xrightarrow{\tilde{o}_1}_{\star} \lambda x.e$ and $e_2 \xrightarrow{\tilde{o}_2}_{\star} v_2$, and by inductive hypothesis $e[x \leftarrow v_2] \xrightarrow{\tilde{o}}_{\star} \mathcal{E}[\text{out } v]$; therefore we get that $e_1 \ e_2 \xrightarrow{\tilde{o}_1}_{\star} \lambda x.e \ e_2 \xrightarrow{\tilde{o}_2}_{\star} \lambda x.e \ v_2 \xrightarrow{\tau} e[x \leftarrow v_2] \xrightarrow{\tilde{o}}_{\star} \mathcal{E}[\text{out } v]$ as needed.

(div) Analogous to case (app).

Proof of Theorem 11 We prove separately the three points.

1. We start by constructing inductively the sequences $(e_n)_{n \in \mathbb{N}}$ and $(r_n)_{n \in \mathbb{N}}$. The case $n = 0$ is given by the hypothesis. If we assume e_n and r_n to satisfy $e_n \Rightarrow r_n$, then we have two cases:

- if $e_n = v$, then we set $e_{n+1} = v$ and $r_{n+1} = (v, [])$;
- otherwise, by Lemma 8 there are e' and ℓ such that $e_n \xrightarrow{\ell} e'$, and by Lemma 9 there is r' such that $e' \Rightarrow r'$ and $r_n \approx [\ell] \cdot r'$; therefore we set $e_{n+1} = e'$ and $r_{n+1} = r'$.

In this way, by construction $(e_n)_{n \in \mathbb{N}}$ and $(r_n)_{n \in \mathbb{N}}$ satisfy the requirements.

In order to construct the sequence $(\tilde{r}_n)_{n \in \mathbb{N}}$ we need some more effort. First of all, we have to find an infinite sequence of streams satisfying the requirements of point 1, that is, a function $s: \mathbb{N} \rightarrow \mathcal{V}_\tau^\infty$ such that, for all $n \in \mathbb{N}$, if e_n is a value, then $s(n) = []$, and if $e_n \xrightarrow{\ell} e_{n+1}$, then $s(n) = [\ell] \cdot s(n+1)$. To do this, we give a coalgebra structure on \mathbb{N} for the functor $FX = \mathbf{1} + \mathcal{V}_\tau \times X$, given by the map $\gamma: \mathbb{N} \rightarrow F\mathbb{N}$ defined by

$$\gamma(n) = \begin{cases} \star & e_n = v \\ (\ell, n+1) & e_n \xrightarrow{\ell} e_{n+1} \end{cases}$$

Now, since \mathcal{V}_τ^∞ carries a terminal F -coalgebra, and requirements of point 1 impose that s is a homomorphism, it is uniquely determined. Therefore we set $\tilde{r}_n = (v, s(n))$ if there is $k \geq n$ such that $e_k = v$, otherwise $\tilde{r}_n = (\infty, s(n))$.

2. We proceed by coinduction, distinguishing two cases:

- if e_n is a value v , then, by point 1, $\tilde{r}_n = (v, [])$, hence $e_n \rightsquigarrow \tilde{r}_n$ holds by the first axiom;
- otherwise, by point 1, $e_n \xrightarrow{\ell} e_{n+1}$, $e_{n+1} \Rightarrow r_{n+1}$ and $\tilde{r}_n = [\ell] \cdot \tilde{r}_{n+1}$, therefore by coinductive hypothesis we get $e_{n+1} \rightsquigarrow \tilde{r}_{n+1}$ and so by the second rule we get the thesis.

3. We distinguish two cases:

- if $v_\infty = v$, then, since for all $n \in \mathbb{N}$ by points 1 and 2 we have $e_n \Rightarrow (v, o_n)$ and $e_n \rightsquigarrow \tilde{r}_n$, by Lemma 10 we get the thesis;
- consider a result r_n in the sequence, and assume $v_\infty = \infty$, then for all $k \geq n$ we have $e_k \neq v$, since otherwise we would get $v_\infty = v$, because $r_n \approx o \cdot r_k$, by point 1; therefore by construction (point 1) we get $\tilde{r}_n = (\infty, \tilde{o}_\infty^n)$.

$$\begin{array}{c}
 \frac{\text{fields}(c') = \bar{f}}{\text{fields}(\text{Object}) = \epsilon} \quad \frac{\text{class } c \text{ extends } c' \{ \bar{g}; \overline{md} \} \quad \bar{f} \cap \bar{g} = \emptyset}{\text{fields}(c) = \bar{f}, \bar{g}} \\
 \\
 \frac{\text{class } c \text{ extends } c' \{ \overline{fd} \overline{md} \ m(\bar{x}) \{ e \} \overline{md}' \}}{\text{meth}(c, m) = \bar{x}.e} \quad \frac{\text{class } c \text{ extends } c' \{ \overline{fd} \overline{md} \} \quad m \text{ not declared in } \overline{md}}{\text{meth}(c, m) = \bar{x}.e} \\
 \\
 \text{restore}(\Pi, r) = \begin{cases} r & \text{if } r = (\infty, \theta) \\ (v, \theta); \Pi; \mathcal{H} & \text{if } r = (v, \theta); \Pi'; \mathcal{H} \end{cases} \quad \theta \cdot ((v, \theta'); \Pi; \mathcal{H}) = (v, \theta \cdot \theta'); \Pi; \mathcal{H} \\
 \theta \cdot (\infty, \theta') = (\infty, \theta \cdot \theta') \\
 \\
 \frac{}{\text{ser}(\mathcal{H}, v) = v} \quad v \in \{\mathbf{false}, \mathbf{true}\} \quad \frac{\forall i = 1..n. \text{ser}(\mathcal{H}, v_i) = \bar{v}_i}{\text{ser}(\mathcal{H}, \iota) = \text{obj}(c, \bar{f}^n \mapsto \bar{v}^n)} \quad \mathcal{H}(\iota) = \text{obj}(c, \bar{f}^n \mapsto \bar{v}^n) \\
 \\
 \text{ser}(\mathcal{H}, v) = v \\
 \\
 \frac{\text{deser}(\mathcal{H}, v, \emptyset) = (\mathcal{H}', v)}{\text{deser}(\mathcal{H}, v) = (\mathcal{H}', v)} \quad \frac{}{\text{deser}(\mathcal{H}, v, M) = (\mathcal{H}, v)} \quad v \in \{\mathbf{false}, \mathbf{true}\} \\
 \\
 \frac{}{\text{deser}(\mathcal{H}, v, M \uplus \{v \mapsto \iota\}) = (\mathcal{H}, \iota)} \\
 \\
 \frac{\forall i = 1..n. \text{deser}(\mathcal{H}_{i-1}, v_i, M \cup \{v \mapsto \iota\}) = (\mathcal{H}_i, v_i)}{\text{deser}(\mathcal{H}_0, v, M) = (\mathcal{H}_n \cup \{\iota \mapsto \text{obj}(c, \bar{f}^n \mapsto \bar{v}^n)\}, \iota)} \quad \begin{array}{l} v \notin \text{dom}(M) \\ v = \text{obj}(c, \bar{f}^n \mapsto \bar{v}^n) \\ \iota \notin \text{dom}(\mathcal{H}_0) \cup \text{img}(M) \end{array}
 \end{array}$$

■ **Figure 12** Definition of auxiliary functions and operators

C Auxiliary definitions and additional examples for the imperative Java-like language

Auxiliary definitions. Fig. 12 shows the definition of all auxiliary functions and operators used in the rules.

Functions *fields* and *meth* are standard. Function *restore* replaces the stack frame of the callee with that of the caller after the computation returns from a method invocation (rule (INV)).

The operation $\theta \cdot r$ updates the result r by appending the finite trace θ to the left of the possibly infinite trace θ' of r .

Function *ser* corresponds to built-in serialization of an internal value v into a corresponding serialized value v ; because object values are heap references, the function depends also on the heap. Serialization of primitive values is trivial, and serialization of objects does not yield a sequence (as happens in practice), but rather preserves the tree structure of the original object, allowed to be infinite, but still regular because heaps have finite domains. For this reason, we provide a standard coinductive definition of *ser*.

The inverse function *deser* depends on heaps as well, since deserialization of objects requires object allocation; for the same reason, the function returns a pair consisting of a new heap and a value. Deserialization of cyclic objects requires particular care, because one has to avoid infinite unfolding which would lead to a heap with an infinite domain; our choice is to

minimize unfolding, therefore recursive deserialization stops as soon as a loop is detected in the structure of the serialized object, and no redundant cycles are introduced in the heap. To this aim, the definition of *deser* is based on an overloaded function *deser* taking a third argument M , which is a finite map from serialized objects to their associated reference, to keep track of cycles. If a serialized object v is already in the domain of M , then a cycle has been detected, and the associated reference in M and the unchanged heap are returned. Otherwise, deserialization is propagated to the fields of v with an updated map M where a fresh reference ι is associated with v . Finally, an updated heap and the new reference ι are returned. Because of the use of the third argument M , the definition of $\text{deser}(\mathcal{H}, v, M)$ is inductive.

Example 2. This example is a simple variation of Example 1, where method m does not output the input value.

```
class C extends Object{m(x){this.m(in)}}
```

In this case the only derivable judgments have shape $\emptyset; \emptyset; e \Rightarrow (\infty, \theta_0)$, where the event trace θ_0 is defined by $\theta_0 = [\text{in } v_0 \text{ in } v_1 \dots]$. The corresponding infinite proof is obtained from that of Example 1 by slightly changing the trees ∇_i (with the same simplifying assumption that only primitive input values are considered); in this case we have

$$\nabla_i = \frac{\frac{(\text{VAR}) \overline{\Pi_i; \mathcal{H}; \text{this} \Rightarrow (\iota, [\]); \Pi_i; \mathcal{H}}}{\Pi_i; \mathcal{H}; \text{this.m(in)} \Rightarrow (\infty, [\text{in } v_i] \cdot \theta_{i+1})} \quad (\text{INV}) \quad \frac{(\text{IN}) \overline{\Pi_i; \mathcal{H}; \text{in} \Rightarrow (v_i, [\text{in } v_i]); \Pi_i; \mathcal{H}}}{\nabla_{i+1}}}{\nabla_i}$$

where for all $i \in \mathbb{N}$, $\theta_i = [\text{in } v_i] \cdot \theta_{i+1}$, whereas \mathcal{H} , and Π_i are defined as before. The following finite trees built with corules show that the infinite trees ∇_i are valid:

$$\frac{(\text{VAR}) \overline{\Pi_i; \mathcal{H}; \text{this} \Rightarrow (\iota, [\]); \Pi_i; \mathcal{H}} \quad (\text{CO-IN}) \overline{\Pi_i; \mathcal{H}; \text{in} \Rightarrow (\infty, [\text{in } v_i] \cdot \theta_{i+1}); \Pi_i; \mathcal{H}}}{(\text{DIV}) \overline{\Pi_i; \mathcal{H}; \text{this.m(in)} \Rightarrow (\infty, [\text{in } v_i] \cdot \theta_{i+1})}}$$

Example 3. This example is a more elaborated variation of Example 1, where some computation is performed on input values before they are output, and a simple cache object is employed. To simplify the code we employ the primitive operator `==`, together with integer literals and addition.

```
class H extends Object{ // simple cache objects
  // fields store the last input and its associated output
  input; output;
  get(i){
    if(i==this.input) this.output
    else this.output=this.calc(this.input=i)
  }
  /* performs some computation on i and returns the result */
  calc(i){ i+1 }
}
class C extends Object{cache; m(x){this.m(out cache.get(in))}}
```

If we consider the main expression $e = \text{new } C(\text{new } H(0,0)).m(\text{true})$, and restrict the observation to positive integers as input, then the only derivable judgments have shape $\emptyset; \emptyset; e \Rightarrow (\infty, \theta_0)$, where $\theta_0 = [\text{in } v_0 \text{ out } v_0 + 1 \text{ in } v_1 \text{ out } v_1 + 1 \dots]$, with v_i a positive integer for all $i \in \mathbb{N}$.

The Essence of Nested Composition

Xuan Bi¹

The University of Hong Kong, Hong Kong, China
xbi@cs.hku.hk

Bruno C. d. S. Oliveira¹

The University of Hong Kong, Hong Kong, China
bruno@cs.hku.hk

Tom Schrijvers²

KU Leuven, Belgium
tom.schrijvers@cs.kuleuven.be

Abstract

Calculi with *disjoint intersection types* support an introduction form for intersections called the *merge operator*, while retaining a *coherent* semantics. Disjoint intersection types have great potential to serve as a foundation for powerful, flexible and yet type-safe and easy to reason OO languages. This paper shows how to significantly increase the expressive power of disjoint intersection types by adding support for *nested subtyping and composition*, which enables simple forms of *family polymorphism* to be expressed in the calculus. The extension with nested subtyping and composition is challenging, for two different reasons. Firstly, the subtyping relation that supports these features is non-trivial, especially when it comes to obtaining an algorithmic version. Secondly, the syntactic method used to prove coherence for previous calculi with disjoint intersection types is too inflexible, making it hard to extend those calculi with new features (such as nested subtyping). We show how to address the first problem by adapting and extending the Barendregt, Coppo and Dezani (BCD) subtyping rules for intersections with records and coercions. A sound and complete algorithmic system is obtained by using an approach inspired by Pierce's work. To address the second problem we replace the syntactic method to prove coherence, by a semantic proof method based on *logical relations*. Our work has been fully formalized in Coq, and we have an implementation of our calculus.

2012 ACM Subject Classification Software and its engineering → Object oriented languages

Keywords and phrases nested composition, family polymorphism, intersection types, coherence

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2018.22

Supplement Material ECOOP Artifact Evaluation approved artifact available at <http://dx.doi.org/10.4230/DARTS.4.3.5>

Acknowledgements We thank the anonymous reviewers for their helpful comments.

1 Introduction

Intersection types [49, 18] have a long history in programming languages. They were originally introduced to characterize exactly all strongly normalizing lambda terms. Since then, starting with Reynolds's work on Forsythe [54], they have also been employed to express

¹ Funded by Hong Kong Research Grant Council projects number 17210617 and 17258816

² Funded by The Research Foundation - Flanders



useful programming language constructs, such as key aspects of *multiple inheritance* [17] in Object-Oriented Programming (OOP). One notable example is the Scala language [44] and its DOT calculus [3], which make fundamental use of intersection types to express a class/trait that extends multiple other traits. Other modern languages, such as TypeScript [39], Flow [28] and Ceylon [51], also adopt some form of intersection types.

Intersection types come in different varieties in the literature. Some calculi provide an *explicit* introduction form for intersections, called the *merge operator*. This operator was introduced by Reynolds in Forsythe [54] and adopted by a few other calculi [15, 23, 46, 2]. Unfortunately, while the merge operator is powerful, it also makes it hard to get a *coherent* (or unambiguous) semantics. Unrestricted uses of the merge operator can be ambiguous, leading to an incoherent semantics where the same program can evaluate to different values. A far more common form of intersection types are the so-called *refinement types* [30, 21, 24]. Refinement types restrict the formation of intersection types so that the two types in an intersection are refinements of the same simple (unrefined) type. Refinement intersection increases only the expressiveness of types and not of terms. For this reason, Dunfield [23] argues that refinement intersection is unsuited for encoding various useful language features that require the merge operator (or an equivalent term-level operator).

Disjoint Intersection Types. λ_i is a recent calculus with a variant of intersection types called *disjoint intersection types* [46]. Calculi with disjoint intersection types feature the merge operator, with restrictions that all expressions in a merge operator must have disjoint types and all well-formed intersections are also disjoint. A bidirectional type system and the disjointness restrictions ensure that the semantics of the resulting calculi remains coherent.

Disjoint intersection types have great potential to serve as a foundation for powerful, flexible and yet type-safe OO languages that are easy to reason about. As shown by Alpuim et al. [2], calculi with disjoint intersection types are very expressive and can be used to statically type-check JavaScript-style programs using mixins. Yet they retain both type safety and coherence. While coherence may seem at first of mostly theoretical relevance, it turns out to be very relevant for OOP. Multiple inheritance is renowned for being tricky to get right, largely because of the possible *ambiguity* issues caused by the same field/method names inherited from different parents [9, 58]. Disjoint intersection types enforce that the types of parents are disjoint and thus that no conflicts exist. Any violations are statically detected and can be manually resolved by the programmer. This is very similar to existing trait models [29, 22]. Therefore in an OO language modelled on top of disjoint intersection types, coherence implies that no ambiguity arises from multiple inheritance. This makes reasoning a lot simpler.

Family Polymorphism. One powerful and long-standing idea in OOP is *family polymorphism* [25]. In family polymorphism inheritance is extended to work on a *whole family of classes*, rather than just a single class. This enables high degrees of modularity and reuse, including simple solutions to hard programming language problems, like the Expression Problem [64]. An essential feature of family polymorphism is *nested composition* [19, 27, 42], which allows the automatic inheritance/composition of nested (or inner) classes when the top-level classes containing them are composed. Designing a sound type system that fully supports family polymorphism and nested composition is notoriously hard; there are only a few, quite sophisticated, languages that manage this [27, 42, 16, 57].

NeColus. This paper presents an improved variant of λ_i called **NeColus**³ (or λ_i^+): a simple calculus with records and disjoint intersection types that supports *nested composition*. Nested composition enables encoding simple forms of family polymorphism. More complex forms of family polymorphism, involving binary methods [11] and mutable state are not yet supported, but are interesting avenues for future work. Nevertheless, in **NeColus**, it is possible, for example, to encode Ernst’s elegant family-polymorphism solution [25] to the Expression Problem. Compared to λ_i the essential novelty of **NeColus** are distributivity rules between function/record types and intersection types. These rules are the delta that enable extending the simple forms of multiple inheritance/composition supported by λ_i into a more powerful form supporting nested composition. The distributivity rule between function types and intersections is common in calculi with intersection types aimed at capturing the set of all strongly normalizable terms, and was first proposed by Barendregt et al. [4] (BCD). However the distributivity rule is not common in calculi or languages with intersection types aimed at programming. For example the rules employed in languages that support intersection types (such as Scala, TypeScript, Flow or Ceylon) lack distributivity rules. Moreover distributivity is also missing from several calculi with a merge operator. This includes all calculi with disjoint intersection types and Dunfield’s work on elaborating intersection types, which was the original foundation for λ_i . A possible reason for this omission in the past is that distributivity adds substantial complexity (both algorithmically and metatheoretically), without having any obvious practical applications. This paper shows how to deal with the complications of BCD subtyping, while identifying a major reason to include it in a programming language: BCD enables nested composition and subtyping, which is of significant practical interest.

NeColus differs significantly from previous BCD-based calculi in that it has to deal with the possibility of incoherence, introduced by the merge operator. Incoherence is a non-issue in the previous BCD-based calculi because they do not feature this merge operator or any other source of incoherence. Although previous work on disjoint intersection types proposes a solution to coherence, the solution imposes several ad-hoc restrictions to guarantee the uniqueness of the elaboration and thus allow for a simple syntactic proof of coherence. Most importantly, it makes it hard or impossible to adapt the proof to extensions of the calculus, such as the new subtyping rules required by the BCD system.

In this work we remove the brittleness of the previous syntactic method to prove coherence, by employing a more semantic proof method based on *logical relations* [63, 48, 61]. This new proof method has several advantages. Firstly, with the new proof method, several restrictions that were enforced by λ_i to enable the syntactic proof method are removed. For example the work on λ_i has to carefully distinguish between so-called *top-like types* and other types. In **NeColus** this distinction is not necessary; top-like types are handled like all other types. Secondly, the method based on logical relations is more powerful because it is based on semantic rather than syntactic equality. Finally, the removal of the ad-hoc side-conditions makes adding new extensions, such as support for BCD-style subtyping, easier. In order to deal with the complexity of the elaboration semantics of **NeColus**, we employ binary logical relations that are heterogeneous, parameterized by two types; the fundamental property is also reformulated to account for bidirectional type-checking.

In summary the contributions of this paper are:

- **NeColus**: a calculus with (disjoint) intersection types that features both *BCD-style subtyping* and *the merge operator*. This calculus is both type-safe and coherent, and supports *nested composition*.

³ **NeColus** stands for **Nested Composition calculus**.

- A more flexible notion of disjoint intersection types where only merges need to be checked for disjointness. This removes the need for enforcing disjointness for all well-formed types, making the calculus more easily extensible.
- An extension of BCD subtyping with both records and elaboration into coercions, and algorithmic subtyping rules with coercions, inspired by Pierce’s decision procedure [47].
- A more powerful proof strategy for coherence of disjoint intersection types based on logical relations.
- Illustrations of how the calculus can encode essential features of *family polymorphism* through nested composition.
- A comprehensive Coq mechanization of all meta-theory. This has notably revealed several missing lemmas and oversights in Pierce’s manual proof [47] of BCD’s algorithmic subtyping. We also have an implementation of a language built on top of NeColus; it runs and type-checks all examples shown in the paper.⁴

2 Overview

This section illustrates NeColus with an encoding of a family polymorphism solution to the Expression Problem, and informally presents its salient features.

2.1 Motivation: Family Polymorphism

In OOP *family polymorphism* is the ability to simultaneously refine a family of related classes through inheritance. This is motivated by a need to not only refine individual classes, but also to preserve and refine their mutual relationships. Nystrom et al. [42] call this *scalable extensibility*: “the ability to extend a body of code while writing new code proportional to the differences in functionality”. A well-studied mechanism to achieve family inheritance is *nested inheritance* [42]. Nested inheritance combines two aspects. Firstly, a class can have nested class members; the outer class is then a family of (inner) classes. Secondly, when one family extends another, it inherits (and can override) all the class members, as well as the relationships within the family (including subtyping) between the class members. However, the members of the new family do not become subtypes of those in the parent family.

The Expression Problem, Scandinavian Style. Ernst [25] illustrates the benefits of nested inheritance for modularity and extensibility with one of the most elegant and concise solutions to the *Expression Problem* [64]. The objective of the Expression Problem is to extend a datatype, consisting of several cases, together with several associated operations in two dimensions: by adding more cases to the datatype and by adding new operations for the datatype. Ernst solves the Expression Problem in the gbeta language, which he adorns with a Java-like syntax for presentation purposes, for a small abstract syntax tree (AST) example. His starting point is the code shown in Fig. 1a. The outer class `Lang` contains a family of related AST classes: the common superclass `Exp` and two cases, `Lit` for literals and `Add` for addition. The AST comes equipped with one operation, `toString`, which is implemented by both cases.

⁴ The Coq formalization and implementation are available at <https://goo.gl/R5hUAp>.

```

class Lang {
  virtual class Exp {
    String toString() {}
  }
  virtual class Lit extends Exp {
    int value;
    Lit(int value) {
      this.value = value;
    }
    String toString() {
      return value;
    }
  }
  virtual class Add extends Exp {
    Exp left, right;
    Add(Exp left, Exp right) {
      this.left = left;
      this.right = right;
    }
    String toString() {
      return left + "+" + right;
    }
  }
}

// Adding a new operation
class LangEval extends Lang {
  refine class Exp {
    int eval() {}
  }
  refine class Lit {
    int eval { return value; }
  }
  refine class Add {
    int eval { return
      left.eval() + right.eval();
    }
  }
}

// Adding a new case
class LangNeg extends Lang {
  virtual class Neg extends Exp {
    Neg(Exp exp) { this.exp = exp; }
    String toString() {
      return "-" + exp + ";
    }
    Exp exp;
  }
}

```

(a) Base family: the language `Lang`.

(b) Extending in two dimensions.

■ **Figure 1** The Expression Problem, Scandinavian Style.

Adding a New Operation. One way to extend the family is to add an additional evaluation operation, as shown in the top half of Fig. 1b. This is done by subclassing the `Lang` class and refining all the contained classes by implementing the additional `eval` method. Note that the inheritance between, e.g., `Lang.Exp` and `Lang.Lit` is transferred to `LangEval.Exp` and `LangEval.Lit`. Similarly, the `Lang.Exp` type of the `left` and `right` fields in `Lang.Add` is automatically refined to `LangEval.Exp` in `LangEval.Add`.

Adding a New Case. A second dimension to extend the family is to add a case for negation, shown in the bottom half of Fig. 1b. This is similarly achieved by subclassing `Lang`, and now adding a new contained class `Neg`, for negation, that implements the `toString` operation.

Finally, the two extensions are naturally combined by means of multiple inheritance, closing the diamond.

```

class LangNegEval extends LangEval & LangNeg {
  refine class Neg {
    int eval() { return -exp.eval(); }
  }
}

```

The only effort required is to implement the one missing operation case, evaluation of negated expressions.

2.2 The Expression Problem, NeColus Style

The NeColus calculus allows us to solve the Expression Problem in a way that is very similar to Ernst's gbeta solution. However, the underlying mechanisms of NeColus are quite different

22:6 The Essence of Nested Composition

from those of `gbeta`. In particular, `NeColus` features a structural type system in which we can model objects with records, and object types with record types. For instance, we model the interface of `Lang.Exp` with the singleton record type `{ print : String }`. For the sake of conciseness, we use type aliases to abbreviate types.

```
type IPrint = { print : String };
```

Similarly, we capture the interface of the `Lang` family in a record, with one field for each case's constructor.

```
type Lang = { lit : Int → IPrint, add : IPrint → IPrint → IPrint };
```

Here is the implementation of `Lang`.

```
implLang : Lang = {
  lit (value : Int) = { print = value.toString },
  add (left : IPrint) (right : IPrint) = {
    print = left.print ++ "+" ++ right.print
  }
};
```

Adding Evaluation. We obtain `IPrint & IEval`, which is the corresponding type for `LangEval.Exp`, by intersecting `IPrint` with `IEval` where

```
type IEval = { eval : Int };
```

The type for `LangEval` is then:

```
type LangEval = {
  lit : Int → IPrint & IEval,
  add : IPrint & IEval → IPrint & IEval → IPrint & IEval
};
```

We obtain an implementation for `LangEval` by merging the existing `Lang` implementation `implLang` with the new evaluation functionality `implEval` using the merge operator `.,.`

```
implEval = {
  lit (value : Int) = { eval = value },
  add (left : IEval) (right : IEval) = {
    eval = left.eval + right.eval
  }
};
implLangEval : LangEval = implLang ,. implEval;
```

Adding Negation. Adding negation to `Lang` works similarly.

```
type NegPrint = { neg : IPrint → IPrint };
type LangNeg = Lang & NegPrint;

implNegPrint : NegPrint = {
  neg (exp : IPrint) = { print = "-" ++ exp.print }
};
implLangNeg : LangNeg = implLang ,. implNegPrint;
```

Putting Everything Together. Finally, we can combine the two extensions and provide the missing implementation of evaluation for the negation case.

```

type NegEval = { neg : IEval → IEval };
implNegEval : NegEval = {
  neg (exp : IEval) = { eval = 0 - exp.eval }
};

type NegEvalExt = { neg : IPrint & IEval → IPrint & IEval };
type LangNegEval = LangEval & NegEvalExt;
implLangNegEval : LangNegEval = implLangEval ,, implNegPrint ,, implNegEval;

```

We can test `implLangNegEval` by creating an object `e` of expression $-2 + 3$ that is able to print and evaluate at the same time.

```

fac = implLangNegEval;
e = fac.add (fac.neg (fac.lit 2)) (fac.lit 3);
main = e.print ++ " = " ++ e.eval.toString -- Output: "-2+3 = 1"

```

Multi-Field Records. One relevant remark is that NeColus does not have multi-field record types built in. They are merely syntactic sugar for intersections of single-field record types. Hence, the following is an equivalent definition of `Lang`:

```

type Lang = {lit : Int → IPrint} & {add : IPrint → IPrint → IPrint};

```

Similarly, the multi-field record expression in the definition of `implLang` is syntactic sugar for the explicit merge of two single-field records.

```

implLang : Lang = { lit = ... } ,, { add = ... };

```

Subtyping. A big difference compared to `gbeta` is that many more NeColus types are related through subtyping. Indeed, `gbeta` is unnecessarily conservative [26]: none of the families is related through subtyping, nor is any of the class members of one family related to any of the class members in another family. For instance, `LangEval` is not a subtype of `Lang`, nor is `LangNeg.Lit` a subtype of `Lang.Lit`.

In contrast, subtyping in NeColus is much more nuanced and depends entirely on the structure of types. The primary source of subtyping are intersection types: any intersection type is a subtype of its components. For instance, `IPrint & IEval` is a subtype of both `IPrint` and `IEval`. Similarly `LangNeg = Lang & NegPrint` is a subtype of `Lang`. Compare this to `gbeta` where `LangEval.Expr` is not a subtype of `Lang.Expr`, nor is the family `LangNeg` a subtype of the family `Lang`.

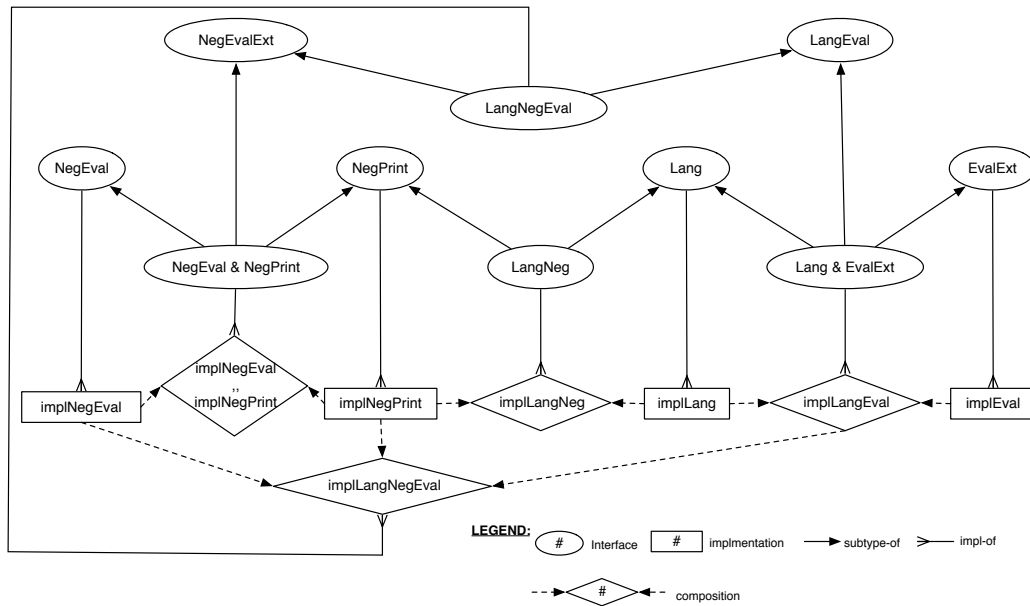
However, `gbeta` and NeColus agree that `LangEval` is not a subtype of `Lang`. The NeColus-side of this may seem contradictory at first, as we have seen that intersection types arise from the use of the merge operator, and we have created an implementation for `LangEval` with `implLang ,, implEval` where `implLang : Lang`. That suggests that `LangEval` is a subtype of `Lang`. Yet, there is a flaw in our reasoning: strictly speaking, `implLang ,, implEval` is not of type `LangEval` but instead of type `Lang & EvalExt`, where `EvalExt` is the type of `implEval`:

```

type EvalExt = { lit : Int → IEval, add : IEval → IEval → IEval };

```

Nevertheless, the definition of `implLangEval` is valid because `Lang & EvalExt` is a subtype of `LangEval`. Indeed, if we consider for the sake of simplicity only the `lit` field, we have that `(Int → IPrint) & (Int → IEval)` is a subtype of `Int → IPrint & IEval`. This follows from



■ **Figure 2** Summary diagram of the relationships between language components.

a standard subtyping axiom for distributivity of functions and intersections in the BCD system inherited by NeColus. In conclusion, `Lang & EvalExt` is a subtype of both `Lang` and of `LangEval`. However, neither of the latter two types is a subtype of the other. Indeed, `LangEval` is not a subtype of `Lang` as the type of `add` is not covariantly refined and thus admitting the subtyping is unsound. For the same reason `Lang` is not a subtype of `LangEval`.

Figure 2 shows the various relationships between the language components. Admittedly, the figure looks quite complex because our calculus features a structural type system (as often more foundational calculi do), whereas mainstream OO languages have nominal type systems. This is part of the reason why we have so many subtyping relations in Fig. 2.

Stand-Alone Extensions. Unlike in `gbeta` and other class-based inheritance systems, in NeColus the extension `implEval` is not tied to `LangEval`. In that sense, it resembles trait and mixin systems that can apply the same extension to different classes. However, unlike those systems, `implEval` can also exist as a value on its own, i.e., it is not an extension per se.

2.3 Disjoint Intersection Types and Ambiguity

The above example shows that intersection types and the merge operator are closely related to multiple inheritance. Indeed, they share a major concern with multiple inheritance, namely ambiguity. When a subclass inherits an implementation of the same method from two different parent classes, it is unclear which of the two methods is to be adopted by the subclass. In the case where the two parent classes have a common superclass, this is known as the *diamond problem*. The ambiguity problem also appears in NeColus, e.g., if we merge two numbers to obtain `1, 2` of type `Nat & Nat`. Is the result of `1, 2 + 3` either 4 or 5?

Disjoint intersection types offer to statically detect potential ambiguity and to ask the programmer to explicitly resolve the ambiguity by rejecting the program in its ambiguous form. In the previous work on λ_i , ambiguity is avoided by dictating that all intersection

types have to be disjoint, i.e., $\text{Nat} \& \text{Nat}$ is ill-formed because the first component has the same type as the second.

Duplication is Harmless. While requiring that all intersections are disjoint is sufficient to guarantee coherence, it is not necessary. In fact, such requirement unnecessarily encumbers the subtyping definition with disjointness constraints and an ad-hoc treatment of “top-like” types. Indeed, the value $1, , 1$ of the non-disjoint type $\text{Nat} \& \text{Nat}$ is entirely unambiguous, and $(1, , 1) + 3$ can obviously only result in 4. More generally, when the overlapping components of an intersection type have the same value, there is no ambiguity problem. **NeColus** uses this idea to relax λ_i ’s enforcement of disjointness. In the case of a merge, it is hard to statically decide whether the two arguments have the same value, and thus **NeColus** still requires disjointness. This is why in Fig. 2 we cannot define `implLangNegEval` by directly composing the two existing `implLangEval` and `implLangNeg`, even though the latter two both contain the same `implLang`. Yet, disjointness is no longer required for the well-formedness of types and overlapping intersections can be created implicitly through subtyping, which results in duplicating values at runtime. For instance, while $1, , 1$ is not expressible $1 : \text{Nat} \& \text{Nat}$ creates the equivalent value implicitly. In short, duplication is harmless and subtyping only generates duplicated values for non-disjoint types.

2.4 Logical Relations for Coherence

Coherence is easy to establish for λ_i as its rigid rules mean that there is at most one possible subtyping derivation between any two types. As a consequence there is only one possible elaboration and thus one possible behavior for any program.

Two factors make establishing coherence for **NeColus** much more difficult: the relaxation of disjointness and the adoption of the more expressive subtyping rules from the BCD system (for which λ_i lacks). These two factors mean that subtyping proofs are no longer unique and hence that there are multiple elaborations of the same source program. For instance, $\text{Nat} \& \text{Nat}$ is a subtype of Nat in two ways: by projection on either the first or second component. Hence the fact that all elaborations yield the same result when evaluated has become a much more subtle property that requires sophisticated reasoning. For instance, in the example, we can see that coherence holds because at runtime any value of type $\text{Nat} \& \text{Nat}$ has identical components, and thus both projections yield the same result.

For **NeColus** in general, we show coherence by capturing the non-ambiguity invariant in a logical relation and showing that it is preserved by the operational semantics. A complicating factor is that not one, but two languages are involved: the source language **NeColus** and the target language, essentially the simply-typed lambda calculus extended with coercions and records. The logical relation does not hold for target programs and program contexts in general, but only for those that are the image of a corresponding source program or program context. Thus we must view everything through the lens of elaboration.

3 NeColus: Syntax and Semantics

In this section we formally present the syntax and semantics of **NeColus**. Compared to prior work [2, 46], **NeColus** has a more powerful subtyping relation. The new subtyping relation is inspired by BCD-style subtyping, but with two noteworthy differences: subtyping is coercive (in contrast to traditional formulations of BCD); and it is extended with records. We also have a new target language with explicit coercions inspired by the coercion calculus of Henglein [32]. A full technical comparison between λ_i^+ and λ_i can be found in Section 3.5.

Types	A, B, C	$::=$	$\text{Nat} \mid \top \mid A \rightarrow B \mid A \& B \mid \{l : A\}$
Expressions	E	$::=$	$x \mid i \mid \top \mid \lambda x. E \mid E_1 E_2 \mid E_1 , , E_2 \mid E : A \mid \{l = E\} \mid E.l$
Typing contexts	Γ	$::=$	$\bullet \mid \Gamma, x : A$

■ **Figure 3** Syntax of NeColus.

$A <: B \rightsquigarrow c$		<i>(Declarative subtyping)</i>
S-REFL	S-TRANS	S-TOP
$\frac{}{A <: A \rightsquigarrow \text{id}}$	$\frac{A_2 <: A_3 \rightsquigarrow c_1 \quad A_1 <: A_2 \rightsquigarrow c_2}{A_1 <: A_3 \rightsquigarrow c_1 \circ c_2}$	$\frac{}{A <: \top \rightsquigarrow \text{top}}$
S-RCD	S-ARR	S-ANDL
$\frac{A <: B \rightsquigarrow c}{\{l : A\} <: \{l : B\} \rightsquigarrow \{l : c\}}$	$\frac{B_1 <: A_1 \rightsquigarrow c_1 \quad A_2 <: B_2 \rightsquigarrow c_2}{A_1 \rightarrow A_2 <: B_1 \rightarrow B_2 \rightsquigarrow c_1 \rightarrow c_2}$	$\frac{}{A_1 \& A_2 <: A_1 \rightsquigarrow \pi_1}$
S-ANDR	S-AND	
$\frac{}{A_1 \& A_2 <: A_2 \rightsquigarrow \pi_2}$	$\frac{A_1 <: A_2 \rightsquigarrow c_1 \quad A_1 <: A_3 \rightsquigarrow c_2}{A_1 <: A_2 \& A_3 \rightsquigarrow \langle c_1, c_2 \rangle}$	
S-DISTARR	S-TOPARR	
$\frac{}{(A_1 \rightarrow A_2) \& (A_1 \rightarrow A_3) <: A_1 \rightarrow A_2 \& A_3 \rightsquigarrow \text{dist}_{\rightarrow}}$	$\frac{}{\top <: \top \rightarrow \top \rightsquigarrow \text{top}_{\rightarrow}}$	
S-DISTRCD	S-TOPRCD	
$\frac{}{\{l : A\} \& \{l : B\} <: \{l : A \& B\} \rightsquigarrow \text{dist}_{\{l\}}}$	$\frac{}{\top <: \{l : \top\} \rightsquigarrow \text{top}_{\{l\}}}$	

■ **Figure 4** Declarative specification of subtyping.

3.1 Syntax

Figure 3 shows the syntax of NeColus. For brevity of the meta-theoretic study, we do not consider primitive operations on natural numbers, or other primitive types. They can be easily added to the language, and our prototype implementation is indeed equipped with common primitive types and their operations. Metavariables A, B, C range over types. Types include naturals Nat , a top type \top , function types $A \rightarrow B$, intersection types $A \& B$, and singleton record types $\{l : A\}$. Metavariable E ranges over expressions. Expressions include variables x , natural numbers i , a canonical top value \top , lambda abstractions $\lambda x. E$, applications $E_1 E_2$, merges $E_1 , , E_2$, annotated terms $E : A$, singleton records $\{l = E\}$, and record selections $E.l$.

3.2 Declarative Subtyping

Figure 4 presents the subtyping relation. We ignore the **highlighted** parts, and explain them later in Section 3.4.

BCD-Style Subtyping. The subtyping rules are essentially those of the BCD type system [4], extended with subtyping for singleton records. Rules S-TOP and S-RCD for top types and record types are straightforward. Rule S-ARR for function subtyping is standard. Rules S-ANDL, S-ANDR, and S-AND for intersection types axiomatize that $A \& B$ is the greatest lower bound of A and B . Rule S-DISTARR is perhaps the most interesting rule. This, so-called “distributivity” rule, describes the interaction between the subtyping relations for

$$\boxed{\Gamma \vdash E \Rightarrow A \rightsquigarrow e} \quad (\text{Inference})$$

$$\begin{array}{c}
\text{T-TOP} \\
\hline
\Gamma \vdash \top \Rightarrow \top \rightsquigarrow \langle \rangle
\end{array}
\quad
\begin{array}{c}
\text{T-LIT} \\
\hline
\Gamma \vdash i \Rightarrow \text{Nat} \rightsquigarrow i
\end{array}
\quad
\begin{array}{c}
\text{T-VAR} \\
x : A \in \Gamma \\
\hline
\Gamma \vdash x \Rightarrow A \rightsquigarrow x
\end{array}
\quad
\begin{array}{c}
\text{T-APP} \\
\Gamma \vdash E_1 \Rightarrow A_1 \rightarrow A_2 \rightsquigarrow e_1 \\
\Gamma \vdash E_2 \Leftarrow A_1 \rightsquigarrow e_2 \\
\hline
\Gamma \vdash E_1 E_2 \Rightarrow A_2 \rightsquigarrow e_1 e_2
\end{array}$$

$$\begin{array}{c}
\text{T-ANNO} \\
\Gamma \vdash E \Leftarrow A \rightsquigarrow e \\
\hline
\Gamma \vdash E : A \Rightarrow A \rightsquigarrow e
\end{array}
\quad
\begin{array}{c}
\text{T-PROJ} \\
\Gamma \vdash E \Rightarrow \{l : A\} \rightsquigarrow e \\
\hline
\Gamma \vdash E.l \Rightarrow A \rightsquigarrow e.l
\end{array}
\quad
\begin{array}{c}
\text{T-MERGE} \\
\Gamma \vdash E_1 \Rightarrow A_1 \rightsquigarrow e_1 \\
\Gamma \vdash E_2 \Rightarrow A_2 \rightsquigarrow e_2 \quad A_1 * A_2 \\
\hline
\Gamma \vdash E_1 \text{ , } E_2 \Rightarrow A_1 \& A_2 \rightsquigarrow \langle e_1, e_2 \rangle
\end{array}$$

$$\begin{array}{c}
\text{T-RCD} \\
\Gamma \vdash E \Rightarrow A \rightsquigarrow e \\
\hline
\Gamma \vdash \{l = E\} \Rightarrow \{l : A\} \rightsquigarrow \{l = e\}
\end{array}$$

$$\boxed{\Gamma \vdash E \Leftarrow A \rightsquigarrow e} \quad (\text{Checking})$$

$$\begin{array}{c}
\text{T-ABS} \\
\Gamma, x : A \vdash E \Leftarrow B \rightsquigarrow e \\
\hline
\Gamma \vdash \lambda x. E \Leftarrow A \rightarrow B \rightsquigarrow \lambda x. e
\end{array}
\quad
\begin{array}{c}
\text{T-SUB} \\
\Gamma \vdash E \Rightarrow B \rightsquigarrow e \quad B <: A \rightsquigarrow c \\
\hline
\Gamma \vdash E \Leftarrow A \rightsquigarrow ce
\end{array}$$

■ **Figure 5** Bidirectional type system of NeColus.

function types and those for intersection types. It can be shown⁵ that the other direction $(A_1 \rightarrow A_2 \& A_3 <: (A_1 \rightarrow A_2) \& (A_1 \rightarrow A_3))$ and the contravariant distribution $((A_1 \rightarrow A_2) \& (A_3 \rightarrow A_2) <: A_1 \& A_3 \rightarrow A_2)$ are both derivable from the existing subtyping rules. Rule S-DISTRCD, which is not found in the original BCD system, prescribes the distribution of records over intersection types. The two distributivity rules are the key to enable nested composition. The rule S-TOPARR is standard in BCD subtyping, and the new rule S-TOPRCD plays a similar role for record types.

Non-Algorithmic. The subtyping relation in Fig. 4 is clearly no more than a specification due to the two subtyping axioms: rules S-REFL and S-TRANS. A sound and complete algorithmic version is discussed in Section 5. Nevertheless, for the sake of establishing coherence, the declarative subtyping relation is sufficient.

3.3 Typing of NeColus

The bidirectional type system for NeColus is shown in Fig. 5. Again we ignore the highlighted parts for now.

Typing Rules and Disjointness. As with traditional bidirectional type systems, we employ two modes: the inference mode (\Rightarrow) and the checking mode (\Leftarrow). The inference judgement $\Gamma \vdash E \Rightarrow A$ says that we can synthesize a type A for expression E in the context Γ . The

⁵ The full derivations are found in the appendix.

$A * B$				<i>(Disjointness)</i>
D-TOPL $\frac{}{\top * A}$	D-TOPR $\frac{}{A * \top}$	D-ARR $\frac{A_2 * B_2}{A_1 \rightarrow A_2 * B_1 \rightarrow B_2}$	D-ANDL $\frac{A_1 * B \quad A_2 * B}{A_1 \& A_2 * B}$	D-ANDR $\frac{A * B_1 \quad A * B_2}{A * B_1 \& B_2}$
D-RCDEQ $\frac{A * B}{\{l : A\} * \{l : B\}}$	D-RCDNEQ $\frac{l_1 \neq l_2}{\{l_1 : A\} * \{l_2 : B\}}$	D-AXNATARR $\frac{}{\text{Nat} * A_1 \rightarrow A_2}$	D-AXARRNAT $\frac{}{A_1 \rightarrow A_2 * \text{Nat}}$	
D-AXNATRCD $\frac{}{\text{Nat} * \{l : A\}}$	D-AXRCDNAT $\frac{}{\{l : A\} * \text{Nat}}$	D-AXARRRCD $\frac{}{A_1 \rightarrow A_2 * \{l : A\}}$	D-AXRCDARR $\frac{}{\{l : A\} * A_1 \rightarrow A_2}$	

■ **Figure 6** Disjointness.

Target types	τ	::=	$\text{Nat} \mid \langle \rangle \mid \tau_1 \times \tau_2 \mid \tau_1 \rightarrow \tau_2 \mid \{l : \tau\}$
Typing contexts	Δ	::=	$\bullet \mid \Delta, x : \tau$
Target terms	e	::=	$x \mid i \mid \langle \rangle \mid \lambda x. e \mid e_1 e_2 \mid \langle e_1, e_2 \rangle \mid \{l = e\} \mid e.l \mid c e$
Coercions	c	::=	$\text{id} \mid c_1 \circ c_2 \mid \text{top} \mid \text{top}_{\rightarrow} \mid \text{top}_{\{l\}} \mid c_1 \rightarrow c_2 \mid \langle c_1, c_2 \rangle$ $\mid \pi_1 \mid \pi_2 \mid \{l : c\} \mid \text{dist}_{\{l\}} \mid \text{dist}_{\rightarrow}$
Target values	v	::=	$\lambda x. e \mid \langle \rangle \mid i \mid \langle v_1, v_2 \rangle \mid (c_1 \rightarrow c_2) v \mid \text{dist}_{\rightarrow} v \mid \text{top}_{\rightarrow} v$

■ **Figure 7** λ_c types, terms and coercions.

checking judgement $\Gamma \vdash E \Leftarrow A$ checks E against A in the context Γ . The disjointness judgement $A * B$ used in rule T-MERGE is shown in Fig. 6, which states that the types A and B are *disjoint*. The intuition of two types being disjoint is that their least upper bound is (isomorphic to) \top . The disjointness judgement is important in order to rule out ambiguous expressions such as $1, , 2$. Most of the typing and disjointness rules are standard and are explained in detail in previous work [46, 2].

3.4 Elaboration Semantics

The operational semantics of NeColus is given by elaborating source expressions E into target terms e . Our target language λ_c is the standard simply-typed call-by-value λ -calculus extended with singleton records, products and coercions. The syntax of λ_c is shown in Fig. 7. The meta-function $|\cdot|$ transforms NeColus types to λ_c types, and extends naturally to typing contexts. Its definition is in the appendix.

Explicit Coercions and Coercive Subtyping. The separate syntactic category for explicit coercions is a distinct difference from the prior works (in which they are regular terms). Our coercions are based on those of Henglein [32], and we add more forms due to our extra subtyping rules. Metavariable c ranges over coercions.⁶ Coercions express the conversion of a term from one type to another. Because of the addition of coercions, the grammar contains explicit coercion applications $c e$ as a term, and various unsaturated coercion applications as values. The use of explicit coercions is useful for the new semantic proof of coherence based

⁶ Coercions π_1 and π_2 subsume the first and second projection of pairs, respectively.

$$\boxed{c \vdash \tau_1 \triangleright \tau_2} \quad (\text{Coercion typing})$$

$$\begin{array}{c}
\text{COTYP-REFL} \\
\hline
\text{id} \vdash \tau \triangleright \tau
\end{array}
\quad
\begin{array}{c}
\text{COTYP-TRANS} \\
\frac{c_1 \vdash \tau_2 \triangleright \tau_3 \quad c_2 \vdash \tau_1 \triangleright \tau_2}{c_1 \circ c_2 \vdash \tau_1 \triangleright \tau_3}
\end{array}
\quad
\begin{array}{c}
\text{COTYP-TOP} \\
\hline
\text{top} \vdash \tau \triangleright \langle \rangle
\end{array}
\quad
\begin{array}{c}
\text{COTYP-TOPARR} \\
\hline
\text{top}_{\rightarrow} \vdash \langle \rangle \triangleright \langle \rangle \rightarrow \langle \rangle
\end{array}$$

$$\begin{array}{c}
\text{COTYP-TOPRCD} \\
\hline
\text{top}_{\{l\}} \vdash \langle l : \langle \rangle \rangle \triangleright \{l : \langle \rangle\}
\end{array}
\quad
\begin{array}{c}
\text{COTYP-ARR} \\
\frac{c_1 \vdash \tau'_1 \triangleright \tau_1 \quad c_2 \vdash \tau_2 \triangleright \tau'_2}{c_1 \rightarrow c_2 \vdash \tau_1 \rightarrow \tau_2 \triangleright \tau'_1 \rightarrow \tau'_2}
\end{array}
\quad
\begin{array}{c}
\text{COTYP-PAIR} \\
\frac{c_1 \vdash \tau_1 \triangleright \tau_2 \quad c_2 \vdash \tau_1 \triangleright \tau_3}{\langle c_1, c_2 \rangle \vdash \tau_1 \triangleright \tau_2 \times \tau_3}
\end{array}$$

$$\begin{array}{c}
\text{COTYP-PROJL} \\
\hline
\pi_1 \vdash \tau_1 \times \tau_2 \triangleright \tau_1
\end{array}
\quad
\begin{array}{c}
\text{COTYP-PROJR} \\
\hline
\pi_2 \vdash \tau_1 \times \tau_2 \triangleright \tau_2
\end{array}
\quad
\begin{array}{c}
\text{COTYP-RCD} \\
\frac{c \vdash \tau_1 \triangleright \tau_2}{\{l : c\} \vdash \{l : \tau_1\} \triangleright \{l : \tau_2\}}
\end{array}$$

$$\begin{array}{c}
\text{COTYP-DISTRCD} \\
\hline
\text{dist}_{\{l\}} \vdash \{l : \tau_1\} \times \{l : \tau_2\} \triangleright \{l : \tau_1 \times \tau_2\}
\end{array}
\quad
\begin{array}{c}
\text{COTYP-DISTARR} \\
\hline
\text{dist}_{\rightarrow} \vdash (\tau_1 \rightarrow \tau_2) \times (\tau_1 \rightarrow \tau_3) \triangleright \tau_1 \rightarrow \tau_2 \times \tau_3
\end{array}$$

■ **Figure 8** Coercion typing.

on logical relations. The subtyping judgement in Fig. 4 has the form $A <: B \rightsquigarrow c$, which says that the subtyping derivation of $A <: B$ produces a coercion c that converts terms of type $|A|$ to type $|B|$. Each subtyping rule has its own specific form of coercion.

Target Typing. The typing of λ_c has the form $\Delta \vdash e : \tau$, which is entirely standard. Only the typing of coercion applications, shown below, deserves attention:

$$\frac{\Delta \vdash e : \tau \quad c \vdash \tau \triangleright \tau'}{\Delta \vdash c e : \tau'} \text{ TYP-CAPP}$$

Here the judgement $c \vdash \tau_1 \triangleright \tau_2$ expresses the typing of coercions, which are essentially functions from τ_1 to τ_2 . Their typing rules correspond exactly to the subtyping rules of NeColus, as shown in Fig. 8.

Target Operational Semantics and Type Safety. The operational semantics of λ_c is mostly unremarkable. What may be interesting is the operational semantics of coercions. Figure 9 shows the single-step (\longrightarrow) reduction rules for coercions. Our coercion reduction rules are quite standard but not efficient in terms of space. Nevertheless, there is existing work on space-efficient coercions [60, 33], which should be applicable to our work as well. As standard, \longrightarrow^* is the reflexive, transitive closure of \longrightarrow . We show that λ_c is type safe:

► **Theorem 1** (Preservation). *If $\bullet \vdash e : \tau$ and $e \longrightarrow e'$, then $\bullet \vdash e' : \tau$.*

► **Theorem 2** (Progress). *If $\bullet \vdash e : \tau$, then either e is a value, or $\exists e'$ such that $e \longrightarrow e'$.*

Elaboration. We are now in a position to explain the elaboration judgements $\Gamma \vdash E \Rightarrow A \rightsquigarrow e$ and $\Gamma \vdash E \Leftarrow A \rightsquigarrow e$ in Fig. 5. The only interesting rule is rule T-SUB, which applies the coercion c produced by subtyping to the target term e to form a coercion application $c e$. All the other rules do straightforward translations between source and target expressions.

To conclude, we show two lemmas that relate NeColus expressions to λ_c terms.

► **Lemma 3** (Coercions preserve types). *If $A <: B \rightsquigarrow c$, then $c \vdash |A| \triangleright |B|$.*

$e \longrightarrow e'$ (Coercion reduction)			
STEP-ID	STEP-TRANS	STEP-TOP	STEP-TOPARR
$\overline{\text{id } v \longrightarrow v}$	$\overline{(c_1 \circ c_2) v \longrightarrow c_1 (c_2 v)}$	$\overline{\text{top } v \longrightarrow \langle \rangle}$	$\overline{(\text{top}_{\rightarrow} \langle \rangle) \langle \rangle \longrightarrow \langle \rangle}$
STEP-TOPRCD	STEP-PAIR	STEP-ARR	
$\overline{\text{top}_{\{l\}} \langle \rangle \longrightarrow \{l = \langle \rangle\}}$	$\overline{\langle c_1, c_2 \rangle v \longrightarrow \langle c_1 v, c_2 v \rangle}$	$\overline{((c_1 \rightarrow c_2) v_1) v_2 \longrightarrow c_2 (v_1 (c_1 v_2))}$	
STEP-DISTARR	STEP-PROJL		STEP-PROJR
$\overline{(\text{dist}_{\rightarrow} \langle v_1, v_2 \rangle) v_3 \longrightarrow \langle v_1 v_3, v_2 v_3 \rangle}$	$\overline{\pi_1 \langle v_1, v_2 \rangle \longrightarrow v_1}$		$\overline{\pi_2 \langle v_1, v_2 \rangle \longrightarrow v_2}$
STEP-CRCD	STEP-DISTRCD		
$\overline{\{l : c\} \{l = v\} \longrightarrow \{l = c v\}}$	$\overline{\text{dist}_{\{l\}} \{\{l = v_1\}, \{l = v_2\}\} \longrightarrow \{l = \langle v_1, v_2 \rangle\}}$		

■ **Figure 9** Coercion reduction.

Proof. By structural induction on the derivation of subtyping. ◀

► **Lemma 4** (Elaboration soundness). *We have that:*

- If $\Gamma \vdash E \Rightarrow A \rightsquigarrow e$, then $|\Gamma| \vdash e : |A|$.
- If $\Gamma \vdash E \Leftarrow A \rightsquigarrow e$, then $|\Gamma| \vdash e : |A|$.

Proof. By structural induction on the derivation of typing. ◀

3.5 Comparison with λ_i

Below we identify major differences between λ_i^+ and λ_i , which, when taken together, yield a simpler and more elegant system. The differences may seem superficial, but they have far-reaching impacts on the semantics, especially on coherence, our major topic in Section 4.

No Ordinary Types. Apart from the extra subtyping rules, there is an important difference from the λ_i subtyping relation. The subtyping relation of λ_i employs an auxiliary unary relation called **ordinary**, which plays a fundamental role for ensuring coherence and obtaining an algorithm [21]. The NeColus calculus discards the notion of ordinary types completely; this yields a clean and elegant formulation of the subtyping relation. Another minor difference is that due to the addition of the transitivity axiom (rule S-TRANS), rules S-ANDL and S-ANDR are simplified: an intersection type $A \& B$ is a subtype of both A and B , instead of the more general form $A \& B <: C$.

No Top-Like Types. There is a notable difference from the coercive subtyping of λ_i . Because of their syntactic proof method, they have special treatment for coercions of *top-like types* in order to retain coherence. For NeColus, as with ordinary types, we do not need this kind of ad-hoc treatment, thanks to the adoption of a more powerful proof method (cf. Section 4).

No Well-Formedness Judgement. A key difference from the type system of λ_i is the complete omission of the well-formedness judgement. In λ_i , the well-formedness judgement

$\Gamma \vdash A$ appears in both rules T-ABS and T-SUB. The sole purpose of this judgement is to enforce the invariant that all intersection types are disjoint. However, as Section 4 will explain, the syntactic restriction is unnecessary for coherence, and merely complicates the type system. The NeColus calculus discards this well-formedness judgement altogether in favour of a simpler design that is still coherent. An important implication is that even without adding BCD subtyping, NeColus is already more expressive than λ_i : an expression such as $1 : \text{Nat} \& \text{Nat}$ is accepted in NeColus but rejected in λ_i . This simplification is based on an important observation: incoherence can only originate in merges. Therefore disjointness checking is only necessary in rule T-MERGE.

4 Coherence

This section constructs logical relations to establish the coherence of NeColus. Finding a suitable definition of coherence for NeColus is already challenging in its own right. In what follows we reproduce the steps of finding a definition for coherence that is both intuitive and applicable. Then we present the construction of logical (equivalence) relations tailored to this definition, and the connection between logical equivalence and coherence.

4.1 In Search of Coherence

In λ_i the definition of coherence is based on α -equivalence. More specifically, their coherence property states that any two target terms that a source expression elaborates into must be exactly the same (up to α -equivalence). Unfortunately this syntactic notion of coherence is very fragile with respect to extensions. For example, it is not obvious how to retain this notion of coherence when adding more subtyping rules such as those in Fig. 4.

If we permit ourselves to consider only the syntactic aspects of expressions, then very few expressions can be considered equal. The syntactic view also conflicts with the intuition that the significance of an expression lies in its contribution to the *outcome* of a computation [31]. Drawing inspiration from a wide range of literature on contextual equivalence [41], we want a context-based notion of coherence. It is helpful to consider several examples before presenting the formal definition of our new semantically founded notion of coherence.

► **Example 5.** The same NeColus expression 3 can be typed Nat in many ways: for instance, by rule T-LIT; by rules T-SUB and S-REFL; or by rules T-SUB, S-TRANS, and S-REFL, resulting in λ_c terms 3 , $\text{id } 3$ and $(\text{id} \circ \text{id}) 3$, respectively. It is apparent that these three λ_c terms are “equal” in the sense that all reduce to the same numeral 3.

Expression Contexts and Contextual Equivalence. To formalize the intuition, we introduce the notion of *expression contexts*. An expression context \mathcal{D} is a term with a single hole $[\cdot]$ (possibly under some binders) in it. The syntax of λ_c expression contexts can be found in Fig. 10. The typing judgement for expression contexts has the form $\mathcal{D} : (\Delta \vdash \tau) \rightsquigarrow (\Delta' \vdash \tau')$ where $(\Delta \vdash \tau)$ indicates the type of the hole. This judgement essentially says that plugging a well-typed term $\Delta \vdash e : \tau$ into the context \mathcal{D} gives another well-typed term $\Delta' \vdash \mathcal{D}\{e\} : \tau'$. We define a *complete program* to mean any closed term of type Nat. The following two definitions capture the notion of *contextual equivalence*.

► **Definition 6** (Kleene Equality). Two complete programs, e and e' , are Kleene equal, written $e \simeq e'$, iff there exists i such that $e \longrightarrow^* i$ and $e' \longrightarrow^* i$.

► **Definition 7** (λ_c Contextual Equivalence).

λ_c contexts	\mathcal{D}	$::=$	$[\cdot] \mid \lambda x. \mathcal{D} \mid \mathcal{D} e_2 \mid e_1 \mathcal{D} \mid \langle \mathcal{D}, e_2 \rangle \mid \langle e_1, \mathcal{D} \rangle \mid c \mathcal{D} \mid \{l = \mathcal{D}\} \mid \mathcal{D}.l$
NeColus contexts	\mathcal{C}	$::=$	$[\cdot] \mid \lambda x. \mathcal{C} \mid \mathcal{C} E_2 \mid E_1 \mathcal{C} \mid E_1 , , \mathcal{C} \mid \mathcal{C} , , E_2 \mid \mathcal{C} : A \mid \{l = \mathcal{C}\} \mid \mathcal{C}.l$

■ **Figure 10** Expression contexts of NeColus and λ_c .

$$\Delta \vdash e_1 \simeq_{ctx} e_2 : \tau \quad \triangleq \quad \Delta \vdash e_1 : \tau \wedge \Delta \vdash e_2 : \tau \wedge \forall \mathcal{D}. \mathcal{D} : (\Delta \vdash \tau) \rightsquigarrow (\bullet \vdash \mathbf{Nat}) \implies \mathcal{D}\{e_1\} \simeq \mathcal{D}\{e_2\}$$

Regarding Example 5, it seems adequate to say that 3 and id 3 are contextually equivalent. Does this imply that coherence can be based on Definition 7? Unfortunately it cannot, as demonstrated by the following example.

► **Example 8.** It may be counter-intuitive that two λ_c terms $\lambda x. \pi_1 x$ and $\lambda x. \pi_2 x$ should also be considered equal. To see why, first note that they are both translations of the same NeColus expression: $(\lambda x. x) : \mathbf{Nat} \& \mathbf{Nat} \rightarrow \mathbf{Nat}$. What can we do with this lambda abstraction? We can apply it to $1 : \mathbf{Nat} \& \mathbf{Nat}$ for example. In that case, we get two translations $(\lambda x. \pi_1 x) \langle 1, 1 \rangle$ and $(\lambda x. \pi_2 x) \langle 1, 1 \rangle$, which both reduce to the same numeral 1. However, $\lambda x. \pi_1 x$ and $\lambda x. \pi_2 x$ are definitely not equal according to Definition 7, as one can find a context $[\cdot] \langle 1, 2 \rangle$ where the two terms reduce to two different numerals. The problem is that not every well-typed λ_c term can be obtained from a well-typed NeColus expression through the elaboration semantics. For example, $[\cdot] \langle 1, 2 \rangle$ should not be considered because the (non-disjoint) source expression $1 , , 2$ is rejected by the type system of the source calculus NeColus and thus never gets elaborated into $\langle 1, 2 \rangle$.

NeColus Contexts and Refined Contextual Equivalence. Example 8 hints at a shift from λ_c contexts to NeColus contexts \mathcal{C} , whose syntax is shown in Fig. 10. Due to the bidirectional nature of the type system, the typing judgement of \mathcal{C} features 4 different forms:

$$\begin{array}{ll} \mathcal{C} : (\Gamma \Rightarrow A) \mapsto (\Gamma' \Rightarrow A') \rightsquigarrow \mathcal{D} & \mathcal{C} : (\Gamma \Leftarrow A) \mapsto (\Gamma' \Rightarrow A') \rightsquigarrow \mathcal{D} \\ \mathcal{C} : (\Gamma \Rightarrow A) \mapsto (\Gamma' \Leftarrow A') \rightsquigarrow \mathcal{D} & \mathcal{C} : (\Gamma \Leftarrow A) \mapsto (\Gamma' \Leftarrow A') \rightsquigarrow \mathcal{D} \end{array}$$

We write $\mathcal{C} : (\Gamma \Leftrightarrow A) \mapsto (\Gamma' \Leftrightarrow A') \rightsquigarrow \mathcal{D}$ to abbreviate the above 4 different forms. Take $\mathcal{C} : (\Gamma \Rightarrow A) \mapsto (\Gamma' \Rightarrow A') \rightsquigarrow \mathcal{D}$ for example, it reads given a well-typed NeColus expression $\Gamma \vdash E \Rightarrow A$, we have $\Gamma' \vdash \mathcal{C}\{E\} \Rightarrow A'$. The judgement also generates a λ_c context \mathcal{D} such that $\mathcal{D} : (|\Gamma| \vdash |A|) \rightsquigarrow (|\Gamma'| \vdash |A'|)$ holds by construction. The typing rules appear in the appendix. Now we are ready to refine Definition 7's contextual equivalence to take into consideration both NeColus and λ_c contexts.

► **Definition 9** (NeColus Contextual Equivalence).

$$\Gamma \vdash E_1 \simeq_{ctx} E_2 : A \quad \triangleq \quad \forall e_1, e_2, \mathcal{C}, \mathcal{D}. \Gamma \vdash E_1 \Rightarrow A \rightsquigarrow e_1 \wedge \Gamma \vdash E_2 \Rightarrow A \rightsquigarrow e_2 \wedge \mathcal{C} : (\Gamma \Rightarrow A) \mapsto (\bullet \Rightarrow \mathbf{Nat}) \rightsquigarrow \mathcal{D} \implies \mathcal{D}\{e_1\} \simeq \mathcal{D}\{e_2\}$$

► **Remark.** For brevity we only consider expressions in the inference mode. Our Coq formalization is complete with two modes.

Regarding Example 8, a possible NeColus context is $[\cdot] 1 : (\bullet \Rightarrow \mathbf{Nat} \& \mathbf{Nat} \rightarrow \mathbf{Nat}) \mapsto (\bullet \Rightarrow \mathbf{Nat}) \rightsquigarrow [\cdot] \langle 1, 1 \rangle$. We can verify that both $\lambda x. \pi_1 x$ and $\lambda x. \pi_2 x$ produce 1 in the context $[\cdot] \langle 1, 1 \rangle$. Of course we should consider all possible contexts to be certain they are truly equal. From now on, we use the symbol \simeq_{ctx} to refer to contextual equivalence in Definition 9. With Definition 9 we can formally state that NeColus is coherent in the following theorem:

$$\begin{aligned}
(v_1, v_2) \in \mathcal{V}[\mathbf{Nat}; \mathbf{Nat}] &\triangleq \exists i, v_1 = v_2 = i \\
(v_1, v_2) \in \mathcal{V}[\tau_1 \rightarrow \tau_2; \tau'_1 \rightarrow \tau'_2] &\triangleq \forall (v, v') \in \mathcal{V}[\tau_1; \tau'_1], (v_1 v, v_2 v') \in \mathcal{E}[\tau_2; \tau'_2] \\
(\{l = v_1\}, \{l = v_2\}) \in \mathcal{V}[\{l : \tau_1\}; \{l : \tau_2\}] &\triangleq (v_1, v_2) \in \mathcal{V}[\tau_1; \tau_2] \\
(\langle v_1, v_2 \rangle, v_3) \in \mathcal{V}[\tau_1 \times \tau_2; \tau_3] &\triangleq (v_1, v_3) \in \mathcal{V}[\tau_1; \tau_3] \wedge (v_2, v_3) \in \mathcal{V}[\tau_2; \tau_3] \\
(v_3, \langle v_1, v_2 \rangle) \in \mathcal{V}[\tau_3; \tau_1 \times \tau_2] &\triangleq (v_3, v_1) \in \mathcal{V}[\tau_3; \tau_1] \wedge (v_3, v_2) \in \mathcal{V}[\tau_3; \tau_2] \\
(v_1, v_2) \in \mathcal{V}[\tau_1; \tau_2] &\triangleq \mathbf{true} \quad \text{otherwise} \\
(e_1, e_2) \in \mathcal{E}[\tau_1; \tau_2] &\triangleq \exists v_1 v_2, e_1 \longrightarrow^* v_1 \wedge e_2 \longrightarrow^* v_2 \wedge (v_1, v_2) \in \mathcal{V}[\tau_1; \tau_2]
\end{aligned}$$

■ **Figure 11** Logical relations for λ_c .

► **Theorem 10** (Coherence). *If $\Gamma \vdash E \Rightarrow A$ then $\Gamma \vdash E \simeq_{ctx} E : A$.*

For the same reason as in Definition 9, we only consider expressions in the inference mode. The rest of the section is devoted to proving that Theorem 10 holds.

4.2 Logical Relations

Intuitive as Definition 9 may seem, it is generally very hard to prove contextual equivalence directly, since it involves quantification over *all* possible contexts. Worse still, two kinds of contexts are involved in Theorem 10, which makes reasoning even more tedious. The key to simplifying the reasoning is to exploit types by using logical relations [63, 61, 48].

In Search of a Logical Relation. It is worth pausing to ponder what kind of relation we are looking for. The high-level intuition behind the relation is to capture the notion of “coherent” values. These values are unambiguous in every context. A moment of thought leads us to the following important observations:

► **Observation 1** (Disjoint values are unambiguous). The relation should relate values originating from disjoint intersection types. Those values are essentially translated from merges, and since rule T-MERGE ensures disjointness, they are unambiguous. For example, $\langle 1, \{l = 1\} \rangle$ corresponds to the type $\mathbf{Nat} \& \{l : \mathbf{Nat}\}$. It is always clear which one to choose (1 or $\{l = 1\}$) no matter how this pair is used in certain contexts.

► **Observation 2** (Duplication is unambiguous). The relation should relate values originating from non-disjoint intersection types, only if the values are duplicates. This may sound baffling since the whole point of disjointness is to rule out (ambiguous) expressions such as $1, , 2$. However, $1, , 2$ never gets elaborated, and the only values corresponding to $\mathbf{Nat} \& \mathbf{Nat}$ are those pairs such as $\langle 1, 1 \rangle, \langle 2, 2 \rangle$, etc. Those values are essentially generated from rule T-SUB and are also unambiguous.

To formalize values being “coherent” based on the above observations, Figure 11 defines two (binary) logical relations for λ_c , one for values ($\mathcal{V}[\tau_1; \tau_2]$) and one for terms ($\mathcal{E}[\tau_1; \tau_2]$). We require that any two values $(v_1, v_2) \in \mathcal{V}[\tau_1; \tau_2]$ are closed and well-typed. For succinctness, we write $\mathcal{V}[\tau]$ to mean $\mathcal{V}[\tau; \tau]$, and similarly for $\mathcal{E}[\tau]$.

► **Remark.** The logical relations are heterogeneous, parameterized by two types, one for each argument. This is intended to relate values of different types.

► **Remark.** The logical relations resemble those given by Biernacki and Polesiuk [8], as both are heterogeneous. However, two important differences are worth pointing out. Firstly, our

22:18 The Essence of Nested Composition

value relation for product types ($\mathcal{V}[\tau_1 \times \tau_2; \tau_3]$ and $\mathcal{V}[\tau_3; \tau_1 \times \tau_2]$) is unusual. Secondly, their value relation disallows relating functions with natural numbers, while ours does not. As we explain shortly, both points are related to disjointness.

First let us consider $\mathcal{V}[\tau_1; \tau_2]$. The first three cases are standard: Two natural numbers are related iff they are the same numeral. Two functions are related iff they map related arguments to related results. Two singleton records are related iff they have the same label and their fields are related. These cases reflect Observation 2: the same type corresponds to the same value.

In the next two cases one of the parameterized types is a product type. In those cases, the relation distributes over the product constructor \times . This may look strange at first, since the traditional way of relating pairs is by relating their components pairwise. That is, $\langle v_1, v_2 \rangle$ and $\langle v'_1, v'_2 \rangle$ are related iff (1) v_1 and v'_1 are related and (2) v_2 and v'_2 are related. According to our definition, we also require that (3) v_1 and v'_2 are related and (4) v_2 and v'_1 are related. The design of these two cases is influenced by the disjointness of intersection types. Below are two rules dealing with intersection types:

$$\frac{A_1 * B \quad A_2 * B}{A_1 \& A_2 * B} \text{D-ANDL} \qquad \frac{A * B_1 \quad A * B_2}{A * B_1 \& B_2} \text{D-ANDR}$$

Notice the structural similarity between these two rules and the two cases. Now it is clear that the cases for products manifests disjointness of intersection types, reflecting Observation 1. Together with the last case, we can show that disjointness and the value relation are connected by the following lemma:

► **Lemma 11** (Disjoint values are in a value relation). *If $A_1 * A_2$ and $v_1 : |A_1|$ and $v_2 : |A_2|$, then $(v_1, v_2) \in \mathcal{V}[|A_1|; |A_2|]$.*

Proof. By induction on the derivation of disjointness. ◀

Next we consider $\mathcal{E}[\tau_1; \tau_2]$, which is standard. Informally it expresses that two closed terms e_1 and e_2 are related iff they evaluate to two values v_1 and v_2 that are related.

Logical Equivalence. The logical relations can be lifted to open terms in the usual way. First we give the semantic interpretation of typing contexts:

► **Definition 12** (Interpretation of Typing Contexts). $(\gamma_1, \gamma_2) \in \mathcal{G}[\Delta_1; \Delta_2]$ is defined as follows:

$$\frac{}{(\emptyset, \emptyset) \in \mathcal{G}[\bullet; \bullet]} \qquad \frac{\begin{array}{c} (v_1, v_2) \in \mathcal{V}[\tau_1; \tau_2] \\ (\gamma_1, \gamma_2) \in \mathcal{G}[\Delta_1; \Delta_2] \quad \text{fresh } x \end{array}}{(\gamma_1[x \mapsto v_1], \gamma_2[x \mapsto v_2]) \in \mathcal{G}[\Delta_1, x : \tau_1; \Delta_2, x : \tau_2]}$$

Two open terms are related if every pair of related closing substitutions makes them related:

► **Definition 13** (Logical equivalence). Let $\Delta_1 \vdash e_1 : \tau_1$ and $\Delta_2 \vdash e_2 : \tau_2$.

$$\Delta_1; \Delta_2 \vdash e_1 \simeq_{log} e_2 : \tau_1; \tau_2 \triangleq \forall \gamma_1, \gamma_2. (\gamma_1, \gamma_2) \in \mathcal{G}[\Delta_1; \Delta_2] \implies (\gamma_1 e_1, \gamma_2 e_2) \in \mathcal{E}[\tau_1; \tau_2]$$

For succinctness, we write $\Delta \vdash e_1 \simeq_{log} e_2 : \tau$ to mean $\Delta; \Delta \vdash e_1 \simeq_{log} e_2 : \tau; \tau$.

4.3 Establishing Coherence

With all the machinery in place, we are now ready to prove Theorem 10. But we need several lemmas to set the stage.

First we show compatibility lemmas, which state that logical equivalence is preserved by every language construct. Most are standard and thus are omitted. We show only one compatibility lemma that is specific to our relations:

- **Lemma 14** (Coercion Compatibility). *Suppose that $c \vdash \tau_1 \triangleright \tau_2$,*
- *If $\Delta_1; \Delta_2 \vdash e_1 \simeq_{\text{log}} e_2 : \tau_1; \tau_0$ then $\Delta_1; \Delta_2 \vdash c e_1 \simeq_{\text{log}} e_2 : \tau_2; \tau_0$.*
 - *If $\Delta_1; \Delta_2 \vdash e_1 \simeq_{\text{log}} e_2 : \tau_0; \tau_1$ then $\Delta_1; \Delta_2 \vdash e_1 \simeq_{\text{log}} c e_2 : \tau_0; \tau_2$.*

Proof. By induction on the typing derivation of the coercion c . ◀

The “Fundamental Property” states that any well-typed expression is related to itself by the logical relation. In our elaboration setting, we rephrase it so that any two λ_c terms elaborated from the *same* NeColus expression are related by the logical relation. To prove it, we require Theorem 15.

- **Theorem 15** (Inference Uniqueness). *If $\Gamma \vdash E \Rightarrow A_1$ and $\Gamma \vdash E \Rightarrow A_2$, then $A_1 \equiv A_2$.*

- **Theorem 16** (Fundamental Property). *We have that:*
- *If $\Gamma \vdash E \Rightarrow A \rightsquigarrow e$ and $\Gamma \vdash E \Rightarrow A \rightsquigarrow e'$, then $|\Gamma| \vdash e \simeq_{\text{log}} e' : |A|$.*
 - *If $\Gamma \vdash E \Leftarrow A \rightsquigarrow e$ and $\Gamma \vdash E \Leftarrow A \rightsquigarrow e'$, then $|\Gamma| \vdash e \simeq_{\text{log}} e' : |A|$.*

Proof. The proof follows by induction on the first derivation. The most interesting case is rule T-SUB where we need Theorem 15 to be able to apply the induction hypothesis. Then we apply Lemma 14 to say that the coercion generated preserves the relation between terms. For the other cases we use the appropriate compatibility lemmas. ◀

► **Remark.** It is interesting to ask whether the Fundamental Property holds in the target language. That is, if $\Delta \vdash e : \tau$ then $\Delta \vdash e \simeq_{\text{log}} e : \tau$. Clearly this does not hold for every well-typed λ_c term. However, as we have emphasized, we do not need to consider every λ_c term. Our logical relation is carefully formulated so that the Fundamental Property holds in the source language.

We show that logical equivalence is preserved by NeColus contexts:

- **Lemma 17** (Congruence). *If $\mathcal{C} : (\Gamma \Leftarrow A) \mapsto (\Gamma' \Leftarrow A') \rightsquigarrow \mathcal{D}$, $\Gamma \vdash E_1 \Leftarrow A \rightsquigarrow e_1$, $\Gamma \vdash E_2 \Leftarrow A \rightsquigarrow e_2$ and $|\Gamma| \vdash e_1 \simeq_{\text{log}} e_2 : |A|$, then $|\Gamma'| \vdash \mathcal{D}\{e_1\} \simeq_{\text{log}} \mathcal{D}\{e_2\} : |A'|$.*

Proof. By induction on the typing derivation of the context \mathcal{C} , and applying the compatibility lemmas where appropriate. ◀

- **Lemma 18** (Adequacy). *If $\bullet \vdash e_1 \simeq_{\text{log}} e_2 : \text{Nat}$ then $e_1 \simeq e_2$.*

Proof. Adequacy follows easily from the definition of the logical relation. ◀

Next up is the proof that logical relation is sound with respect to contextual equivalence:

- **Theorem 19** (Soundness w.r.t. Contextual Equivalence). *If $\Gamma \vdash E_1 \Rightarrow A \rightsquigarrow e_1$ and $\Gamma \vdash E_2 \Rightarrow A \rightsquigarrow e_2$ and $|\Gamma| \vdash e_1 \simeq_{\text{log}} e_2 : |A|$ then $\Gamma \vdash E_1 \simeq_{\text{ctx}} E_2 : A$.*

Proof. From Definition 9, we are given a context $\mathcal{C} : (\Gamma \Rightarrow A) \mapsto (\bullet \Rightarrow \text{Nat}) \rightsquigarrow \mathcal{D}$. By Lemma 17 we have $\bullet \vdash \mathcal{D}\{e_1\} \simeq_{\text{log}} \mathcal{D}\{e_2\} : \text{Nat}$, thus $\mathcal{D}\{e_1\} \simeq \mathcal{D}\{e_2\}$ by Lemma 18. ◀

Armed with Theorem 16 and Theorem 19, coherence follows directly.

► **Theorem 10** (Coherence). *If $\Gamma \vdash E \Rightarrow A$ then $\Gamma \vdash E \simeq_{ctx} E : A$.*

Proof. Immediate from Theorem 16 and Theorem 19. ◀

4.4 Some Interesting Corollaries

To showcase the strength of the new proof method, we can derive some interesting corollaries. For the most part, they are direct consequences of logical equivalence which carry over to contextual equivalence.

Corollary 20 says that merging a term of some type with something else does not affect its semantics. Corollary 21 and Corollary 22 express that merges are commutative and associative, respectively. Corollary 23 states that coercions from the same types are “coherent”.

► **Corollary 20** (Merge is Neutral). *If $\Gamma \vdash E_1 \Rightarrow A$ and $\Gamma \vdash E_1, E_2 \Rightarrow A$, then $\Gamma \vdash E_1 \simeq_{ctx} E_1, E_2 : A$*

► **Corollary 21** (Merge is Commutative). *If $\Gamma \vdash E_1, E_2 \Rightarrow A$ and $\Gamma \vdash E_2, E_1 \Rightarrow A$, then $\Gamma \vdash E_1, E_2 \simeq_{ctx} E_2, E_1 : A$.*

► **Corollary 22** (Merge is Associative). *If $\Gamma \vdash (E_1, E_2), E_3 \Rightarrow A$ and $\Gamma \vdash E_1, (E_2, E_3) \Rightarrow A$, then $\Gamma \vdash (E_1, E_2), E_3 \simeq_{ctx} E_1, (E_2, E_3) : A$.*

► **Corollary 23** (Coercions Preserve Semantics). *If $A <: B \rightsquigarrow c_1$ and $A <: B \rightsquigarrow c_2$, then $\Delta \vdash \lambda x. c_1 x \simeq_{log} \lambda x. c_2 x : |A| \rightarrow |B|$.*

5 Algorithmic Subtyping

This section presents an algorithm that implements the subtyping relation in Fig. 4. While BCD subtyping is well-known, the presence of a transitivity axiom in the rules means that the system is not algorithmic. This raises an obvious question: how to obtain an algorithm for this subtyping relation? Laurent [37] has shown that simply dropping the transitivity rule from the BCD system is not possible without losing expressivity. Hence, this avenue for obtaining an algorithm is not available. Instead, we adapt Pierce’s decision procedure [47] for a subtyping system (closely related to BCD) to obtain a sound and complete algorithm for our BCD extension. Our algorithm extends Pierce’s decision procedure with subtyping of singleton records and coercion generation. We prove in Coq that the algorithm is sound and complete with respect to the declarative version. At the same time we find some errors and missing lemmas in Pierce’s original manual proofs.

5.1 The Subtyping Algorithm

Figure 12 shows the algorithmic subtyping judgement $\mathcal{L} \vdash A <: B \rightsquigarrow c$. This judgement is the algorithmic counterpart of the declarative judgement $A <: \mathcal{L} \rightarrow B \rightsquigarrow c$, where the symbol \mathcal{L} stands for a queue of types and labels. Definition 24 converts a queue to a type:

► **Definition 24.** $\mathcal{L} \rightarrow A$ is inductively defined as follows:

$$\boxed{\quad} \rightarrow A = A \quad (\mathcal{L}, B) \rightarrow A = \mathcal{L} \rightarrow (B \rightarrow A) \quad (\mathcal{L}, \{l\}) \rightarrow A = \mathcal{L} \rightarrow \{l : A\}$$

$$\boxed{\mathcal{L} \vdash A \prec: B \rightsquigarrow c} \quad (\text{Algorithmic subtyping})$$

$$\begin{array}{c}
\text{A-AND} \\
\frac{\mathcal{L} \vdash A \prec: B_1 \rightsquigarrow c_1 \quad \mathcal{L} \vdash A \prec: B_2 \rightsquigarrow c_2}{\mathcal{L} \vdash A \prec: B_1 \& B_2 \rightsquigarrow \llbracket \mathcal{L} \rrbracket_{\&} \circ \langle c_1, c_2 \rangle}
\end{array}
\quad
\begin{array}{c}
\text{A-ARR} \\
\frac{\mathcal{L}, B_1 \vdash A \prec: B_2 \rightsquigarrow c}{\mathcal{L} \vdash A \prec: B_1 \rightarrow B_2 \rightsquigarrow c}
\end{array}
\quad
\begin{array}{c}
\text{A-RCD} \\
\frac{\mathcal{L}, \{l\} \vdash A \prec: B \rightsquigarrow c}{\mathcal{L} \vdash A \prec: \{l: B\} \rightsquigarrow c}
\end{array}$$

$$\begin{array}{c}
\text{A-TOP} \\
\frac{}{\mathcal{L} \vdash A \prec: \top \rightsquigarrow \llbracket \mathcal{L} \rrbracket_{\top} \circ \text{top}}
\end{array}
\quad
\begin{array}{c}
\text{A-ARRNAT} \\
\frac{\boxed{\vdash} \vdash A \prec: A_1 \rightsquigarrow c_1 \quad \mathcal{L} \vdash A_2 \prec: \text{Nat} \rightsquigarrow c_2}{A, \mathcal{L} \vdash A_1 \rightarrow A_2 \prec: \text{Nat} \rightsquigarrow c_1 \rightarrow c_2}
\end{array}$$

$$\begin{array}{c}
\text{A-RCDNAT} \\
\frac{\mathcal{L} \vdash A \prec: \text{Nat} \rightsquigarrow c}{\{l\}, \mathcal{L} \vdash \{l: A\} \prec: \text{Nat} \rightsquigarrow \{l: c\}}
\end{array}
\quad
\begin{array}{c}
\text{A-ANDN1} \\
\frac{\mathcal{L} \vdash A_1 \prec: \text{Nat} \rightsquigarrow c}{\mathcal{L} \vdash A_1 \& A_2 \prec: \text{Nat} \rightsquigarrow c \circ \pi_1}
\end{array}$$

$$\begin{array}{c}
\text{A-ANDN2} \\
\frac{\mathcal{L} \vdash A_2 \prec: \text{Nat} \rightsquigarrow c}{\mathcal{L} \vdash A_1 \& A_2 \prec: \text{Nat} \rightsquigarrow c \circ \pi_2}
\end{array}
\quad
\begin{array}{c}
\text{A-NAT} \\
\frac{}{\boxed{\vdash} \vdash \text{Nat} \prec: \text{Nat} \rightsquigarrow \text{id}}
\end{array}$$

■ **Figure 12** Algorithmic subtyping of NeColus.

For instance, if $\mathcal{L} = A, B, \{l\}$, then $\mathcal{L} \rightarrow C$ abbreviates $A \rightarrow B \rightarrow \{l: C\}$.

The basic idea of $\mathcal{L} \vdash A \prec: B \rightsquigarrow c$ is to first perform a structural analysis of B , which descends into both sides of $\&$'s (rule A-AND), into the right side of \rightarrow 's (rule A-ARR), and into the fields of records (rule A-RCD) until it reaches one of the two base cases, Nat or \top . If the base case is \top , then the subtyping holds trivially (rule A-TOP). If the base case is Nat , the algorithm performs a structural analysis of A , in which \mathcal{L} plays an important role. The left sides of \rightarrow 's are pushed onto \mathcal{L} as they are encountered in B and popped off again later, left to right, as \rightarrow 's are encountered in A (rule A-ARRNAT). Similarly, the labels are pushed onto \mathcal{L} as they are encountered in B and popped off again later, left to right, as records are encountered in A (rule A-RCDNAT). The remaining rules are similar to their declarative counterparts. Let us illustrate the algorithm with an example derivation (for space reasons we use N and S to denote Nat and String respectively), which is essentially the one used by the `add` field in Section 2. The readers can try to give a corresponding derivation using the declarative subtyping and see how rule S-TRANS plays an essential role there.

$$\frac{\frac{\frac{D \quad D'}{\{l\}, \text{N} \& \text{S}, \text{N} \& \text{S} \vdash \{l: \text{N} \rightarrow \text{N} \rightarrow \text{N}\} \& \{l: \text{S} \rightarrow \text{S} \rightarrow \text{S}\} \prec: \text{N} \& \text{S}}\{l\} \vdash \{l: \text{N} \rightarrow \text{N} \rightarrow \text{N}\} \& \{l: \text{S} \rightarrow \text{S} \rightarrow \text{S}\} \prec: \text{N} \& \text{S} \rightarrow \text{N} \& \text{S} \rightarrow \text{N} \& \text{S}}\{l: \text{N} \rightarrow \text{N} \rightarrow \text{N}\} \& \{l: \text{S} \rightarrow \text{S} \rightarrow \text{S}\} \prec: \{l: \text{N} \& \text{S} \rightarrow \text{N} \& \text{S} \rightarrow \text{N} \& \text{S}\}}{\text{A-AND}} \quad \text{A-ARR}(twice) \quad \text{A-RCD}$$

where the sub-derivation D is shown below (D' is similar):

$$\frac{\frac{\frac{\dots}{\text{N} \& \text{S} \prec: \text{N}} \quad \frac{\dots}{\text{N} \& \text{S} \vdash \text{N} \rightarrow \text{N} \prec: \text{N}}}{\text{N} \& \text{S}, \text{N} \& \text{S} \vdash \text{N} \rightarrow \text{N} \prec: \text{N}} \quad \text{A-ARRNAT}}{\{l\}, \text{N} \& \text{S}, \text{N} \& \text{S} \vdash \{l: \text{N} \rightarrow \text{N} \rightarrow \text{N}\} \prec: \text{N}} \quad \text{A-RCDNAT}}{\{l\}, \text{N} \& \text{S}, \text{N} \& \text{S} \vdash \{l: \text{N} \rightarrow \text{N} \rightarrow \text{N}\} \& \{l: \text{S} \rightarrow \text{S} \rightarrow \text{S}\} \prec: \text{N}} \quad \text{A-ANDN1}$$

Now consider the coercions. Algorithmic subtyping uses the same set of coercions as declarative subtyping. However, because algorithmic subtyping has a different structure, the rules generate slightly more complicated coercions. Two meta-functions $\llbracket \cdot \rrbracket_{\top}$ and $\llbracket \cdot \rrbracket_{\&}$ used in rules A-TOP and A-AND respectively, are meant to generate correct forms of coercions. They are defined recursively on \mathcal{L} and are shown in Fig. 13.

$$\begin{array}{ll}
\llbracket \square \rrbracket_{\top} = \text{top} & \llbracket \square \rrbracket_{\&} = \text{id} \\
\llbracket \{l\}, \mathcal{L} \rrbracket_{\top} = \{l : \llbracket \mathcal{L} \rrbracket_{\top}\} \circ \text{top}_{\{l\}} & \llbracket \{l\}, \mathcal{L} \rrbracket_{\&} = \{l : \llbracket \mathcal{L} \rrbracket_{\&}\} \circ \text{dist}_{\{l\}} \\
\llbracket A, \mathcal{L} \rrbracket_{\top} = (\text{top} \rightarrow \llbracket \mathcal{L} \rrbracket_{\top}) \circ (\text{top}_{\rightarrow} \circ \text{top}) & \llbracket A, \mathcal{L} \rrbracket_{\&} = (\text{id} \rightarrow \llbracket \mathcal{L} \rrbracket_{\&}) \circ \text{dist}_{\rightarrow}
\end{array}$$

■ **Figure 13** Meta-functions of coercions.

5.2 Correctness of the Algorithm

To establish the correctness of the algorithm, we must show that the algorithm is both sound and complete with respect to the declarative specification. While soundness follows quite easily, completeness is much harder. The proof of completeness essentially follows that of Pierce [47] in that we need to show the algorithmic subtyping is reflexive and transitive.

Soundness of the Algorithm. The proof of soundness is straightforward.

► **Theorem 25** (Soundness). *If $\mathcal{L} \vdash A \prec: B \rightsquigarrow c$ then $A \prec: \mathcal{L} \rightarrow B \rightsquigarrow c$.*

Proof. By induction on the derivation of the algorithmic subtyping. ◀

Completeness of the Algorithm. Completeness, however, is much harder. The reason is that, due to the use of \mathcal{L} , reflexivity and transitivity are not entirely obvious. We need to strengthen the induction hypothesis by introducing the notion of a set, $\mathcal{U}(A)$, of “reflexive supertypes” of A , as defined below:

$$\begin{array}{lll}
\mathcal{U}(\top) \triangleq \{\top\} & \mathcal{U}(\text{Nat}) \triangleq \{\text{Nat}\} & \mathcal{U}(\{l : A\}) \triangleq \{\{l : B\} \mid B \in \mathcal{U}(A)\} \\
\mathcal{U}(A \& B) \triangleq \mathcal{U}(A) \cup \mathcal{U}(B) \cup \{A \& B\} & & \mathcal{U}(A \rightarrow B) \triangleq \{A \rightarrow C \mid C \in \mathcal{U}(B)\}
\end{array}$$

We show two lemmas about $\mathcal{U}(A)$ that are crucial in the subsequent proofs.

► **Lemma 26.** $A \in \mathcal{U}(A)$

Proof. By induction on the structure of A . ◀

► **Lemma 27.** *If $A \in \mathcal{U}(B)$ and $B \in \mathcal{U}(C)$, then $A \in \mathcal{U}(C)$.*

Proof. By induction on the structure of B . ◀

► **Remark.** Lemma 27 is not found in Pierce’s proofs [47], which is crucial in Lemma 28, from which reflexivity (Lemma 29) follows immediately.

► **Lemma 28.** *If $\mathcal{L} \rightarrow B \in \mathcal{U}(A)$ then there exists c such that $\mathcal{L} \vdash A \prec: B \rightsquigarrow c$.*

Proof. By induction on $\text{size}(A) + \text{size}(B) + \text{size}(\mathcal{L})$. ◀

Now it immediately follows that the algorithmic subtyping is reflexive.

► **Lemma 29** (Reflexivity). *For every A there exists c such that $\square \vdash A \prec: A \rightsquigarrow c$.*

Proof. Immediate from Lemma 26 and Lemma 28. ◀

We omit the details of the proof of transitivity.

► **Lemma 30** (Transitivity). *If $\square \vdash A_1 \prec: A_2 \rightsquigarrow c_1$ and $\square \vdash A_2 \prec: A_3 \rightsquigarrow c_2$, then there exists c such that $\square \vdash A_1 \prec: A_3 \rightsquigarrow c$.*

With reflexivity and transitivity in position, we show the main theorem.

► **Theorem 31** (Completeness). *If $A <: B \rightsquigarrow c$ then there exists c' such that $\square \vdash A <: B \rightsquigarrow c'$.*

Proof. By induction on the derivation of the declarative subtyping and applying Lemmas 29 and 30 where appropriate. ◀

► **Remark.** Pierce's proof is wrong [47, pp. 20, Case F] in the case

$$\frac{B_1 <: A_1 \rightsquigarrow c_1 \quad A_2 <: B_2 \rightsquigarrow c_2}{A_1 \rightarrow A_2 <: B_1 \rightarrow B_2 \rightsquigarrow c_1 \rightarrow c_2} \text{S-ARR}$$

where he concludes from the inductive hypotheses $\square \vdash B_1 <: A_1$ and $\square \vdash A_2 <: B_2$ that $\square \vdash A_1 \rightarrow A_2 <: B_1 \rightarrow B_2$ (rules 6a and 3). However his rule 6a (our rule A-ARRNAT) only works for *primitive types*, and is thus not applicable in this case. Instead we need a few technical lemmas to support the argument.

► **Remark.** It is worth pointing out that the two coercions c and c' in Theorem 31 are contextually equivalent, which follows from Theorem 25 and Corollary 23.

6 Related Work

Coherence. In calculi that feature coercive subtyping, a semantics that interprets the subtyping judgement by introducing explicit coercions is typically defined on typing derivations rather than on typing judgements. A natural question that arises for such systems is whether the semantics is *coherent*, i.e., distinct typing derivations of the same typing judgement possess the same meaning. Since Reynolds [55] proved the coherence of a calculus with intersection types, based on the denotational semantics for intersection types, many researchers have studied the problem of coherence in a variety of typed calculi. Below we summarize two commonly-found approaches in the literature.

Breazu-Tannen et al. [10] proved the coherence of a coercion translation from Fun [13] extended with recursive types to System F by showing that any two typing derivations of the same judgement are normalizable to a unique normal derivation. Ghelli [20] presented a translation of System F_{\leq} into a calculus with explicit coercions and showed that any derivations of the same judgement are translated to terms that are normalizable to a unique normal form. Following the same approach, Schwinghammer [59] proved the coherence of coercion translation from Moggi's computational lambda calculus [40] with subtyping.

Central to the first approach is to find a normal form for a representation of the derivation and show that normal forms are unique for a given typing judgement. However, this approach cannot be directly applied to Curry-style calculi, i.e, where the lambda abstractions are not type annotated. Also this line of reasoning cannot be used when the calculus has general recursion. Biernacki and Polesiuk [8] considered the coherence problem of coercion semantics. Their criterion for coherence of the translation is *contextual equivalence* in the target calculus. They presented a construction of logical relations for establishing so constructed coherence for coercion semantics, applicable in a variety of calculi, including delimited continuations and control-effect subtyping.

As far as we know, our work is the first to use logical relations to show the coherence for intersection types and the merge operator. The BCD subtyping in our setting poses a non-trivial complication over Biernacki and Polesiuk's simple structural subtyping. Indeed, because any two coercions between given types are behaviorally equivalent in the target language, their coherence reasoning can all take place in the target language. This is not

true in our setting, where coercions can be distinguished by arbitrary target programs, but not those that are elaborations of source programs. Hence, we have to restrict our reasoning to the latter class, which is reflected in a more complicated notion of contextual equivalence and our logical relation’s non-trivial treatment of pairs.

Intersection Types and the Merge Operator. Forsythe [54] has intersection types and a merge-like operator. However to ensure coherence, various restrictions were added to limit the use of merges. For example, in Forsythe merges cannot contain more than one function. Castagna et al. [15] proposed a coherent calculus with a special merge operator that works on functions only. More recently, Dunfield [23] shows significant expressiveness of type systems with intersection types and a merge operator. However his calculus lacks coherence. The limitation was addressed by Oliveira et al. [46], who introduced disjointness to ensure coherence. The combination of intersection types, a merge operator and parametric polymorphism, while achieving coherence was first studied in the F_i calculus [2]. Compared to prior work, NeColus simplifies type systems with disjoint intersection types by removing several restrictions. Furthermore, NeColus adopts a more powerful subtyping relation based on BCD subtyping, which in turn requires the use of a more powerful logical relations based method for proving coherence.

BCD Type System and Decidability. The BCD type system was first introduced by Barendregt et al. [4]. It is derived from a filter lambda model in order to characterize exactly the strongly normalizing terms. The BCD type system features a powerful subtyping relation, which serves as a base for our subtyping relation. Bessai et al. [5] showed how to type classes and mixins in a BCD-style record calculus with Bracha-Cook’s merge operator [9]. Their merge can only operate on records, and they only study a type assignment system. The decidability of BCD subtyping has been shown in several works [47, 35, 52, 62]. Laurent [36] has formalized the relation in Coq in order to eliminate transitivity cuts from it, but his formalization does not deliver an algorithm. Based on Statman’s work [62], Bessai et al. [6] show a formally verified subtyping algorithm in Coq. Our Coq formalization follows a different idea based on Pierce’s decision procedure [47], which is shown to be easily extensible to coercions and records. In the course of our mechanization we identified several mistakes in Pierce’s proofs, as well as some important missing lemmas.

Family Polymorphism. There has been much work on family polymorphism since Ernst’s original proposal [25]. Family polymorphism provides an elegant solution to the Expression Problem. Although a simple Scala solution does exist without requiring family polymorphism (e.g., see Wang and Oliveira [65]), Scala does not support nested composition: programmers need to manually compose all the classes from multiple extensions. Generally speaking, systems that support family polymorphism usually require quite sophisticated mechanisms such as dependent types.

There are two approaches to family polymorphism: the original *object family* approach of Beta (e.g., virtual classes [38]) treats nested classes as attributes of objects of the family classes. Path-dependent types are used to ensure type safety for virtual types and virtual classes in the calculus *vc* [27]. As for conflicts, *vc* follows the mixin-style by allowing the rightmost class to take precedence. This is in contrast to NeColus where conflicts are detected statically and resolved explicitly. In the *class family* approach of Concord [34], Jx and J& [42, 43], nested classes and types are attributes of the family classes directly. Jx supports *nested inheritance*, a class family mechanism that allows nesting of arbitrary depth. J& is a language that supports *nested intersection*, building on top of Jx. Similar to NeColus,

intersection types play an important role in J&, which are used to compose packages/classes. Unlike NeColus, J& does not have a merge-like operator. When conflicts arise, prefix types can be exploited to resolve the ambiguity. J&_s [50] is an extension of the Java language that adds class sharing to J&. Saito et al. [57] identified a minimal, lightweight set of language features to enable family polymorphism, Corradi et al. [19] present a language design that integrates modular composition and nesting of Java-like classes. It features a set of composition operators that allow to manipulate nested classes at any depth level. More recently, a Java-like language called Familia [66] were proposed to combine subtyping polymorphism, parametric polymorphism and family polymorphism. The object and class family approaches have even been combined by the work on Tribe [16].

Compared with those systems, which usually focus on getting a relatively complex Java-like language with family polymorphism, NeColus focuses on a minimal calculus that supports nested composition. NeColus shows that a calculus with the merge operator and a variant of BCD captures the essence of nested composition. Moreover NeColus enables new insights on the subtyping relations of families. NeColus's goal is not to support full family polymorphism which, besides nested composition, also requires dealing with other features such as self types [12, 56] and mutable state. Supporting these features is not the focus of this paper, but we expect to investigate those features in the future.

7 Conclusions and Future Work

We have proposed NeColus, a type-safe and coherent calculus with disjoint intersection types, and support for nested composition/subtyping. It improves upon earlier work with a more flexible notion of disjoint intersection types, which leads to a clean and elegant formulation of the type system. Due to the added flexibility we have had to employ a more powerful proof method based on logical relations to rigorously prove coherence. We also show how NeColus supports essential features of family polymorphism, such as nested composition. We believe NeColus provides insights into family polymorphism, and has potential for practical applications for extensible software designs.

A natural direction for future work is to enrich NeColus with parametric polymorphism. There is abundant literature on logical relations for parametric polymorphism [53]. The main challenge in the definition of the logical relation is the clause that relates type variables with arbitrary types. Careful measures are to be taken to avoid potential circularity due to impredicativity.⁷ With the combination of parametric polymorphism and nested composition, an interesting application that we intend to investigate is native support for a highly modular form of *Object Algebras* [45, 7] and VISITORS (or the finally tagless approach [14]).

Another direction for future work is to add mutable references, which would affect two aspects of our metatheory: type safety and coherence. For type safety, we expect that lessons learned from previous work on family polymorphism and mutability on OO apply to our work. For example, it is well-known that subtyping in the presence of mutable state often needs restrictions. Given such suitable restrictions we expect that type-safety in the presence of mutability still holds. For coherence, it would be a major technical challenge to adjust our coherence proof and its Coq mechanization: logical relations that account for mutable state (e.g., see Ahmed's thesis [1]) introduce significant complexity.

⁷ Our prototype implementation already supports polymorphism, but we are still in the process of extending our Coq development with polymorphism.

References

- 1 Amal Jamil Ahmed. *Semantics of types for mutable state*. PhD thesis, Princeton University, 2004.
- 2 João Alpuim, Bruno C. d. S. Oliveira, and Zhiyuan Shi. Disjoint polymorphism. In *European Symposium on Programming*, 2017.
- 3 Nada Amin, Adriaan Moors, and Martin Odersky. Dependent object types. In *Workshop on Foundations of Object-Oriented Languages*, 2012.
- 4 Henk Barendregt, Mario Coppo, and Mariangiola Dezani-Ciancaglini. A filter lambda model and the completeness of type assignment. *The journal of symbolic logic*, 48(04):931–940, 1983.
- 5 Jan Bessai, Boris Döder, Andrej Dudenhefner, Tzu-Chun Chen, and Ugo de'Liguoro. Typing classes and mixins with intersection types. In *Workshop on Intersection Types and Related Systems (ITRS)*, 2014.
- 6 Jan Bessai, Andrej Dudenhefner, Boris Döder, and Jakob Rehof. Extracting a formally verified subtyping algorithm for intersection types from ideals and filters. In *TYPES*, 2016.
- 7 Xuan Bi and Bruno C. d. S. Oliveira. Typed first-class traits. In *European Conference on Object-Oriented Programming*, 2018.
- 8 Dariusz Biernacki and Piotr Polesiuk. Logical relations for coherence of effect subtyping. In *LIPICs*, 2015.
- 9 Gilad Bracha and William R. Cook. Mixin-based inheritance. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 1990.
- 10 Val Breazu-Tannen, Thierry Coquand, Carl A. Gunter, and Andre Scedrov. Inheritance as implicit coercion. *Information and Computation*, 93(1):172–221, 1991.
- 11 Kim Bruce, Luca Cardelli, Giuseppe Castagna, Gary T Leavens, Benjamin Pierce, et al. On binary methods. In *Theory and Practice of Object Systems*, 1996.
- 12 Kim B. Bruce, Angela Schuett, and Robert van Gent. Polytoil: A type-safe polymorphic object-oriented language. In *European Conference on Object-Oriented Programming*, 1995.
- 13 Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–523, 1985.
- 14 Jacques Carette, Oleg Kiselyov, and Chung-Chieh Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *Journal of Functional Programming*, 19(05):509, 2009.
- 15 Giuseppe Castagna, Giorgio Ghelli, and Giuseppe Longo. A calculus for overloaded functions with subtyping. In *LFP*, 1992.
- 16 David Clarke, Sophia Drossopoulou, James Noble, and Tobias Wrigstad. Tribe: More Types for Virtual Classes. In *AOSD*, 2007.
- 17 Adriana B Compagnoni and Benjamin C Pierce. Higher-order intersection types and multiple inheritance. *MSCS*, 6(5):469–501, 1996.
- 18 Mario Coppo, Mariangiola Dezani-Ciancaglini, and Betti Venneri. Functional characters of solvable terms. *Mathematical Logic Quarterly*, 1981.
- 19 Andrea Corradi, Marco Servetto, and Elena Zucca. DeepFJig — modular composition of nested classes. *The Journal of Object Technology*, 11(2):1:1, 2012.
- 20 Pierre-Louis Curien and Giorgio Ghelli. Coherence of subsumption, minimum typing and type-checking in $f \leq$. *MSCS*, 2(01):55, 1992.
- 21 Rowan Davies and Frank Pfenning. Intersection types and computational effects. In *International Conference on Functional Programming*, 2000.
- 22 Stéphane Ducasse, Oscar Nierstrasz, Nathanael Schärli, Roel Wuyts, and Andrew P Black. Traits: A mechanism for fine-grained reuse. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(2):331–388, 2006.

- 23 Joshua Dunfield. Elaborating intersection and union types. *Journal of Functional Programming*, 24:133–165, 2014.
- 24 Joshua Dunfield and Frank Pfenning. Type assignment for intersections and unions in call-by-value languages. In *Foundations of Software Science and Computation Structure (FoSSaCS)*, 2003.
- 25 Erik Ernst. Family polymorphism. In *European Conference on Object-Oriented Programming*, 2001.
- 26 Erik Ernst. Higher-order hierarchies. In *European Conference on Object-Oriented Programming*, 2003.
- 27 Erik Ernst, Klaus Ostermann, and William R. Cook. A virtual class calculus. In *Symposium on Principles of Programming Languages*, 2006.
- 28 Facebook. Flow. <https://flow.org/>, 2014.
- 29 Kathleen Fisher and John Reppy. A typed calculus of traits. In *Workshop on Foundations of Object-Oriented Languages*, 2004.
- 30 Tim Freeman and Frank Pfenning. Refinement types for ML. In *Conference on Programming Language Design and Implementation*, 1991.
- 31 Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, 2016.
- 32 Fritz Henglein. Dynamic typing: syntax and proof theory. *Science of Computer Programming*, 22(3):197–230, jun 1994.
- 33 David Herman, Aaron Tomb, and Cormac Flanagan. Space-efficient gradual typing. *Higher-Order and Symbolic Computation*, 23(2):167, 2010.
- 34 Paul Jolly, Sophia Drossopoulou, Christopher Anderson, and Klaus Ostermann. Simple dependent types: Concord. In *European Conference on Object-Oriented Programming Workshop on Formal Techniques for Java Programs (FTfJP)*, 2004.
- 35 Toshihiko Kurata and Masako Takahashi. Decidable properties of intersection type systems. *Typed Lambda Calculi and Applications*, pages 297–311, 1995.
- 36 Olivier Laurent. Intersection types with subtyping by means of cut elimination. *Fundamenta Informaticae*, 121(1-4):203–226, 2012.
- 37 Olivier Laurent. A syntactic introduction to intersection types. Unpublished note, 2012.
- 38 O. L. Madsen and B. Moller-Pedersen. Virtual classes: a powerful mechanism in object-oriented programming. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 1989.
- 39 Microsoft. Typescript. <https://www.typescriptlang.org/>, 2012.
- 40 Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.
- 41 James Hiram Morris Jr. *Lambda-calculus models of programming languages*. PhD thesis, Massachusetts Institute of Technology, 1969.
- 42 Nathaniel Nystrom, Stephen Chong, and Andrew C. Myers. Scalable extensibility via nested inheritance. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2004.
- 43 Nathaniel Nystrom, Xin Qi, and Andrew C. Myers. J&: Nested Intersection for Scalable Software Composition. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2006.
- 44 Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An overview of the scala programming language. Technical report, EPFL, 2004.
- 45 Bruno C. d. S. Oliveira and William R. Cook. Extensibility for the masses. In *European Conference on Object-Oriented Programming*, 2012.

- 46 Bruno C. d. S. Oliveira, Zhiyuan Shi, and João Alpuim. Disjoint intersection types. In *International Conference on Functional Programming*, 2016.
- 47 Benjamin C Pierce. A decision procedure for the subtype relation on intersection types with bounded variables. Technical report, Carnegie Mellon University, 1989.
- 48 Gordon Plotkin. *Lambda-definability and logical relations*. Edinburgh University, 1973.
- 49 Garrel Pottinger. A type assignment for the strongly normalizable λ -terms. *To HB Curry: essays on combinatory logic, lambda calculus and formalism*, 1980.
- 50 Xin Qi and Andrew C. Myers. Sharing classes between families. In *Conference on Programming Language Design and Implementation*, 2009.
- 51 Redhat. Ceylon. <https://ceylon-lang.org/>, 2011.
- 52 Jakob Rehof and Paweł Urzyczyn. Finite combinatory logic with intersection types. In *International Conference on Typed Lambda Calculi and Applications*, 2011.
- 53 John C. Reynolds. Types, abstraction and parametric polymorphism. In *IFIP*, 1983.
- 54 John C Reynolds. Preliminary design of the programming language forsythe. Technical report, Carnegie Mellon University, 1988.
- 55 John C. Reynolds. The coherence of languages with intersection types. In *Lecture Notes in Computer Science (LNCS)*, pages 675–700. Springer Berlin Heidelberg, 1991.
- 56 Chieri Saito and Atsushi Igarashi. Matching *ThisType* to subtyping. In *Symposium on Applied Computing (SAC)*, 2009.
- 57 Chieri Saito, Atsushi Igarashi, and Mirko Viroli. Lightweight family polymorphism. *Journal of Functional Programming*, 18(03), 2007.
- 58 Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P Black. Traits: Composable units of behaviour. In *European Conference on Object-Oriented Programming*, 2003.
- 59 Jan Schwinghammer. Coherence of subsumption for monadic types. *Journal of Functional Programming*, 19(02):157, 2008.
- 60 Jeremy Siek, Peter Thiemann, and Philip Wadler. Blame and coercion: together again for the first time. In *Conference on Programming Language Design and Implementation*, 2015.
- 61 Richard Statman. Logical relations and the typed λ -calculus. *Information and Control*, 65(2-3):85–97, 1985.
- 62 Rick Statman. A finite model property for intersection types. *Electronic Proceedings in Theoretical Computer Science*, 177:1–9, 2015.
- 63 W. W. Tait. Intensional interpretations of functionals of finite type i. *The Journal of symbolic logic*, 32(2):198–212, 1967.
- 64 Philip Wadler. The expression problem. *Java-genericity mailing list*, 1998.
- 65 Yanlin Wang and Bruno C d S Oliveira. The expression problem, trivially! In *Proceedings of the 15th International Conference on Modularity*, 2016.
- 66 Yizhou Zhang and Andrew C. Myers. Familia: unifying interfaces, type classes, and family polymorphism. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2017.

A Some Definitions

► **Definition 32** (Type translation).

$$\begin{aligned}
 |\text{Nat}| &= \text{Nat} \\
 |\top| &= \langle \rangle \\
 |A \rightarrow B| &= |A| \rightarrow |B| \\
 |A \& B| &= |A| \times |B| \\
 |\{l : A\}| &= \{l : |A|\}
 \end{aligned}$$

► **Example 33** (Derivation of other direction of distribution).

$$\frac{\frac{A_1 <: A_1 \quad A_2 \& A_3 <: A_2}{A_1 \rightarrow A_2 \& A_3 <: A_1 \rightarrow A_2} \text{S-ARR} \quad \frac{A_1 <: A_1 \quad A_2 \& A_3 <: A_3}{A_1 \rightarrow A_2 \& A_3 \rightarrow A_1 \rightarrow A_3} \text{S-ARR}}{A_1 \rightarrow A_2 \& A_3 <: (A_1 \rightarrow A_2) \& (A_1 \rightarrow A_3)} \text{S-AND}$$

► **Example 34** (Derivation of contravariant distribution).

$$\frac{\frac{}{(A_1 \rightarrow A_2) \& (A_3 \rightarrow A_2) <: A_1 \rightarrow A_2} \text{S-ANDL} \quad \frac{A_1 \& A_3 <: A_1 \quad A_2 <: A_2}{A_1 \rightarrow A_2 <: A_1 \& A_3 \rightarrow A_2} \text{S-ARR}}{(A_1 \rightarrow A_2) \& (A_3 \rightarrow A_2) <: A_1 \& A_3 \rightarrow A_2} \text{S-TRANS}$$

B Full Type System of NeColus

$$\boxed{A <: B \rightsquigarrow c}$$

(Declarative subtyping)

$$\begin{array}{c}
 \begin{array}{ccc}
 \text{S-REFL} & \text{S-TRANS} & \text{S-TOP} \\
 \frac{}{A <: A \rightsquigarrow \text{id}} & \frac{A_2 <: A_3 \rightsquigarrow c_1 \quad A_1 <: A_2 \rightsquigarrow c_2}{A_1 <: A_3 \rightsquigarrow c_1 \circ c_2} & \frac{}{A <: \top \rightsquigarrow \text{top}}
 \end{array} \\
 \\
 \begin{array}{cc}
 \text{S-RCD} & \text{S-ARR} \\
 \frac{A <: B \rightsquigarrow c}{\{l : A\} <: \{l : B\} \rightsquigarrow \{l : c\}} & \frac{B_1 <: A_1 \rightsquigarrow c_1 \quad A_2 <: B_2 \rightsquigarrow c_2}{A_1 \rightarrow A_2 <: B_1 \rightarrow B_2 \rightsquigarrow c_1 \rightarrow c_2}
 \end{array} \\
 \\
 \begin{array}{ccc}
 \text{S-ANDL} & \text{S-ANDR} & \text{S-AND} \\
 \frac{}{A_1 \& A_2 <: A_1 \rightsquigarrow \pi_1} & \frac{}{A_1 \& A_2 <: A_2 \rightsquigarrow \pi_2} & \frac{A_1 <: A_2 \rightsquigarrow c_1 \quad A_1 <: A_3 \rightsquigarrow c_2}{A_1 <: A_2 \& A_3 \rightsquigarrow \langle c_1, c_2 \rangle}
 \end{array} \\
 \\
 \text{S-DISTARR} \\
 \frac{}{(A_1 \rightarrow A_2) \& (A_1 \rightarrow A_3) <: A_1 \rightarrow A_2 \& A_3 \rightsquigarrow \text{dist}_{\rightarrow}} \\
 \\
 \begin{array}{cc}
 \text{S-DISTRCD} & \text{S-TOPARR} \\
 \frac{}{\{l : A\} \& \{l : B\} <: \{l : A \& B\} \rightsquigarrow \text{dist}_{\{l\}}} & \frac{}{\top <: \top \rightarrow \top \rightsquigarrow \text{top}_{\rightarrow}}
 \end{array} \\
 \\
 \text{S-TOPRCD} \\
 \frac{}{\top <: \{l : \top\} \rightsquigarrow \text{top}_{\{l\}}}
 \end{array}$$

22:30 The Essence of Nested Composition

$\boxed{\Gamma \vdash E \Rightarrow A \rightsquigarrow e}$ (Inference)

$\text{T-TOP} \quad \frac{}{\Gamma \vdash \top \Rightarrow \top \rightsquigarrow \langle \rangle}$	$\text{T-LIT} \quad \frac{}{\Gamma \vdash i \Rightarrow \text{Nat} \rightsquigarrow i}$	$\text{T-VAR} \quad \frac{x : A \in \Gamma}{\Gamma \vdash x \Rightarrow A \rightsquigarrow x}$
$\text{T-APP} \quad \frac{\Gamma \vdash E_1 \Rightarrow A_1 \rightarrow A_2 \rightsquigarrow e_1 \quad \Gamma \vdash E_2 \Leftarrow A_1 \rightsquigarrow e_2}{\Gamma \vdash E_1 E_2 \Rightarrow A_2 \rightsquigarrow e_1 e_2}$	$\text{T-ANNO} \quad \frac{\Gamma \vdash E \Leftarrow A \rightsquigarrow e}{\Gamma \vdash E : A \Rightarrow A \rightsquigarrow e}$	$\text{T-MERGE} \quad \frac{\Gamma \vdash E_1 \Rightarrow A_1 \rightsquigarrow e_1 \quad \Gamma \vdash E_2 \Rightarrow A_2 \rightsquigarrow e_2 \quad A_1 * A_2}{\Gamma \vdash E_1, , E_2 \Rightarrow A_1 \& A_2 \rightsquigarrow \langle e_1, e_2 \rangle}$
$\text{T-RCD} \quad \frac{\Gamma \vdash E \Rightarrow A \rightsquigarrow e}{\Gamma \vdash \{l = E\} \Rightarrow \{l : A\} \rightsquigarrow \{l = e\}}$	$\text{T-PROJ} \quad \frac{\Gamma \vdash E \Rightarrow \{l : A\} \rightsquigarrow e}{\Gamma \vdash E.l \Rightarrow A \rightsquigarrow e.l}$	

$\boxed{\Gamma \vdash E \Leftarrow A \rightsquigarrow e}$ (Checking)

$\text{T-ABS} \quad \frac{\Gamma, x : A \vdash E \Leftarrow B \rightsquigarrow e}{\Gamma \vdash \lambda x. E \Leftarrow A \rightarrow B \rightsquigarrow \lambda x. e}$	$\text{T-SUB} \quad \frac{\Gamma \vdash E \Rightarrow B \rightsquigarrow e \quad B < : A \rightsquigarrow c}{\Gamma \vdash E \Leftarrow A \rightsquigarrow c e}$
---	--

$\boxed{A * B}$ (Disjointness)

$\text{D-TOPL} \quad \frac{}{\top * A}$	$\text{D-TOPR} \quad \frac{}{A * \top}$	$\text{D-ARR} \quad \frac{A_2 * B_2}{A_1 \rightarrow A_2 * B_1 \rightarrow B_2}$	$\text{D-ANDL} \quad \frac{A_1 * B \quad A_2 * B}{A_1 \& A_2 * B}$
$\text{D-ANDR} \quad \frac{A * B_1 \quad A * B_2}{A * B_1 \& B_2}$	$\text{D-RCDEQ} \quad \frac{A * B}{\{l : A\} * \{l : B\}}$	$\text{D-RCDNEQ} \quad \frac{l_1 \neq l_2}{\{l_1 : A\} * \{l_2 : B\}}$	$\text{D-AXNATARR} \quad \frac{}{\text{Nat} * A_1 \rightarrow A_2}$
$\text{D-AXARRNAT} \quad \frac{}{A_1 \rightarrow A_2 * \text{Nat}}$	$\text{D-AXNATRCD} \quad \frac{}{\text{Nat} * \{l : A\}}$	$\text{D-AXRCDNAT} \quad \frac{}{\{l : A\} * \text{Nat}}$	$\text{D-AXARRRCD} \quad \frac{}{A_1 \rightarrow A_2 * \{l : A\}}$
$\text{D-AXRCDARR} \quad \frac{}{\{l : A\} * A_1 \rightarrow A_2}$			

$\boxed{\mathcal{C} : (\Gamma \Rightarrow A) \mapsto (\Gamma' \Rightarrow B) \rightsquigarrow \mathcal{D}}$ (Context typing I)

$\text{CTYP-EMPTY1} \quad \frac{}{[\cdot] : (\Gamma \Rightarrow A) \mapsto (\Gamma \Rightarrow A) \rightsquigarrow [\cdot]}$	$\text{CTYP-APPL1} \quad \frac{\mathcal{C} : (\Gamma \Rightarrow A) \mapsto (\Gamma' \Rightarrow A_1 \rightarrow A_2) \rightsquigarrow \mathcal{D} \quad \Gamma' \vdash E_2 \Leftarrow A_1 \rightsquigarrow e}{\mathcal{C} E_2 : (\Gamma \Rightarrow A) \mapsto (\Gamma' \Rightarrow A_2) \rightsquigarrow \mathcal{D} e}$
$\text{CTYP-APPR1} \quad \frac{\Gamma' \vdash E_1 \Rightarrow A_1 \rightarrow A_2 \rightsquigarrow e \quad \mathcal{C} : (\Gamma \Rightarrow A) \mapsto (\Gamma' \Leftarrow A_1) \rightsquigarrow \mathcal{D}}{E_1 \mathcal{C} : (\Gamma \Rightarrow A) \mapsto (\Gamma' \Rightarrow A_2) \rightsquigarrow e \mathcal{D}}$	$\text{CTYP-MERGE1} \quad \frac{\mathcal{C} : (\Gamma \Rightarrow A) \mapsto (\Gamma' \Rightarrow A_1) \rightsquigarrow \mathcal{D} \quad \Gamma' \vdash E_2 \Rightarrow A_2 \rightsquigarrow e \quad A_1 * A_2}{\mathcal{C}, , E_2 : (\Gamma \Rightarrow A) \mapsto (\Gamma' \Rightarrow A_1 \& A_2) \rightsquigarrow \langle \mathcal{D}, e \rangle}$

$$\text{CTYP-MERGER1} \quad \frac{\Gamma' \vdash E_1 \Rightarrow A_1 \rightsquigarrow e \quad \mathcal{C} : (\Gamma \Rightarrow A) \mapsto (\Gamma' \Rightarrow A_2) \rightsquigarrow \mathcal{D} \quad A_1 * A_2}{E_1, \mathcal{C} : (\Gamma \Rightarrow A) \mapsto (\Gamma' \Rightarrow A_1 \& A_2) \rightsquigarrow \langle e, \mathcal{D} \rangle}$$

$$\text{CTYP-RCD1} \quad \frac{\mathcal{C} : (\Gamma \Rightarrow A) \mapsto (\Gamma' \Rightarrow B) \rightsquigarrow \mathcal{D}}{\{l = \mathcal{C}\} : (\Gamma \Rightarrow A) \mapsto (\Gamma' \Rightarrow \{l : B\}) \rightsquigarrow \{l = \mathcal{D}\}}$$

$$\text{CTYP-PROJ1} \quad \frac{\mathcal{C} : (\Gamma \Rightarrow A) \mapsto (\Gamma' \Rightarrow \{l : B\}) \rightsquigarrow \mathcal{D}}{\mathcal{C}.l : (\Gamma \Rightarrow A) \mapsto (\Gamma' \Rightarrow B) \rightsquigarrow \mathcal{D}.l}$$

$$\text{CTYP-ANNO1} \quad \frac{\mathcal{C} : (\Gamma \Rightarrow B) \mapsto (\Gamma' \Leftarrow A) \rightsquigarrow \mathcal{D}}{\mathcal{C} : A : (\Gamma \Rightarrow B) \mapsto (\Gamma' \Rightarrow A) \rightsquigarrow \mathcal{D}}$$

$$\boxed{\mathcal{C} : (\Gamma \Leftarrow A) \mapsto (\Gamma' \Leftarrow B) \rightsquigarrow \mathcal{D}}$$

(Context typing II)

$$\text{CTYP-EMPTY2} \quad \frac{[\cdot] : (\Gamma \Leftarrow A) \mapsto (\Gamma \Leftarrow A) \rightsquigarrow [\cdot]}{\lambda x. \mathcal{C} : (\Gamma \Leftarrow A) \mapsto (\Gamma' \Leftarrow A_1 \rightarrow A_2) \rightsquigarrow \lambda x. \mathcal{D}}$$

$$\text{CTYP-ABS2} \quad \frac{\mathcal{C} : (\Gamma \Leftarrow A) \mapsto (\Gamma', x : A_1 \Leftarrow A_2) \rightsquigarrow \mathcal{D} \quad x \notin \Gamma'}{\lambda x. \mathcal{C} : (\Gamma \Leftarrow A) \mapsto (\Gamma' \Leftarrow A_1 \rightarrow A_2) \rightsquigarrow \lambda x. \mathcal{D}}$$

$$\boxed{\mathcal{C} : (\Gamma \Leftarrow A) \mapsto (\Gamma' \Rightarrow B) \rightsquigarrow \mathcal{D}}$$

(Context typing III)

$$\text{CTYP-APPL2} \quad \frac{\mathcal{C} : (\Gamma \Leftarrow A) \mapsto (\Gamma' \Rightarrow A_1 \rightarrow A_2) \rightsquigarrow \mathcal{D} \quad \Gamma' \vdash E_2 \Leftarrow A_1 \rightsquigarrow e}{\mathcal{C} E_2 : (\Gamma \Leftarrow A) \mapsto (\Gamma' \Rightarrow A_2) \rightsquigarrow \mathcal{D} e}$$

$$\text{CTYP-APPR2} \quad \frac{\Gamma' \vdash E_1 \Rightarrow A_1 \rightarrow A_2 \rightsquigarrow e \quad \mathcal{C} : (\Gamma \Leftarrow A) \mapsto (\Gamma' \Leftarrow A_1) \rightsquigarrow \mathcal{D}}{E_1 \mathcal{C} : (\Gamma \Leftarrow A) \mapsto (\Gamma' \Rightarrow A_2) \rightsquigarrow e \mathcal{D}}$$

$$\text{CTYP-MERGERL2} \quad \frac{\mathcal{C} : (\Gamma \Leftarrow A) \mapsto (\Gamma' \Rightarrow A_1) \rightsquigarrow \mathcal{D} \quad \Gamma' \vdash E_2 \Rightarrow A_2 \rightsquigarrow e \quad A_1 * A_2}{\mathcal{C}, E_2 : (\Gamma \Leftarrow A) \mapsto (\Gamma' \Rightarrow A_1 \& A_2) \rightsquigarrow \langle \mathcal{D}, e \rangle}$$

$$\text{CTYP-MERGER2} \quad \frac{\Gamma' \vdash E_1 \Rightarrow A_1 \rightsquigarrow e \quad \mathcal{C} : (\Gamma \Leftarrow A) \mapsto (\Gamma' \Rightarrow A_2) \rightsquigarrow \mathcal{D} \quad A_1 * A_2}{E_1, \mathcal{C} : (\Gamma \Leftarrow A) \mapsto (\Gamma' \Rightarrow A_1 \& A_2) \rightsquigarrow \langle e, \mathcal{D} \rangle}$$

$$\text{CTYP-RCD2} \quad \frac{\mathcal{C} : (\Gamma \Leftarrow A) \mapsto (\Gamma' \Rightarrow B) \rightsquigarrow \mathcal{D}}{\{l = \mathcal{C}\} : (\Gamma \Leftarrow A) \mapsto (\Gamma' \Rightarrow \{l : B\}) \rightsquigarrow \{l = \mathcal{D}\}}$$

$$\text{CTYP-PROJ2} \quad \frac{\mathcal{C} : (\Gamma \Leftarrow A) \mapsto (\Gamma' \Rightarrow \{l : B\}) \rightsquigarrow \mathcal{D}}{\mathcal{C}.l : (\Gamma \Leftarrow A) \mapsto (\Gamma' \Rightarrow B) \rightsquigarrow \mathcal{D}.l}$$

$$\text{CTYP-ANNO2} \quad \frac{\mathcal{C} : (\Gamma \Leftarrow B) \mapsto (\Gamma' \Leftarrow A) \rightsquigarrow \mathcal{D}}{\mathcal{C} : A : (\Gamma \Leftarrow B) \mapsto (\Gamma' \Rightarrow A) \rightsquigarrow \mathcal{D}}$$

$$\boxed{\mathcal{C} : (\Gamma \Rightarrow A) \mapsto (\Gamma' \Leftarrow B) \rightsquigarrow \mathcal{D}}$$

(Context typing IV)

$$\begin{array}{c} \text{CTYP-ABS1} \\ \mathcal{C} : (\Gamma \Rightarrow A) \mapsto (\Gamma', x : A_1 \Leftarrow A_2) \rightsquigarrow \mathcal{D} \\ x \notin \Gamma' \\ \hline \lambda x. \mathcal{C} : (\Gamma \Rightarrow A) \mapsto (\Gamma' \Leftarrow A_1 \rightarrow A_2) \rightsquigarrow \lambda x. \mathcal{D} \end{array}$$

$$\boxed{\mathcal{L} \vdash A \prec : B \rightsquigarrow c}$$

(Algorithmic subtyping)

$$\begin{array}{c} \text{A-AND} \\ \mathcal{L} \vdash A \prec : B_1 \rightsquigarrow c_1 \quad \mathcal{L} \vdash A \prec : B_2 \rightsquigarrow c_2 \\ \hline \mathcal{L} \vdash A \prec : B_1 \& B_2 \rightsquigarrow \llbracket \mathcal{L} \rrbracket_{\&} \circ \langle c_1, c_2 \rangle \end{array}$$

$$\begin{array}{c} \text{A-ARR} \\ \mathcal{L}, B_1 \vdash A \prec : B_2 \rightsquigarrow c \\ \hline \mathcal{L} \vdash A \prec : B_1 \rightarrow B_2 \rightsquigarrow c \end{array}$$

$$\begin{array}{c} \text{A-RCD} \\ \mathcal{L}, \{l\} \vdash A \prec : B \rightsquigarrow c \\ \hline \mathcal{L} \vdash A \prec : \{l : B\} \rightsquigarrow c \end{array}$$

$$\begin{array}{c} \text{A-TOP} \\ \hline \mathcal{L} \vdash A \prec : \top \rightsquigarrow \llbracket \mathcal{L} \rrbracket_{\top} \circ \text{top} \end{array}$$

$$\begin{array}{c} \text{A-ARRNAT} \\ \llbracket \cdot \rrbracket \vdash A \prec : A_1 \rightsquigarrow c_1 \quad \mathcal{L} \vdash A_2 \prec : \text{Nat} \rightsquigarrow c_2 \\ \hline A, \mathcal{L} \vdash A_1 \rightarrow A_2 \prec : \text{Nat} \rightsquigarrow c_1 \rightarrow c_2 \end{array}$$

$$\begin{array}{c} \text{A-RCDNAT} \\ \mathcal{L} \vdash A \prec : \text{Nat} \rightsquigarrow c \\ \hline \{l\}, \mathcal{L} \vdash \{l : A\} \prec : \text{Nat} \rightsquigarrow \{l : c\} \end{array}$$

$$\begin{array}{c} \text{A-NAT} \\ \hline \llbracket \cdot \rrbracket \vdash \text{Nat} \prec : \text{Nat} \rightsquigarrow \text{id} \end{array}$$

$$\begin{array}{c} \text{A-ANDN1} \\ \mathcal{L} \vdash A_1 \prec : \text{Nat} \rightsquigarrow c \\ \hline \mathcal{L} \vdash A_1 \& A_2 \prec : \text{Nat} \rightsquigarrow c \circ \pi_1 \end{array}$$

$$\begin{array}{c} \text{A-ANDN2} \\ \mathcal{L} \vdash A_2 \prec : \text{Nat} \rightsquigarrow c \\ \hline \mathcal{L} \vdash A_1 \& A_2 \prec : \text{Nat} \rightsquigarrow c \circ \pi_2 \end{array}$$

C Full Type System of λ_c

$$\boxed{\Delta \vdash e : \tau}$$

(Target typing)

TYP-UNIT

$$\frac{}{\Delta \vdash \langle \rangle : \langle \rangle}$$

TYP-LIT

$$\frac{}{\Delta \vdash i : \text{Nat}}$$

TYP-VAR

$$\frac{x : \tau \in \Delta}{\Delta \vdash x : \tau}$$

TYP-ABS

$$\frac{\Delta, x : \tau_1 \vdash e : \tau_2}{\Delta \vdash \lambda x. e : \tau_1 \rightarrow \tau_2}$$

TYP-APP

$$\frac{\Delta \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Delta \vdash e_2 : \tau_1}{\Delta \vdash e_1 e_2 : \tau_2}$$

TYP-PAIR

$$\frac{\Delta \vdash e_1 : \tau_1 \quad \Delta \vdash e_2 : \tau_2}{\Delta \vdash \langle e_1, e_2 \rangle : \tau_1 \times \tau_2}$$

TYP-CAPP

$$\frac{\Delta \vdash e : \tau \quad c \vdash \tau \triangleright \tau'}{\Delta \vdash ce : \tau'}$$

TYP-RCD

$$\frac{\Delta \vdash e : \tau}{\Delta \vdash \{l = e\} : \{l : \tau\}}$$

TYP-PROJ

$$\frac{\Delta \vdash e : \{l : \tau\}}{\Delta \vdash e.l : \tau}$$

$$\boxed{c \vdash \tau_1 \triangleright \tau_2}$$

(Coercion typing)

COTYP-REFL

$$\frac{}{\text{id} \vdash \tau \triangleright \tau}$$

COTYP-TRANS

$$\frac{c_1 \vdash \tau_2 \triangleright \tau_3 \quad c_2 \vdash \tau_1 \triangleright \tau_2}{c_1 \circ c_2 \vdash \tau_1 \triangleright \tau_3}$$

COTYP-TOP

$$\frac{}{\text{top} \vdash \tau \triangleright \langle \rangle}$$

COTYP-TOPARR

$$\frac{}{\text{top}_{\rightarrow} \vdash \langle \rangle \triangleright \langle \rangle \rightarrow \langle \rangle}$$

COTYP-TOPRCD

$$\frac{}{\text{top}_{\{l\}} \vdash \langle \rangle \triangleright \{l : \langle \rangle\}}$$

COTYP-ARR

$$\frac{c_1 \vdash \tau'_1 \triangleright \tau_1 \quad c_2 \vdash \tau_2 \triangleright \tau'_2}{c_1 \rightarrow c_2 \vdash \tau_1 \rightarrow \tau_2 \triangleright \tau'_1 \rightarrow \tau'_2}$$

COTYP-PAIR

$$\frac{c_1 \vdash \tau_1 \triangleright \tau_2 \quad c_2 \vdash \tau_1 \triangleright \tau_3}{\langle c_1, c_2 \rangle \vdash \tau_1 \triangleright \tau_2 \times \tau_3}$$

$$\begin{array}{c}
\text{COTYP-PROJL} \\
\hline
\pi_1 \vdash \tau_1 \times \tau_2 \triangleright \tau_1
\end{array}
\qquad
\begin{array}{c}
\text{COTYP-PROJR} \\
\hline
\pi_2 \vdash \tau_1 \times \tau_2 \triangleright \tau_2
\end{array}
\qquad
\begin{array}{c}
\text{COTYP-RCD} \\
c \vdash \tau_1 \triangleright \tau_2 \\
\hline
\{l : c\} \vdash \{l : \tau_1\} \triangleright \{l : \tau_2\}
\end{array}$$

$$\begin{array}{c}
\text{COTYP-DISTRCD} \\
\hline
\text{dist}_{\{l\}} \vdash \{l : \tau_1\} \times \{l : \tau_2\} \triangleright \{l : \tau_1 \times \tau_2\}
\end{array}$$

$$\begin{array}{c}
\text{COTYP-DISTARR} \\
\hline
\text{dist}_{\rightarrow} \vdash (\tau_1 \rightarrow \tau_2) \times (\tau_1 \rightarrow \tau_3) \triangleright \tau_1 \rightarrow \tau_2 \times \tau_3
\end{array}$$

$$e \longrightarrow e'$$

(Small-step reduction)

$$\begin{array}{c}
\text{STEP-ID} \\
\hline
\text{id } v \longrightarrow v
\end{array}
\qquad
\begin{array}{c}
\text{STEP-TRANS} \\
\hline
(c_1 \circ c_2) v \longrightarrow c_1 (c_2 v)
\end{array}
\qquad
\begin{array}{c}
\text{STEP-TOP} \\
\hline
\text{top } v \longrightarrow \langle \rangle
\end{array}
\qquad
\begin{array}{c}
\text{STEP-TOPARR} \\
\hline
(\text{top}_{\rightarrow} \langle \rangle) \langle \rangle \longrightarrow \langle \rangle
\end{array}$$

$$\begin{array}{c}
\text{STEP-TOPRCD} \\
\hline
\text{top}_{\{l\}} \langle \rangle \longrightarrow \{l = \langle \rangle\}
\end{array}
\qquad
\begin{array}{c}
\text{STEP-ARR} \\
\hline
((c_1 \rightarrow c_2) v_1) v_2 \longrightarrow c_2 (v_1 (c_1 v_2))
\end{array}$$

$$\begin{array}{c}
\text{STEP-PAIR} \\
\hline
\langle c_1, c_2 \rangle v \longrightarrow \langle c_1 v, c_2 v \rangle
\end{array}
\qquad
\begin{array}{c}
\text{STEP-DISTARR} \\
\hline
(\text{dist}_{\rightarrow} \langle v_1, v_2 \rangle) v_3 \longrightarrow \langle v_1 v_3, v_2 v_3 \rangle
\end{array}$$

$$\begin{array}{c}
\text{STEP-DISTRCD} \\
\hline
\text{dist}_{\{l\}} \langle \{l = v_1\}, \{l = v_2\} \rangle \longrightarrow \{l = \langle v_1, v_2 \rangle\}
\end{array}
\qquad
\begin{array}{c}
\text{STEP-PROJL} \\
\hline
\pi_1 \langle v_1, v_2 \rangle \longrightarrow v_1
\end{array}
\qquad
\begin{array}{c}
\text{STEP-PROJR} \\
\hline
\pi_2 \langle v_1, v_2 \rangle \longrightarrow v_2
\end{array}$$

$$\begin{array}{c}
\text{STEP-CRCD} \\
\hline
\{l : c\} \{l = v\} \longrightarrow \{l = c v\}
\end{array}
\qquad
\begin{array}{c}
\text{STEP-BETA} \\
\hline
(\lambda x. e) v \longrightarrow e[x \mapsto v]
\end{array}
\qquad
\begin{array}{c}
\text{STEP-PROJRCD} \\
\hline
\{l = v\}.l \longrightarrow v
\end{array}$$

$$\begin{array}{c}
\text{STEP-APP1} \\
\hline
e_1 \longrightarrow e'_1 \\
\hline
e_1 e_2 \longrightarrow e'_1 e_2
\end{array}
\qquad
\begin{array}{c}
\text{STEP-APP2} \\
\hline
e_2 \longrightarrow e'_2 \\
\hline
v_1 e_2 \longrightarrow v_1 e'_2
\end{array}
\qquad
\begin{array}{c}
\text{STEP-PAIR1} \\
\hline
e_1 \longrightarrow e'_1 \\
\hline
\langle e_1, e_2 \rangle \longrightarrow \langle e'_1, e_2 \rangle
\end{array}
\qquad
\begin{array}{c}
\text{STEP-PAIR2} \\
\hline
e_2 \longrightarrow e'_2 \\
\hline
\langle v_1, e_2 \rangle \longrightarrow \langle v_1, e'_2 \rangle
\end{array}$$

$$\begin{array}{c}
\text{STEP-CAPP} \\
\hline
e \longrightarrow e' \\
\hline
c e \longrightarrow c e'
\end{array}
\qquad
\begin{array}{c}
\text{STEP-RCD1} \\
\hline
e \longrightarrow e' \\
\hline
\{l = e\} \longrightarrow \{l = e'\}
\end{array}
\qquad
\begin{array}{c}
\text{STEP-RCD2} \\
\hline
e \longrightarrow e' \\
\hline
e.l \longrightarrow e'.l
\end{array}$$

Defensive Points-To Analysis: Effective Soundness via Laziness

Yannis Smaragdakis

Dept. of Informatics and Telecommunications, University of Athens, Greece
yannis@smaragd.org

George Kastrinis

Dept. of Informatics and Telecommunications, University of Athens, Greece
gkastrinis@di.uoa.gr

Abstract

We present a defensive may-point-to analysis approach, which offers soundness even in the presence of arbitrary opaque code: all non-empty points-to sets computed are guaranteed to be over-approximations of the sets of values arising at run time. A key design tenet of the analysis is laziness: the analysis computes points-to relationships only for variables or objects that are guaranteed to never escape into opaque code. This means that the analysis misses some valid inferences, yet it also never wastes work to compute sets of values that are not “complete”, i.e., that may be missing elements due to opaque code. Laziness enables great efficiency, allowing a highly precise points-to analysis (such as a 5-call-site-sensitive, flow-sensitive analysis).

Despite its conservative nature, our analysis yields sound, actionable results for a large subset of the program code, achieving (under worst-case assumptions) 34-74% of the program coverage of an unsound state-of-the-art analysis for real-world programs.

2012 ACM Subject Classification Software and its engineering → Compilers, Theory of computation → Program analysis, Software and its engineering → General programming languages

Keywords and phrases static analysis, soundness, defensive analysis

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2018.23

Acknowledgements We gratefully acknowledge funding by the European Research Council, grant 307334 (SPADE), a Facebook Research and Academic Relations award, and an Oracle Labs collaborative research grant. We thank Anders Møller for helpful presentation suggestions.

1 Introduction

Soundness is a coveted property of static analyses, to the extent that the term is often colloquially used as a synonym for “correctness”. For a may-analysis, soundness means that the analysis abstraction overapproximates all concrete executions. A sound value-flow or points-to analysis is one that computes, per program point or per variable, value sets that represent (at least) all values that could possibly arise at the respective point during any possible execution.

Full soundness is hard to achieve in practice due to code that cannot be analyzed (e.g., dynamically generated/loaded code, binary/native code) or dynamic language features (e.g., reflection, `eval`, `invokedynamic`). We collectively refer to such features as *opaque code*. For instance, the Java code below invokes an unknown method, identified by string `methodName`, over an object, `obj`.

```
Method m = obj.getClass().getMethod(methodName);
m.invoke(obj);
```



© Yannis Smaragdakis and George Kastrinis;

licensed under Creative Commons License CC-BY

32nd European Conference on Object-Oriented Programming (ECOOP 2018).

Editor: Todd Millstein; Article No. 23; pp. 23:1–23:28

Leibniz International Proceedings in Informatics



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

String `methodName` could be a true run-time value – e.g., read from a file or external resource. Object `obj` could itself be of a type not available during analysis – e.g., `obj` could be obtained through the network and statically typed using a vague interface or root-of-hierarchy type.

Faced with such complications, all past analyses that claim soundness have done so under *a priori* qualifications. Prominently, abstract-interpretation-based [8] approaches, such as Astrée [10], have long emphasized soundness. The conceptual form of such a soundness result is as follows:¹

An *Analysis* of programs in language *Lang* is sound relative to language subset *Lang'* and executions set *Exec'* iff:

$$\forall \text{ program } P \in \text{Lang}: P \in \text{Lang}' \wedge e \in \text{Exec}' \implies e \in \gamma(\text{Analysis}(P))$$

(where γ is the concretization function that maps abstractions in the output domain of *Analysis* to concrete executions in a universe *Exec*, superset of *Exec'*).

The problem with this formulation of soundness is that, although it yields provable theorems, the *a priori* qualification excludes virtually all realistic programs. The *Lang'* or *Exec'* of published proofs disqualify the vast majority of modern programs “in the wild”. Language subset *Lang'* will typically exclude all dynamic features (e.g., reflection) and/or executions subset *Exec'* will disqualify all behaviors that are deemed too-dynamic (e.g., invoking dynamically-loaded code). Reflection alone disqualifies $\sim 80\%$ of Java programs in the 461-program corpus of the recent Landman et al. study [16].

The above issues have led several members of the static analysis community to proclaim that “*all published whole-program analyses are unsound*” [21], i.e., their soundness guarantee does not apply to realistic programs, and similarly that “*[there is not] a single realistic whole-program analysis tool [...] that does not purposely make unsound choices*”. The problem is, therefore, both theoretical and practical. Soundness theorems do not give guarantees for realistic programs. Implementations of analyses in tools happily perpetuate the illusion: they handle soundly the language features one can prove theorems about, while cutting corners in the sound handling of all *other* features, in order to demonstrate greater scalability or precision. For instance, in our earlier Java code fragment, even if the type of `obj` is known, many implemented static analyses will not consider all its methods (which now form a small finite set) as possible values of `m`, but will instead ignore the code altogether. This phenomenon has led to the introduction of the term *soundy* [21] to characterize such analyses. Despite the derogatory tone, “soundy” analyses are the current *good* case of static analyses! They are realistic analyses that handle all “normal” language features soundly.

In this work, we propose *defensive analysis*: a static analysis architecture that addresses the above soundness shortcomings. The basis of defensive analysis can be seen as a different conceptual formulation of soundness.

An *Analysis* of program *P* in language *Lang* computes results, *Analysis(P)*, together with soundness marker sets, *Claim(P)*. The *Analysis* is sound iff:

$$\forall \text{ program } P \in \text{Lang}, \text{ execution } e: e[\text{Claim}(P)] \in \gamma(\text{Analysis}(P))[\text{Claim}(P)]$$

(where γ is as before, and $e[\text{Claim}(p)]$ is the restriction of an execution *e* to program points with soundness claims, and the definition is similarly lifted to sets of executions).

In other words, the analysis imposes no (or very liberal) *a priori* restrictions to its soundness claims, but instead *computes* the claimed domain of its soundness: the program parts for which the analysis result is sound. The soundness theorem applies to all (or most) programs,

¹ This formulation is due to Xavier Rival of the Astrée project (e.g., [25]).

under all execution conditions – instead of eagerly disqualifying the vast majority of real-world programs. The extent of soundness is now defined over program points and becomes an experimentally measurable quantity: the size of $Claim(P)$ (which we term the *coverage* of the analysis) can be measured to quantify for which percentage of a program’s points the analysis is guaranteed to produce sound results.

The challenge of defensive analysis is, thus, to distinguish parts of the program that are certain to not be affected by opaque code. Delineating “safe” from “unsafe” parts of the program is an ambitious goal, since opaque code can do virtually anything: it can add dynamically-generated subclasses with never-seen methods that get called (via dynamic dispatch or first-class functions) at unsuspecting program points; it can call any existing method or alter any field via reflection; it can interpose its own implementations at every place where the program uses a reflective lookup; worst of all, it can wreak havoc on all parts of the heap reachable from any reference that escapes into opaque code.

We designed and implemented a defensive may-point-to (henceforth just “points-to”) analysis for Java. The analysis follows the above form, explicitly designating points-to sets that are sound, i.e., that contain at least all the values that may ever arise in actual executions. Soundness guarantees carry over to the implementation: the soundness proof explicitly models all other language features as `<unknown>` instructions and makes only weak, semantically-justified assumptions (e.g., a type-safe heap) about them. Soundness reasoning is defensive in that it establishes when the analysis can be certain to know the full contents of a points-to set, no matter what opaque code can do (within the stated weak assumptions).

In our effort to implement defensive analysis in a realistic package, we found *laziness* to be an essential feature – the analysis cannot scale without it for real-world programs. Laziness means that the analysis does not compute points-to sets unless it can also claim their soundness. That is, program points outside of the $Claim(P)$ set do not get populated at any point – they remain empty throughout the analysis. Consequently, all points-to sets with a potentially unbounded number of objects (e.g., sets that depend on reflection or dynamic loading) are represented as the empty set: the analysis never computes any contents for them. An empty analysis result merely means “I don’t know”, which could signify that the points-to set is affected by opaque code, or simply that the analysis cannot establish that it is *not* affected by opaque code. Laziness yields high efficiency: the analysis can fall-back to an empty set (i.e., implicitly unbounded) without performing any computation or occupying space.

The defensive nature of the analysis combined with laziness result in a very simple specification. The analysis does not need to integrate complex escape or alias reasoning (i.e., “can this object *ever* escape into opaque code?”), but only best-effort logic (i.e., “here are simple, safe cases, when the object cannot possibly be affected by opaque code”). Failure to establish non-escaping merely means that the points-to set remains empty, to denote “I don’t know” or “potentially unbounded”.

Concretely, the work makes the following contributions:

- We offer a general static may-point-to analysis that yields sound results for realistic programs in the presence of opaque code.
- The analysis is efficient, leveraging its lazy representation of points-to sets. As a result, it can be made precise, beyond the limits of standard whole-program points-to analyses – e.g., for a 5-call-site-sensitive and flow-sensitive analysis. The analysis is also modular: it can be applied to any subset of the program, and will merely leave more points-to sets empty if other parts are unknown.
- We show that the analysis, though quite defensive, yields useful coverage. In measurements over large Java benchmarks, our analysis computes guaranteed over-approximate points-to

sets ($Claim(P)$) for 34-74% of the local variables of a conventional unsound analysis. (This number is much higher than that of a conventional sound but intra-procedural analysis.) Similar effectiveness is achieved for other metrics (e.g., number of calls de-virtualized), again with actionable, guaranteed-sound outcomes.

2 Analysis Illustration

We next describe the setting of defensive analysis and illustrate its principles and behavior.

2.1 Soundness and Design Decisions Overview

Defensive analysis is a may-point-to analysis based on access paths, i.e., expressions of the form “ $var(.fld)^*$ ”. That is, the analysis computes *the abstract objects* (i.e., allocation sites in the program text) *that an access path may point to*. The analysis is *flow-sensitive*, hence we will be computing separate points-to information per program point. Both of these design decisions are integral elements of the analysis, as we will justify in Section 2.2.

Soundness in this setting means that the analysis computes an over-approximation of any points-to set – i.e., the analysis computes (abstractions of) all objects that may occur in an actual execution. However, since not all allocation sites are statically known (due to dynamically loaded code), such an over-approximation cannot be explicit: not all possible values in a points-to set can be listed. Thus, there needs to be a special value, \top , to denote “unknown”, i.e., that the analysis cannot bound the contents of a points-to set.

Defensive analysis takes the above observation one step further, by employing a *lazy* approach: it never populates a points-to set if it cannot guarantee that it is bounded. Thus, an empty points-to set for an access path signifies that (as far as the analysis knows) the access path can point to anything.²

In other words, an empty set can be thought to represent a bottom (\perp) value during the analysis computation: it just marks a set as having no known values (yet). A set stops being empty only when all the possible ways (in known or unknown code) to contribute values to it have been examined and are found to have bounded contents. At the end of the analysis, all sets that have remained empty signify that the analysis could not bound their contents, i.e., they do not belong in the set $Claim(P)$ of program points with soundness claims. Therefore an empty set after termination of the analysis is conceptually equivalent to a top (\top) value: the set could contain anything. This is consistent with the defensive nature of the analysis: not knowing all the values of a set is considered just as bad as knowing it can point to anything.

With this representation choice, the analysis does not need to expend effort in order to be sound. All points-to sets (for any valid access path, of any length) start off empty, i.e., if the analysis were to stop at that point it would report them as having \top values, meaning “the set can contain anything”. This is a sound answer, and is only subsequently refined.

This lazy evaluation means that defensive analysis does not need to employ sophisticated mechanisms to simply be sound. For instance, instead of a precise over-approximate escape analysis, defensive analysis can use a simple analysis (including none at all) to compute straightforward cases when an object is guaranteed to never escape into opaque code.

² We use an explicit abstract value for `null`, therefore a points-to set that only contains `null` is not empty. This is standard in flow-sensitive analyses, anyway. (In flow-*ins*sensitive analyses, `null` is typically a member of every points-to set, so it is profitable to not represent it, and hence have an empty set mean a `null`-only reference. No such benefit would arise in our flow-sensitive setting.)

2.2 Background and Illustrating Design Decisions

We can see the rationale behind our design decisions through simple examples.

Baseline intra-procedural reasoning. It is easy for an analysis to be sound locally, in an *intra-procedural* setting. For instance, when a variable is freshly assigned with a newly allocated object, we are guaranteed to soundly know its points-to set:

```
x = new A(); // abstract object a1, x points-to set is {a1}
```

We can also propagate such information transitively through local assignments (a.k.a. “move” instructions), as long as no opaque code can interfere. In the case of local variables, standard concurrency models (for Java, C++, etc.) do not allow interference from other threads, hence points-to sets remain sound, as long as the code itself does not call out to opaque code:

```
x = new A(); // abstract object a1, x points-to set is {a1}
y = x; // y points-to set is {a1}
z = y; // z points-to set is {a1}
```

This approach is one often taken by traditional compilers (ahead-of-time or just-in-time alike) in order to perform intra-procedural optimizations, such as those based on traditional data-flow analysis. (Later, in our experimental evaluation, we compare against such a baseline “intra-procedural sound” analysis.)

However, the challenge is to also reason soundly about *inter-procedural* behavior. This includes reasoning about the heap (i.e., reading fields of objects) and about method calls and returns, whose resolution may be dynamic. This will be the focus of the defensive analysis specification.

Inter-procedural elements. The large potential for opaque code to affect inter-procedural analysis results has prevented past analyses from being sound. For instance, consider a simple heap load instruction:

```
x = y.fld;
```

Imagine that the analysis has (somehow) soundly computed all the objects that `y` may point to. It may also know all the places in the code where field `fld` is assigned and what is assigned to it. However, the analysis still cannot compute soundly the points-to set of `x` unless it also knows that all objects referenced by `y` can never escape to opaque code. This is hard to establish: not only do all sites of opaque code (reflection, unknown instructions, potential dynamic code generation sites, and more) need to be marked, but the analysis needs to know an over-approximation of which objects these sites can reach. This requires to have pre-computed an over-approximate (i.e., sound) points-to analysis, which is the problem we are trying to solve in the first place. Past work has dealt with this problem with unrealistic assumptions. For instance, Sreedhar et al. [33] present a call-specialization analysis that can handle dynamic class loading, but only if given the results of a sound may-point-to analysis as input.

Instead, defensive analysis pessimistically computes that a points-to set is \top (i.e., can contain anything) unless it is certain that its contents are bounded. When can the analysis know this, however? Such a guarantee of bounded contents typically comes from having precisely tracked the contents of a variable or field all the way from its last assignment, and having established that no other code could have interfered. For instance, let us expand our earlier example:

```

1 y.fld = new A(); // abstract object a1, y.fld points-to set is {a1}
2 ... // analyzable, non-interfering code
3 x = y.fld;

```

The analysis can now know that the points-to set of `x` is `{a1}`, i.e., the singleton containing the allocation site for `A` objects on line 1. For this to be true, the analysis has to establish that all code between the store instruction to `y.fld` and the subsequent load does not interfere with the value of `y.fld`. For example, we can be certain of such non-interference if the code does not contain a store to field `fld` of *any* object, does not call any methods, and no other thread can change the heap at that segment of the program. These are simple, local conditions that the analysis may well be able to establish.

In practice, our defensive analysis will do a lot more: it will track method calls, up to a maximum context depth, to ascertain when they can interfere with points-to sets. (If any interference is detected, the points-to set propagated forward is empty.) For instance, in the example code below, the analysis can know with certainty the points-to set of `x` on line 6, whenever method `foo` is called from line 3 of the program fragment.

```

1 y.fld = new A(); // abstract object a1, y.fld points-to set is {a1}
2 z.otherFld = new B();
3 foo(y);
4
5 void foo(W w) {
6   x = w.fld; // x-for-call-site-3 points-to set is {a1}
7 }

```

Note the elements that contribute to such reasoning: The result holds soundly only when `foo` is called from the specific call site. This result is established only by tracking the value of `y.fld` (renamed to `w.fld` inside method `foo`) instruction-by-instruction all the way to line 6. The heap store instruction on line 2 is guaranteed to not affect `y.fld` (regardless of whether `z` and `y` alias or not), since Java guarantees object isolation and the reference is to a different field. (More on language model assumptions in Section 2.3.)

The above example helps illustrate the design choices of defensive analysis: it is a flow-sensitive, context-sensitive analysis because it needs to track all points-to information that is guaranteed to hold, per-instruction, following closely all possible control-flow of the program, even across calls. It is also an analysis computing points-to information on *access paths* because this gives significantly more ability to reason about the heap locally. For instance, in the above program fragment, we may not know which objects `y` may point to.³ However, we do know that `y.fld` certainly points to abstract object `a1` after line 1!

Laziness. Finally, consider the design choice of representing unbounded points-to sets as empty, i.e., to lazily compute the contents of points-to sets only if they can be proven to be finite. Defensive analysis requires laziness for scalability. (Experimentally, a non-lazy analysis does not scale for any non-zero context depth, i.e., cannot be effective inter-procedurally.)

Laziness means skipping an explicit representation of \top , in favor of keeping points-to sets empty (\perp in the usual lattice of sets) as long as possible. (As mentioned earlier, at the end of the analysis, all sets that stayed \perp become implicitly \top .) This has the minor benefit of avoiding storage of \top values, since empty sets are represented without consuming memory. More majorly, however, it enables the analysis to give a convenient meaning to any finite

³ In fact, even if we did know, these would be abstract objects. Static analysis would almost never be able to establish soundly what their `fld` field points to, because this information needs to capture the `fld` values of all *concrete* objects (not just the latest one) represented by the same abstract object.

points-to sets that arise. Instead of “*this set currently has bounded contents, but may become \top during the course of the analysis*”, a non-empty set of values implies “*this set has bounded contents and is guaranteed to always have bounded contents*”. By making this distinction, the analysis never wastes effort computing points-to sets with explicit (non- \top) contents only to later discover that the points-to set is \top . For an example of how much wasted effort can be saved by being lazy, consider an example program involving a heap load and a virtual call:

```

1 y.fld = new A(); // abstract object a1
2 while (...) {
3   x = y.fld;
4   x.foo(y);
5 }
```

An analysis may have computed all the abstract objects that `y.fld` may point to at line 3. One of these computed objects may induce a different resolution of the call instruction (line 4), which can suddenly lead to the discovery that a `y.fld`-aliased object can enter opaque code (while this was not true based on what the analysis had computed earlier). Since the object referenced by `y.fld` can change in code that is not analyzed, the points-to set of `x` at the load instruction will need to be augmented with the implicit over-approximation special value, \top . This means that all previously computed values for the points-to sets of `x` and `y.fld` are subsumed by the single \top value. Computing these values and all others that depend on them constitutes wasted effort. To make matters worse, this is more likely to happen for *large* points-to sets, i.e., the more work the analysis has performed on computing an explicit points-to set, the larger (and less precise) the set will be, and the more likely it is that the work will be wasted because the set will revert to \top .

The design principle of “laziness in order to avoid wasted effort” is responsible for the scalability of defensive analysis. As we show in our experiments, defensive analysis scales to be flow-sensitive, 5-call-site sensitive over large Java benchmarks and the full JDK. (In standard past literature for all-program-points analyses, even a flow-*insensitive*, 2-call-site-sensitive analysis has been infeasible over these benchmarks [31].⁴)

2.3 Soundness Assumptions

The soundness claims of defensive analysis are predicated on assumptions about the environment. These assumptions reflect well the setting of safe languages, such as Java:

- **Object isolation.** Objects can only be accessed via high-level references. This means that objects and fields are isolated: an object can be referenced outside the dynamic scope of a method or by a different thread only if a reference to the object has escaped the method or current thread. (This restriction also implies that objects are not contained in one another, though they can contain references to each other.) A field can only be accessed via a base object pointer and a unique field signature.
- **Stack frame isolation.** Local variables are isolated from each other, thread-private and private to their allocating method. No external code can access the local variables of a method, even if the code is executed (i.e., is a callee) under the dynamic scope of the method.

⁴ It is worth emphasizing that, although defensive analysis is lazy, this is a very different form of laziness than that of *on-demand* points-to analysis (e.g., [32, 2]). An on-demand analysis only computes points-to information for program points that may affect a particular site of interest, instead of the entire program. The defensive analysis we describe is an all-program-points analysis: it computes points-to information for the entire program, i.e., for all possible points-to queries, including ones potentially devised in the future. Yet the analysis is lazy in that it only computes values for points-to sets that it can prove to have bounded contents.

- **Concurrency model.** In the simplified model of the paper, soundness is predicated on the assumption that standard mutexes (or operations on volatile variables) are used to protect all shared memory data. We later discuss how our implementation removes this assumption.⁵

Thus, our setting is clearly that of a safe language with near-unlimited potential for dynamic behavior. Notably, we can have unknown instructions; calls to native code with arbitrary behavior (over a well-typed, isolated heap); generation and loading of unknown code (which may also be called, via dynamic dispatch, by unsuspecting *known* code); arbitrary access to existing or unknown objects (both field read/writes and method calls) via reflection, i.e., without such access being identifiable in the program text; and more.

3 Defensive Analysis, Informally

The discussion of analysis principles in the previous sections gives the main tenets of defensive analysis. However, these need to be concretely applied over all complex language features affecting points-to information: control-flow merging, heap manipulation, and method calls. We give informal examples next. Following these examples should significantly facilitate understanding the formal specification of the analysis, in later sections.

Control-flow merging. Consider a branching example:

```

1  if (complexCondition()) {
2    x = new A(); // abstract object a1
3    // x points-to set is {a1}
4  } else {
5    x = notFullyAnalyzed();
6    // x points-to set is {}
7  } // x points-to set is {}

```

The first branch of the above `if` expression establishes that the points-to set of variable `x` is `{a1}`. For a conventional analysis, this would result in adding `a1` to the points-to set of `x` at the merge point (after line 7). The defensive analysis, however, has to be conservative and not compute values that may later become \top . Therefore, it will add `a1` to the final points-to set of `x` only if it can also prove that the points-to set of `x` in the second branch is bounded, i.e., non-empty. If the analysis is not certain of this, the points-to sets of `x`, both in the second branch and at the merge point, stay empty. Inability to bound the points to set of `x` in the second branch can be due a variety of reasons: e.g., there can be opaque code inside `notFullyAnalyzed`, or the analysis may reach its maximum context depth, so that the return value of the method is not tracked precisely.

Heap manipulation. Similar treatment applies to all cases of points-to sets (e.g., for complex access paths) when information is merged: the analysis yields a non-empty result only if it is certain that the result could not have been invalidated by any other code, available or not. For instance, consider the following example of heap store instructions:

⁵ The reason for the simplified concurrency model in the paper is that it allows presenting the analysis in its purest form, dealing with core language features such as heap loads/stores and calls, but unencumbered by auxiliary considerations (e.g., computing objects that do not escape into other threads).

```

1 x.fld = new A(); // abstract object a1
2 // x.fld points-to set is {a1}
3 y.fld = notFullyAnalyzed();
4 // x.fld points-to set is {}

```

After the first instruction, the points-to set of access path `x.fld` is computed to be `{a1}`. However, in most cases, the analysis will not be able to ascertain that `x` and `y` are not aliased. Therefore, after the second instruction, the points-to set of `x.fld` will be empty, i.e., unknown. This reflects well the defensive nature of the analysis: whenever uncertain, points-to sets will default to empty, i.e., undetermined.

Generally, since the analysis is flow-sensitive and access-path based, store instructions certain to operate on the same object perform *strong updates*, while store instructions that *possibly* operate on the same object perform *weak updates*:

```

1 x.fld = new A(); // abstract object a1
2 // x.fld points-to set is {a1}
3 x.fld = new B(); // abstract object b1
4 // x.fld points-to set is {b1}
5 y.fld = new B(); // abstract object b2
6 // x.fld points-to set is {b1,b2}

```

In this case, the points-to information of access path `x.fld` is set to `{b1}` after the second store instruction, ignoring the previous contents. (The example assumes that types `A` and `B` are both compatible with the static type of `x.fld`.) After the third store instruction, however, a new element is added to the points-to set – again, under the assumption that the analysis cannot determine whether `x` and `y` are aliased.

The different element in defensive analysis is that if any of the involved points-to sets is empty, both strong and weak updates yield an empty points-to set. For instance, replacing either of the last two allocations (`new B()`) above with a call to `opaque` (or not fully analyzed) code would make all subsequent points-to sets of `x.fld` be empty.

Method calls. Defensive analysis computes sound may-point-to information simultaneously with *sound call-graph* information. The analysis employs the same principles for the call-graph representation as for points-to: a finite set of method call targets means that the set is guaranteed bounded, while an empty set of method call targets means that the analysis cannot (yet) establish that all target methods are known.

To compute a sound over-approximation of method call targets, one needs a bounded may-point-to set for the receiver. Otherwise, the receiver object could be unknown – e.g., an instance of a dynamically loaded class – resulting in an unsound call-graph.

When the set of method call targets is not bounded, dynamic calls cannot be resolved and the analysis has to be conservative. For instance, in the example below, a conventional unsound analysis would resolve the virtual call `x.foo()` to, at least, the method `A::foo`, i.e., `foo` in class `A`.

```

1 if (complexCondition()) {
2   x = new A(); // abstract object a1
3 } else {
4   x = notFullyAnalyzed();
5 }
6 x.foo();

```

In contrast, recall that for a defensive analysis the points-to set of `x` at the point of the call to `foo` is empty. Accordingly, the defensive analysis does *not* resolve the virtual call at all: per the lazy evaluation principle, there is no point of computing what *one* target of the call will do, when other targets are unknown and full soundness (i.e., guaranteed

over-approximation) is required. This means that all heap information (i.e., all access-path points-to information, except for *trivial* access paths consisting of a single local variable and no fields) that held before the method call ceases to hold after it! (There are notable exceptions – e.g., for access paths with final fields, or for cases when an escape analysis can establish that some part of the heap does not escape into the called method. Section 5 discusses such intricacies.)

When method calls *can* be resolved, the target methods have to be analyzed under a context uniquely identifying the callee. A defensive analysis may know all methods that can get called at a certain point, but *it cannot know all callers of a method*. Consider the following example:

```

1 void caller() {
2   A x = new A(); // abstract object a1
3   callee(x); // call to callee
4 }
5
6 void callee(A y) {
7   ...
8 }

```

Assume that there is no other discernible call to `callee` anywhere in the program. An unsound analysis would establish that variable `y` in `callee` (i.e., immediately after line 6) points to abstract object `a1`. A defensive analysis, however, cannot do the same unconditionally. The points-to set of `y` without context information has to be the empty set! The reason is that there may be completely unknown callers of `callee` – e.g., in existing code, via reflection, or in dynamically loaded code. Such callers could pass different objects as arguments to `callee` and the analysis cannot upper-bound the set of such arguments. Thus, the only safe answer for a defensive analysis is “undetermined” – i.e., an empty set.

Thus, in order to propagate analysis results inter-procedurally, a defensive analysis has to leverage context information. In the above example, what the analysis will establish is that `y` points to `a1` *conditionally*, under context 3, signifying the call-site instruction (line 3 in our listing).

The above implies that the use of context in a defensive analysis is rather different than in a traditional unsound points-to analysis. Contexts in standard points-to analysis can be *summarizing*: a single context can merge arbitrary concrete (dynamic) executions, as long as any single concrete execution maps uniquely to a context. For instance, a 1-object-sensitive analysis [23] merges all calls to a method as long as they have the same abstract receiver object, independently of call sites.

Context in a conventional analysis only adds *precision*, relative to a context-insensitive analysis. In contrast, context in a defensive analysis is necessary for *correctness*: since information is collected per-program-point, propagating points-to sets from a call site to a callee can only be done under a context that identifies the call-site program point. Contexts cannot freely summarize multiple invocation instructions, because there may be others, yet unknown, invocations that would result in the same context.

Therefore, a context-sensitive defensive analysis has to be, at a minimum, *call-site sensitive* [27, 28]: the call site of an analyzed method has to be part of the context (as will, for deeper context, the call site of the caller, the call site of the caller’s caller, etc.). Other kinds of context (e.g., object-sensitive context [23, 30]) can be added for extra precision.

<i>Instruction</i>	<i>Operand Types</i>	<i>Description</i>
$i : v = \text{new } T ()$	$I \times V \times T$	Heap Allocations
$i : v = u$	$I \times V \times V$	Assignments
$i : v = u.f$	$I \times V \times V \times F$	Field Loads
$i : v.f = u$	$I \times V \times F \times V$	Field Stores
$i : v.\text{meth}()$	$I \times V \times M \times V^n$	Virtual Calls
$i : \text{return}$	I	Method Returns
$i : \langle \text{unknown} \rangle$	I	Unknown instruction (i.e., any other)

■ **Figure 1** Intermediate Representation instruction Set.

4 A Model of Defensive Analysis

We next present a rigorous model of our defensive analysis. The model uses a minimal intermediate language that captures the essence of the approach. The language can be straightforwardly enhanced with features such as arrays, static members and calls, exceptions, etc.

4.1 Preliminaries

Figure 1 shows the form of the input language. The domains of the analysis (and meta-variables used subsequently, plain or primed) comprise:

- $v, u \in V$, a set of variables,
- $T, S \in T$, a set of types,
- $f, g \in F$, a set of fields,
- $\text{meth} \in M$, a set of methods,
- $i, j \in I$, a set of instruction labels,
- $c, d \in C$, a set of contexts,
- $\widehat{o}, \widehat{o}_i \in O$, a set of abstract objects, potentially identifying their allocation instruction,
- $p \in P$, a set of access paths (i.e., the set $V(.F)^*$),
- $n \in \mathbb{N}$, the set of natural numbers.

The analysis input consists of a set of instructions, linked into a control-flow graph, via relation $i \xrightarrow{\text{next}} j$ (over $I \times I$). The input program is assumed to be in a single-return-per-method form. We employ type information as well as other symbol table information, accessed through some auxiliary functions and predicates:

- meth_T is the result of looking up method signature meth in type T .
- $\text{meth}(n)$ is the n -th instruction of method meth .
- We overload the \in operator to more than set membership, in unambiguous contexts, namely: $i \in \text{meth}$ (instruction is in method), $f \in p$ (field is in access path), $\widehat{o} \in T$ (abstract object is of type), $v \in T$ (variable is of type).
- $\text{arg}_n^{\text{meth}}$ and arg_n^i denote the n -th formal or actual arg of a method and invocation instruction, respectively. (By convention, the `this`/base variable of a method invocation is assumed to be the 0-th argument.)
- $p[v/u]$ is the access path resulting from changing the base of access path p from v to u (if applicable).

4.2 Analysis Structure

Figure 2 shows the analysis specification, in terms of constraints. Any solution satisfying these constraints has the desired soundness property and in Section 4.3 we discuss extra considerations so that the constraints can also be used to *compute* a solution. We recommend following the figure together with our text explaining the rules: although the rules are precise (transcribed from a mechanized logical specification) some are hard to follow without explanation of their intent beforehand.

The analysis constraints define the following relations:

- The “access path points-to” relation, in two varieties, before and after an instruction: $i : p \xrightarrow{\text{IN}}_c \hat{o}$ and $i : p \xrightarrow{\text{OUT}}_c \hat{o}$ (p may point to \hat{o} before/after instruction i executed under context c). This is our sound may-point-to relation: if, at the end of the analysis, the set of \hat{o} s for given i, p, c is not empty, it will be a superset of the abstract objects \hat{o} pointed by p at the given program point and context during any dynamic execution.⁶
- The “may-call” relation, i.e., our sound call-graph representation: $i \xrightarrow{\text{calls}}_c \text{meth}$ (instruction i executed under context c may call method meth and the resulting context will be c').
- The “reachable” relation, $\overline{\text{meth}}^c$, denoting that method meth is reachable under context c , and should, thus, be analyzed. This relation is partially populated when the analysis starts: it holds an initial set of methods, under the empty context **Init**, that should be analyzed.

Alloc, Move, Load, Store-1. The first four rules of the analysis are rather straightforward. The ALLOC rule is the only one with some minimal subtlety: if an object is freshly allocated, we know that the variable it is directly assigned to points to it. This inference is valid in any reachable context, even the initial, making-no-assumptions, **Init** context. Therefore this rule is responsible for kickstarting the analysis, producing the first points-to inferences (valid locally) that will then propagate.

Store-2. The STORE-2 rule is the first one exhibiting the defensive and lazy features of the analysis. The rule performs a “weak update” on points-to sets of possibly affected access paths, as long as they are guaranteed to be bounded, i.e., they are non-empty. At a store instruction, $u.f = v$, if an access path $w.f$ has a base explicitly different from u (with f being the same), then its points-to set is augmented with any element (\hat{o}) of the points-to set of v , while maintaining its original elements (\hat{o}'). This rule defensively adds more information to guarantee an over-approximation in the case of access paths that may be aliases for the same object. The subtlety of the rule lies in its handling of empty points-to sets. If *either* of the points-to sets (of v or of $w.f$) is empty before the instruction, the rule does not match, hence the points-to set of $w.f$ after the instruction does not acquire any contents. This is consistent with our sound handling: if the earlier contents or the update cannot be upper-bounded, then the resulting points-to set cannot be, either.

Note the contrast between rules STORE-1 and STORE-2. We do not need to determine precisely the aliasing relationship between base variables u and w . If there is a chance that the variables are aliased, it is safe to conservatively add more possible values to the points-to set of $w.f$. In the case of STORE-1, however, we could do better than the conservative treatment and perform a strong update.

⁶ To be precise, concrete objects arise during execution but we are considering their standard mapping to abstract objects, per allocation site.

$$\begin{array}{c}
\text{(ALLOC)} \frac{i : v = \text{new } T() \quad i \in \text{meth} \quad \overline{\text{meth}}^c}{i : v \xrightarrow{\text{OUT}}_c \widehat{o}_i} \quad \text{(MOVE)} \frac{i : v = u \quad i : p \xrightarrow{\text{IN}}_c \widehat{o}}{i : p[u/v] \xrightarrow{\text{OUT}}_c \widehat{o}} \\
\text{(LOAD)} \frac{i : u = v.f \quad i : v.f \xrightarrow{\text{IN}}_c \widehat{o}}{i : u \xrightarrow{\text{OUT}}_c \widehat{o}} \quad \text{(STORE-1)} \frac{i : u.f = v \quad i : v \xrightarrow{\text{IN}}_c \widehat{o}}{i : u.f \xrightarrow{\text{OUT}}_c \widehat{o}} \\
\text{(STORE-2)} \frac{i : u.f = v \quad i : v \xrightarrow{\text{IN}}_c \widehat{o} \quad i : w.f \xrightarrow{\text{IN}}_c \widehat{o}' \quad w \neq u}{i : w.f \xrightarrow{\text{OUT}}_c \widehat{o} \quad i : w.f \xrightarrow{\text{OUT}}_c \widehat{o}'} \\
\text{(CFG-JOIN)} \frac{j \xrightarrow{\text{next}} i \quad j : p \xrightarrow{\text{OUT}}_c \widehat{o} \quad \forall k : (k \xrightarrow{\text{next}} i) \implies (k : p \xrightarrow{\text{OUT}}_c *)}{i : p \xrightarrow{\text{IN}}_c \widehat{o}} \\
\text{(FRAME-1)} \frac{i : v \xrightarrow{\text{IN}}_c \widehat{o} \quad \neg(i : v = *) \quad \neg(i : \langle \text{unknown} \rangle)}{i : v \xrightarrow{\text{OUT}}_c \widehat{o}} \\
\text{(FRAME-2)} \frac{i : p \xrightarrow{\text{IN}}_c \widehat{o} \quad p = v.* \quad \neg(i : *. \text{meth}(*)) \quad \neg(i : *. g = *) \quad \neg(i : v = *) \quad \neg(i : \langle \text{unknown} \rangle)}{i : p \xrightarrow{\text{OUT}}_c \widehat{o}} \\
\text{(FRAME-3)} \frac{i : *. f = * \quad i : p \xrightarrow{\text{IN}}_c \widehat{o} \quad f \notin p}{i : p \xrightarrow{\text{OUT}}_c \widehat{o}} \\
\text{(CALL)} \frac{i : v. \text{meth}(*), \quad i : v \xrightarrow{\text{IN}}_c \widehat{o}, \quad \widehat{o} \in T, \quad c' = \mathcal{NC}(i, c, \widehat{o})}{\overline{\text{meth}}_{T}^{c'} \quad i \xrightarrow{\text{calls}}_c^c \text{meth}_T} \\
\text{(ARGS)} \frac{i \xrightarrow{\text{calls}}_{c'}^c \text{meth} \quad i : p \xrightarrow{\text{IN}}_c \widehat{o} \quad j = \text{meth}(0)}{j : p[\text{arg}_n^i / \text{arg}_n^{\text{meth}}] \xrightarrow{\text{IN}}_{c'} \widehat{o}} \\
\text{(RET)} \frac{j : \text{return} \quad j \in \text{meth} \quad i \xrightarrow{\text{calls}}_d^c \text{meth} \quad j : p \xrightarrow{\text{IN}}_d \widehat{o} \quad p = v.* \quad \left\{ \forall \text{meth}', c' : (i \xrightarrow{\text{calls}}_{c'}^c \text{meth}') \implies (\exists j', q' : (j' : \text{return}) \wedge (j' \in \text{meth}') \wedge (p = q'[\text{arg}_n^{\text{meth}'}/\text{arg}_n^i]) \wedge (j' : q' \xrightarrow{\text{IN}}_{c'} *)) \right\}}{i : p[\text{arg}_n^{\text{meth}} / \text{arg}_n^i] \xrightarrow{\text{OUT}}_c \widehat{o}}
\end{array}$$

■ **Figure 2** Inference Rules for Defensive Points-to Analysis.

CFG-Join. The next rule deals with merging information from an instruction’s predecessors (or merely propagating it, in the case of a single predecessor).

Informally, the rule states that if *some* predecessor instruction, j , has established that p can point to \widehat{o} , *and* if all other predecessors, k , establish that p points to *something* (so that its points-to set is non-empty, i.e., bounded) then the information is propagated to the points-to relation of the successor instruction. (We use $*$ to mean “any value”, throughout

the rules.) Note the defensive handling: if even a single predecessor has an unbounded (i.e., empty) points-to set for p , then the rule is not triggered and the resulting points-to set remains empty.

Frame-1, Frame-2, Frame-3. The next three rules are *frame rules*, responsible for the propagation of unchanged information.

Informally, the first rule merely says that points-to information for local variables (i.e., an access path consisting of just “ v ”) is maintained after an instruction, if it existed before it, as long as the instruction does not directly assign the local variable (as is the case for a load, move, or allocation directly into this local variable). The soundness of this rule is predicated on our earlier assumption of *stack frame isolation*: local variables are isolated from each other, thread-private, and private to their allocating method. Therefore their points-to set cannot change, except with instruction such as the above.

This is the first time we see a treatment of `<unknown>` instructions, which can encode any richer instruction set than our basic intermediate language. The analysis conservatively avoids propagating any points-to information over an unknown instruction. This is also used to handle concurrency, under our simplified model: both `monitorenter/monitorexit` instructions and all accesses to `volatile` variables in the input program are represented simply as `<unknown>` instructions in our intermediate language. (The treatment of `<unknown>` collectively by the analysis rules ensures that all heap information is dropped at that program point, i.e., points-to sets are empty after the instruction.)

The next two rules apply in the case of complex access paths, i.e., of length 2 or more. (Actually rule FRAME-3 also applies to variable-only access paths, but not meaningfully: that case is subsumed by FRAME-1.) First, similarly to the earlier rule, points-to information for the access path is maintained after an instruction (assuming it held before it) unless the instruction assigns the same base variable (again via a load, move, or allocation), or is a call, store, or unknown. Second, points-to information for complex access paths is propagated over all store instructions that affect fields not participating in the access path.

The soundness of these rules is predicated on the *object isolation* and *concurrency model* assumptions of Section 2.3. Under these assumptions, the only way to change the points-to set of an access path is via store instructions (on the same field), changing the base of the access path, invoking (potentially opaque) methods, and executing unknown instructions (including `monitorenter/monitorexit`). The rules have strong preconditions to preclude these cases. At the level of the model, we only care about soundness under the given assumptions, no matter how strict. In Section 5 we will discuss practical enhancements – e.g., when method calls are fine because the analysis has computed the full potential of their effects on the heap.

Call. The next rule uses points-to information to establish a sound call-graph. The $i \xrightarrow[c]{calls}$ `meth` relation over-approximates information using the same approach as points-to sets: for a given invocation site, i , and context, c , the relation holds either an empty set (i.e., no matching values exist for (i, ctx) – denoting an unbounded set of destinations – or an over-approximation (i.e., a superset) of all possible targets of the invocation at i under c .

The rule is mostly a straightforward lookup of the target method, based on the receiver object’s type. There are a couple of subtleties, however. The receiver object needs to have an upper-bounded (i.e., non-empty) points-to set, a new context is constructed using function \mathcal{NC} , and the target method is considered reachable under the new context. The exact definition of \mathcal{NC} will determine the context-sensitivity of the analysis. (We will return to this point promptly.)

Args. The ARGs rule handles points-to information propagation over calls, from caller to callee. Points-to information for rebased access paths is established for the first instruction ($j = \text{meth}(0)$) of a called method, under the callee’s established context. The rule examines all access paths whose base variable is an actual argument of the call, as long as they have some points-to information (before the invocation).

Recall our discussion of Section 3 regarding method calls and the use of context. The points-to information established at a callee cannot be conflating different callers – there may be unknown callers for the same method, either in existing code (e.g., via reflection) or in dynamically loaded code. Therefore, if we might mix callers, the only sound inference for local points-to sets is \top : we cannot bound the values that all callers may pass. Instead, we need to have contexts that uniquely identify the caller, so that we can safely propagate bounded points-to sets.

A straightforward way to ensure that the pair (meth, c') uniquely identifies invocation instruction i and context c is to use *call-site sensitivity* [27, 28]: c' is formed by combining i and c – that is, $\mathcal{NC}(i, c, \hat{\delta}) = \text{cons}(i, c)$. (Contexts can typically grow only up to a pre-determined depth, at which point the \mathcal{NC} function will not return anything, the CALL rule will fail to make a $\xrightarrow{\text{calls}}$ inference, hence the current rule will not fire, leaving points-to sets at the callee empty, i.e., undetermined.)

Ret. The final rule perform a similar propagation of values, this time from callee to caller. The rule is significantly complicated by its last condition (the forall-exists implication), which is key for soundness. The rule states that if some callee has points-to information for complex access path p at a return point, then this information is propagated to the caller, provided that *all other callees* for the same instruction, i , and caller context, c , also have *some* (i.e., non-empty) points-to information for the same access path p at their return point. A further complication is that access path p will appear rebased differently for each one of the callees – e.g., access path `actual.field` may appear as `formalA.field` and `formalB.field` in two callees A and B. The rule has to also account for such rebasing.

Note also the earlier condition that access path p be complex, i.e., to have length greater than 1. This reflects call-by-value semantics for references: for a call `foo(actual)` to a method with signature `foo(F formal)`, the points-to information of access path `formal` is not reflected back to the caller, yet the points-to information of longer access paths, e.g., `formal.fld`, is.

The handling of a method return is the only point where a context can become stronger. Facts that were inferred to hold under the more specific context, c' , are now established, modulo rebasing, under c . Since c' has to uniquely identify c , typically c will be shorter by one context element.

4.3 Reasoning

We prove the soundness of the analysis under an informal language model. We do not attempt to formalize the full effects of opaque code (e.g., what reflection or native code can or cannot do). Such a formalization would be tedious and partial, as new capabilities are added to reflection or dynamic loading APIs with every JDK version. Instead, we establish that the analysis rules always compute over-approximate finite points-to sets (or empty sets), and that this property cannot be affected by opaque code under the common informal understanding of the assumptions of Section 2.3. For instance, it is clear from the “stack frame isolation” assumption that local variables cannot change values except by action of the current instruction, i.e., that rule FRAME-1 is alone responsible for soundly transferring such points-to information from the program point before an instruction to after.

There are two main properties of the defensive analysis:

- Soundness: the analysis computes an over-approximation of points-to sets that may arise during any program execution. Any non-empty set contains a superset of its dynamic contents under any possible execution. Any empty set is considered trivially “over-approximate”, to avoid special-casing all our statements. In effect, the analysis produces a set of soundness markers, $Claim(P)$, which coincide with the non-empty points-to sets. No claims are made about empty points-to sets.
- Laziness: the analysis does not waste work; elements that enter a points-to set are never removed.

► **Theorem 1.** *There exists an evaluation order of the rules, such that the defensive analysis model is sound: all points-to sets computed are over-approximate, i.e., are either empty or contain all possible values arising during program execution, under the assumptions of Section 2.3.*

Proof. The proof is inductive. Initially, all points-to/call-target sets encoded in relations $i : \mathbf{p} \xrightarrow{IN}_c$, $i : \mathbf{p} \xrightarrow{OUT}_c$, $i \xrightarrow{calls^c}_{c'}$ are empty. (We treat relation $i : \mathbf{p} \xrightarrow{IN}_c \widehat{o}$ as encoding a set of \widehat{o} s for given i, \mathbf{p}, c ; relation $i \xrightarrow{calls^c}_{c'} \mathbf{meth}$ as encoding a set of \mathbf{meth} s for given i, c, c' , etc.)

Therefore, we start from a trivially over-approximate state.

Importantly, the inductive step does *not* hold for a single application of a rule. Intermediate states of evaluation may not be over-approximate: an element may enter a set before the rest of its contents. (For instance, consider a statement $v = u$ and prior points-to set $\{\widehat{o}_1, \widehat{o}_2\}$ for u . A single application of the MOVE rule for \widehat{o}_1 will leave the points-to set of v in a non-over-approximate state: the set will be missing the \widehat{o}_2 value.)

Thus, the inductive step applies to states after past rules have been evaluated fully. Consider a rule R as a monotonic update to a set of values d . That is, $R(d) \supseteq d$. A rule has been fully evaluated at fixpoint, i.e., when $R(d) = d$. The next inductive step considers the state after a full evaluation of any rule.

The inductive step of the proof is captured in a lemma:

► **Lemma 2.** *The analysis rules preserve soundness under full single-rule evaluation. That is, if relations $i : \mathbf{p} \xrightarrow{IN}_c$, $i : \mathbf{p} \xrightarrow{OUT}_c$, and $i \xrightarrow{calls^c}_{c'}$ encode over-approximate points-to/call-target sets before a full evaluation of a rule, they will encode over-approximate sets after a full evaluation.*

Proof sketch of lemma 2. The lemma is established by exhaustive examination of the rules. We mentioned key parts of the reasoning in our earlier presentation of the rules. All rules over complex access paths (i.e., of length ≥ 2) affect the heap and require the “concurrency model” and “object isolation” assumptions of Section 2.3. Rules on plain-variable access paths use the “stack-frame isolation” assumption. Every rule is careful to produce values for points-to/call-target sets only if all input sets are non-empty (i.e., guaranteed over-approximate and bounded), and to consider all possible such values. For rules CALL, ARGS, and RET the lemma holds only under the previously-stated assumption on the \mathcal{NC} constructor: the pair (\mathbf{meth}, c') needs to uniquely identify invocation instruction i and context c . Consider, for example, rule ARGS. We need to establish that the points-to set $j : \mathbf{p}[arg_n^i / arg_n^{\mathbf{meth}}] \xrightarrow{IN}_{c'}$ is over-approximate given that $i : \mathbf{p} \xrightarrow{IN}_c$ is. (The rule form makes the former be a superset of the latter, we need to reason that they are actually the same set.) Instruction j uniquely identifies method \mathbf{meth} and actual-to-formal access-path rebasing can never merge access paths (since different formal variables cannot have the same names). If c' and \mathbf{meth} arise for only a single call-site and caller-context pair, (i, c) , then the property holds. ◀

The lemma establishes the inductive step of our proof. The sets computed by the analysis are initially over-approximate and remain over-approximate after every full evaluation of a single rule. At fixpoint, when full evaluation of any rule no longer changes the output sets, the property holds, concluding the theorem’s proof. ◀

An interesting question is whether *any* evaluation order of the rules is guaranteed to yield sound points-to sets at fixpoint. The answer is “almost yes”. All but one of the analysis rules are monotonic (in the usual domain of sets, i.e., with the empty set at the bottom), therefore yield a confluent evaluation: any order will yield the same result at fixpoint. (We have a machine-checked proof of the latter property, by encoding the rules in the Datalog language, which allows only recursion through monotonic inferences.) The single exception is the RET rule. There is hidden non-monotonicity in the \forall iteration over call-graph edges, which contains an implication. If the CALL rule is not fully evaluated when the RET rule applies, it is possible to produce points-to sets that will later be invalidated, because more callees will be discovered (for whom the points-to relationship does not hold for the given access path). Therefore, for soundness to hold, the analysis rules have to always apply in such a fashion that the CALL rule is fully evaluated (not globally but on its own, per the earlier definition) before the RET rule is considered. This evaluation order should be enforced by any sound implementation of the rules of Figure 2.

Based on the above observation on the rules’ monotonicity, we also establish our laziness result.

► **Theorem 3.** *A points-to set encoded in our analysis relations grows monotonically, as long as the RET rule is applied only during local fixpoints (i.e., after full evaluation) of the CALL rule.*

5 Implementation and Discussion

We have implemented defensive analysis in the Datalog language and integrated it with the DOOP Datalog framework for may-point-to analysis of Java bytecode [5]. The full implementation consists of over 400 logical rules, yet the minimal model of Section 4 captures well its essential features. We also completed a second, largely equivalent, implementation on the Soufflé Datalog engine [26]. Both implementations are publicly available at <https://bitbucket.org/yanniss/doorp>.

The defensive analysis model admits several enhancements and refinements, as well as gives rise to observations. We discuss such topics next, especially noting those that pertain to our full-fledged implementation of the analysis.

Observations. A defensive analysis is naturally modular, yet the question is whether it can produce useful results. The analysis can be applied to any subset of the code of an application or library and it will produce sound inferences. Omitting code merely means that more points-to sets will end up being empty: the analysis only infers points-to sets when an upper-bound of their contents is known based on the current code under analysis. This defensive approach, however, may end up computing too many empty points-to sets. Therefore, the key quality metric is that of the analysis’s *coverage*: for how many program elements (e.g., local variables) can the analysis produce non-empty points-to information? Coverage has similarly been used as a key metric in other work that infers specifications modularly [6].

Additionally, a defensive analysis is not in competition with a conventional, unsound analysis, but instead complements it. The defensive analysis computes which of the points-to

sets have known upper bounds and which are potentially undetermined. If, instead of an empty set, a client desires to receive the (incomplete) subset of known contents for non-bounded points-to sets, the results of the two analyses can be trivially combined.

Pragmatics. With minor adaptation, the analysis logic can work on static single assignment (SSA) input. Our implementation is indeed based on an SSA intermediate language. The benefit is that for trivial access paths (just a single variable) points-to information does not need to be kept per-instruction: the points-to set remains unchanged, since the variable is not re-assigned.

A full-fledged analysis should cover more language features than the model of Section 4. Our implementation handles, in a manner similar to the earlier rules, features such as static and special method invocations, static fields, final fields, constructors (also implicitly initializing fields to `null`), and more.

Expanding the Analysis Reach. Defensive analysis is naturally pessimistic. Its key feature is that it will populate points-to sets only when it can establish that they are bounded. However, the analysis uses simplistic techniques to establish such boundedness, i.e., it recognizes guaranteed-safe cases.

There are several sound inferences that the analysis could make but the model of Section 4 does not. Although defensive analysis will never reach the inferences of an unsound analysis (even without any opaque code), it can be enhanced to approach it. Arbitrarily complex mechanisms can be added to increase the coverage of the analysis (i.e., the true properties it can infer precisely):

- The rule shown earlier for control-flow merge points is conservative. Information propagates at control-flow merge points if all of the predecessors have some points-to information for the access path in question. This condition is too strict: several predecessors will not have points-to information for an access path simply because the access path is not even assigned in the predecessor branch (e.g., it is based on a local variable that is set on a different branch only). Consider a program fragment:

```

1  x.f = new A();
2  while (...) {
3    y = x.f;
4  }
```

The head of the loop has two control-flow predecessors: one due to linear control flow and one due to the loop back-edge. However, the loop itself does not change the points-to set of `x.f`. It is too conservative to demand that the back-edge also have a bounded points-to set for `x.f` before considering the linear control-flow edge.

In our implementation we have special support for detecting that a program path does not affect an access path. We use this to limit the \forall quantification of the rule to range over “relevant” predecessors. We note that this scenario only applies to complex access paths in practice, due to the SSA form of our input.

- When an unknown method call is encountered, the analysis assumes worst-case behavior with respect to its heap information. This can be relaxed arbitrarily by modeling system methods and annotating them appropriately. Possible information about calls includes “this library call does not affect user-level objects”, “this method only affects its arguments”, “this method does not affect static variables”, etc. Additional manual modeling includes library collections (including arrays) which can be represented as abstract objects.

Our current implementation does some minimal modeling of library collections and annotates only a handful of methods, as a proof-of-concept. A representative example is that of method `Float.floatToRawIntBits`. This native method is called by the implementation of the `put` operation in Java `HashMaps` and, since it is opaque, would prevent all propagation of points-to information beyond a `put` call.

- The analysis coverage can be expanded by employing it jointly with a *must-alias* analysis [15, 7, 35], an *escape* analysis [4, 11], and a *thread-escape* analysis. A *must-alias* analysis will increase the applicability of the rule for heap loads, and can be combined with the rule for heap stores to enable more strong updates. An *escape* analysis will result in less conservativeness in the propagation of information to further instructions (i.e., in frame rules). A *thread-escape* analysis can help relax our concurrency model. We currently support simple, conservative versions of all three analyses in our implementation, but do not enable them by default.

Context depth. As seen earlier, a defensive analysis may compute empty (undetermined) points-to sets because it has reached its maximum context depth. It is worth pointing out, however, that method calls *further away* than the maximum context depth can influence the points-to inferences of a method. For an easy example, consider the case of a large number, N , of methods that form a call chain and unconditionally return to their callers what their callee returns to them. If the final (N -th) method returns a new object, then that object will propagate all the way back to the first method of the call chain, regardless of the maximum context depth, D . The limitation of context depth only concerns properties that *depend on* conditions established more than D calls back in the call-stack.

6 Evaluation

There are five research questions that our evaluation seeks to answer:

- **RQ1:** Does defensive analysis produce coverage for large parts of realistic programs? Or do points-to sets overwhelmingly stay empty?
- **RQ2:** Does the coverage of defensive analysis benefit from its advanced features (i.e., inter-procedural handling, as well as handling of control-flow merging)?
- **RQ3:** Does defensive analysis have an acceptable running time, given that it is flow-sensitive and context-sensitive?
- **RQ4:** Does defensive analysis yield results that can benefit a client that requires soundness, such as an optimization?
- **RQ5:** Can benefits be obtained for a fully relaxed concurrency model, as opposed to the model of Section 2.3?

Setup. Since defensive analysis is a unique beast, it is indeed an interesting question to ask what it can be compared against. As closest comparable (though still a very dissimilar analysis) we chose to compare to a highly-precise conventional analysis with state-of-the-art best-effort soundness: a 2-object-sensitive/heap-sensitive analysis (*2objH*) with reflection support. This is the most precise analysis in the DaCapo framework that still manages to scale to the majority of the DaCapo benchmarks. We use static best-effort reflection handling (`-enable-reflection-classic` flag), i.e., the analysis tries to statically resolve all reflection calls based on string matching.

We analyze, under JDK 1.7.0_75, the DaCapo benchmark programs [3] v.2006-10-MR2 as well as v.9.12-Bach. The 9.12-Bach version contains several different programs, as well as more recent versions of some of the same programs. (We show results for all of the

v.2006-10-MR2 benchmarks and for those of the v.9.12-Bach benchmarks that could be analyzed by the DOOP framework in under 3 hours.) We also use the two non-Android benchmarks (NTI, jFlex) from the Julia set by Nikolić and Spoto [24].

We use the LogicBlox Datalog engine, v.3.10.14, on a Xeon E5-2667 v.2 3.3GHz machine with only one thread running at a time and 256GB of RAM.

Defensive analysis is run with a 5-call-site-sensitive context (*5def* for short). 3 instances (of 44 total) did not finish with the default precision in 3hrs: the 2objH baseline did not finish for jython and h2; xalan did not finish for the 5def analysis. In these cases we used lower precision: context-insensitive for the unsound analysis and 4-call-site-sensitive (*4def*) for defensive. We use diacritical marks (* and ^) in the figures to remind the reader of the different analysis setting for these benchmarks.

Coverage. Figure 3 shows the coverage of defensive analysis, i.e., the number of non-empty points-to sets (for local variables) computed for all benchmarks. The input program is in SSA form, therefore the points-to sets for variables are a normalized representation of all points-to information in the program: they reflect the analysis-computed values for all program expressions, separately for each control-flow point.

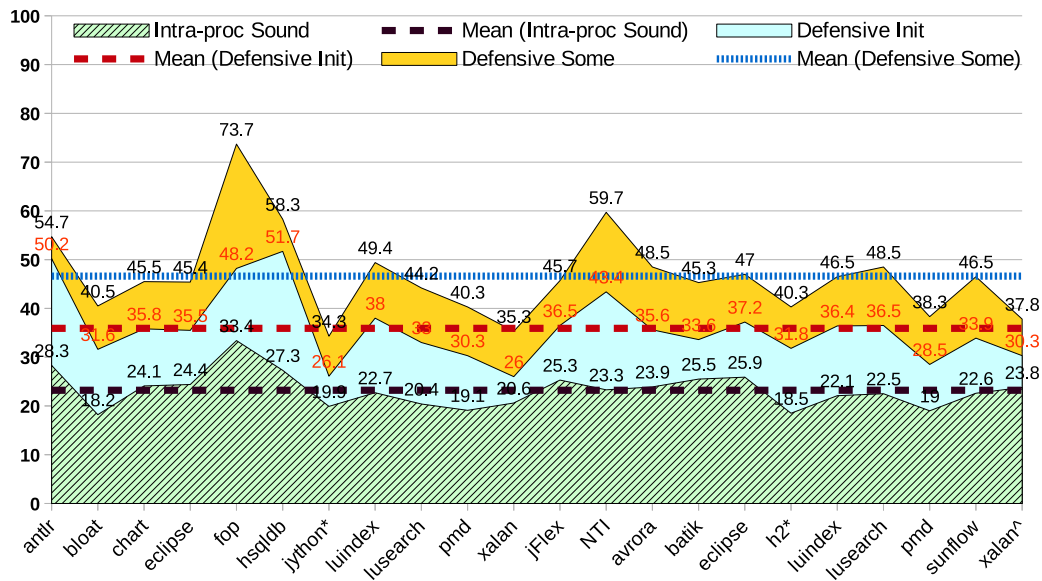
The analysis yields non-empty points-to sets for a significant portion of each program – the median benchmark has **45.6%** of variables with points-to information for *some* context, while **35.5%** have points-to information for a context **Init** (i.e., unconditionally).⁷ It is worth emphasizing that conditional points-to guarantees (under *some* context) are valuable in a defensive analysis: they are often the best any analysis can ever do! Recall our earlier discussion of Section 3: many of the useful inferences of a defensive analysis will be under *some* context even when the inference holds under *all known* contexts in existing code. No analysis can preclude the existence of other callers in opaque, and possibly not yet existing, code. Such callers can arise in dynamically generated code and can invoke existing methods, e.g., using reflection.

Thus, the defensive analysis achieves a large proportion of the benefits of an unsound analysis, while guaranteeing these results against uses of opaque code. We can answer **RQ1** affirmatively: defensive analysis covers a large part of realistic programs (over one-third unconditionally; close to one half under specific calling conditions), despite its conservative nature.

Comparison with intra-procedural. We have earlier referred to the “easy”, intra-procedural parts of the analysis reasoning: what a compiler or VM would likely do to perform sound local data-flow analysis. This is the subject of **RQ2**, also answered by Figure 3. The figure includes results for an intra-procedural baseline analysis that captures the low-hanging fruit of sound reasoning: local variables that directly or transitively (via “move” instructions) get assigned an allocated object. That is, the “Intra-proc Sound” analysis is otherwise the same as the full “defensive” logic, with the exception of the new “interesting” cases (control-flow merging, heap manipulation, and inter-procedural propagation).

The result answers **RQ2** affirmatively: defensive analysis has significantly higher coverage than the baseline intra-procedural analysis. (And the difference only grows when considering an actual client, in later experiments.) Although the benefit is not broken down further in the

⁷ If a variable has a points-to value for context **Init**, then it also has that value under every specific context that arises for the variable. Therefore, points-to sizes for **Init** are always lower than conditional, context-specific sizes.



■ **Figure 3** Percentage of application variables (deemed reachable by baseline 2objH analysis) that have non-empty points-to sets for defensive analysis under some context and INIT context (no assumptions). Intra-procedural sound points-to analysis (defensive minus the complex cases) shown as baseline. Arithmetic means are plotted as lines.

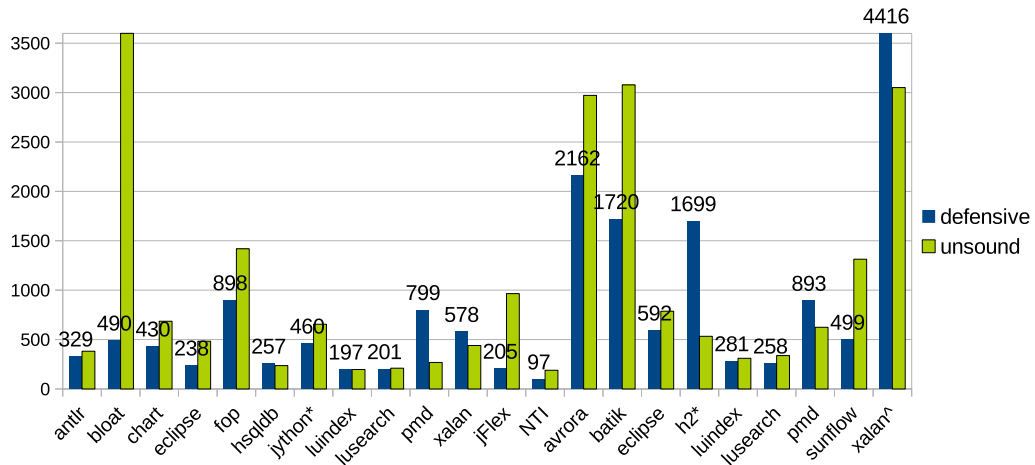
figure, the handling of method calls alone (i.e., rules CALL, ARGS and RET) is responsible for the lion’s share of the difference between the full defensive analysis and the intra-procedural sound analysis.

Running time. Figure 4 shows the running times of the analysis, plotted next to that of 2objH, for reference. Although the two analyses are dissimilar, 2objH is qualitatively the closest one can get to defensive analysis with the current state of the art: it is an analysis with high precision, run with best-effort soundness support. Therefore, 2objH can serve as a realistic point of reference. As can be seen, the running times of defensive analysis are realistically low, although its flow-sensitive and 5-call-site-sensitive nature suggests it would be a prohibitively heavy analysis. This answers **RQ3** and confirms the benefits of laziness: a defensive analysis that only populates points-to sets once they are definitely bounded, achieves scalability for deep context.

Client analysis: devirtualization. Our baseline analysis, 2objH, is highly precise and effective in challenges such as devirtualizing calls (resolving virtual calls to a single target method). On average, it can devirtualize 89.3% of the calls in the benchmarks studied (min: 78.5%, max: 95.2%). However, these results are unsound and a compiler cannot act upon them. For optimization clients, such as devirtualization, soundness is essential. Using sound results, a JIT compiler can skip dynamic tests (of the inline caching optimization) for all calls that the analysis soundly covers.

Figure 5 shows the virtual calls that defensive analysis devirtualizes, as a percentage of those devirtualized by the unsound analysis.

As can be seen, defensive analysis manages to recover a large part of the benefit of an unsound analysis (median **44.8%** for optimization under a context guard, **38.7%** for unconditional, **Init** context, optimization), performing much better than the baseline intra-procedural must-analysis (at 14.6%). This answers **RQ4** affirmatively: the coverage of defensive analysis translates into real benefit for realistic clients.



■ **Figure 4** Running time (sec) of defensive analysis, with running time of 2objH (with unsound reflection handling) shown as a baseline. Labels are shown for defensive analysis only to avoid crowding the plot.

Concurrency model. A compiler (JIT or AOT) author may (rightly) remark that the concurrency model of Section 2.3 is not appropriate for automatic optimizations. The Java concurrency model permits a lot more relaxed behaviors, so the analysis is not sound for full Java as stated. However, the benefit of defensive analysis is that it starts from a sound basis and can add to it conservatively, only when it is certain that soundness cannot possibly be violated. Accordingly, we can remove the assumption that all shared data are accessed while holding mutexes, by applying the load/store rules only when objects trivially do not escape their allocating thread. We show the updated numbers for the devirtualization client (now fully sound for Java!) in Figure 6. The difference in impact is minimal: **43%** of virtual call sites can be devirtualized conditionally, under some context, while **36%** can be devirtualized unconditionally. This helps answer **RQ5**: defensive analysis can yield actionable results for a well-known optimization, under the Java memory model, for a large portion of realistic programs.

Points-to set sizes. Finally, it is interesting to quantify the precision of the defensive analysis, for the points-to sets it covers. This precision is expected to be high, since defensive analysis is flow- and context-sensitive, but exact figures help put it in perspective.

Figure 7 shows average points-to set sizes for the defensive analysis vs. the 2objH analysis. The sets (excluding null values) are computed over variables covered by both analyses, for non-empty defensive analysis sets and under context **Init** of the defensive analysis, i.e., unconditionally. (The numbers are for the simplistic concurrency model, but remain unchanged to two significant digits for the relaxed concurrency model.)

As can be seen, the defensive analysis is highly precise when it produces non-empty points-to sets, typically yielding points-to set sizes very close to 1. 2objH is also a very precise analysis (for variables with bounded points-to sets), so it remains competitive, yet clearly less precise. Notably, points-to set sizes close to 1 are the Holy Grail of points-to analysis: such precision is actionable for nearly all conceivable clients of a points-to analysis.

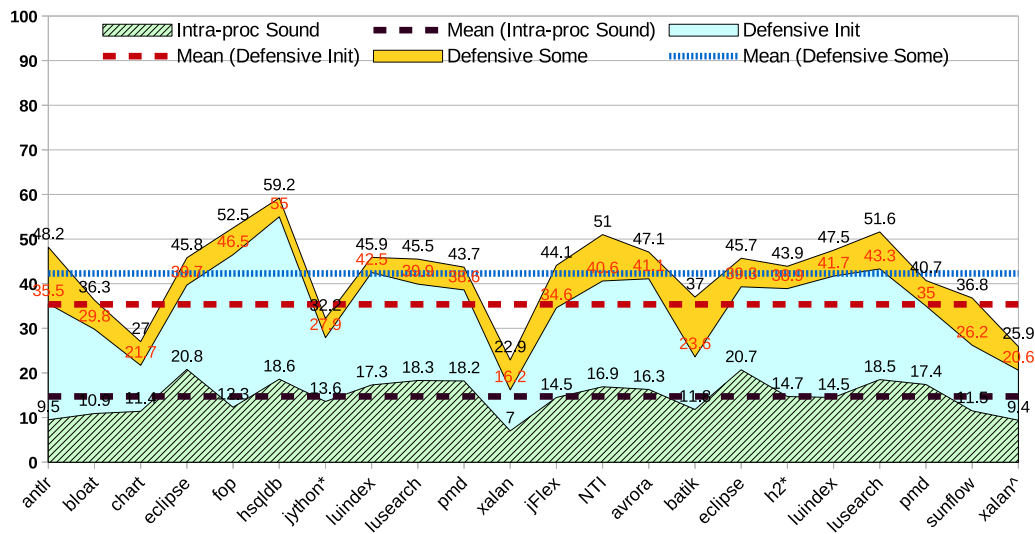


Figure 5 Virtual call sites that are found to have receiver objects of a single type. These call sites can be soundly devirtualized. Numbers are shown as percentages of devirtualization achieved by unsound 2objH analysis.

7 Related Work

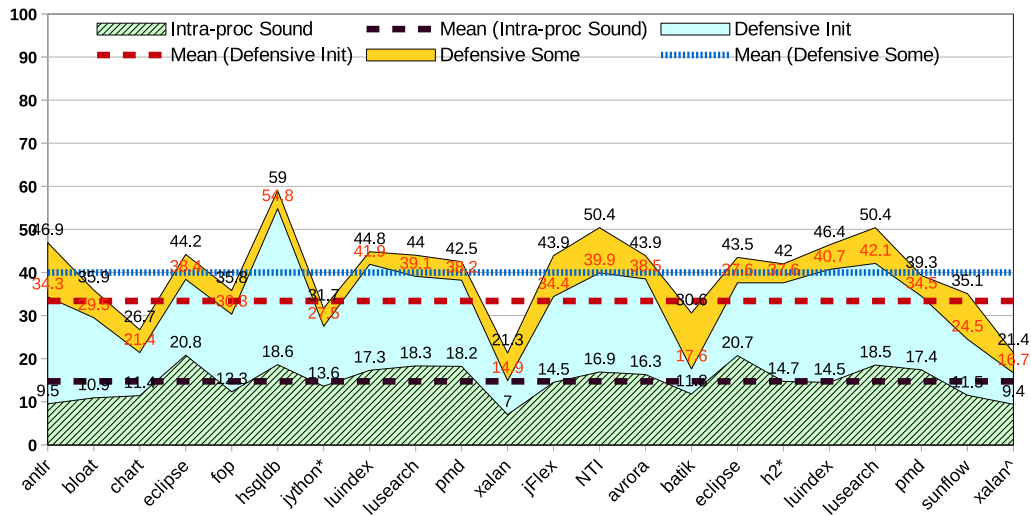
There is certainly past work that attempt to ensure a sound whole-program analysis, but none matches the generality and applicability of our approach. We selectively discuss representative approaches.

The standard past approach to soundness for a careful static analysis has been to “bail out”: the analysis detects whether there are program features that it does not handle soundly, and issues warnings, or refuses to produce answers. This is a common pattern in abstract-interpretation [8] analyses, such as Astrée [10], which have traditionally emphasized sound handling of conventional language features. However, this is far from a solution to the problem of being sound for opaque code: refusing to handle the vast majority of realistic programs can be argued to be sound, but is not usefully so. In contrast, our work handles *all* realistic programs, but returns partial (but sound) results, i.e., produces non-empty points-to sets for a subset of the variables. It is an experimental question to determine whether this subset is usefully large, as we do in our evaluation.

Hirzel et al. [13, 14] use an online pointer analysis to deal with reflection and dynamic loading by monitoring their run-time occurrence, recording their results, and running the analysis again, incrementally. However, this is hardly a *static* analysis and its cost is prohibitive for precise (context-sensitive) analyses, if applied to all reflection actions.

Lattner et al. [17] offer an algorithm that can apply to incomplete programs, but it assumes that the linker can know all callers (i.e., there is no reflection – the analysis is for C/C++) and the approach is closely tied to a specific flow-insensitive, unification-based analysis logic [34], necessary for simultaneously computing inter-related points-to, may-alias, and may-escape information.

Sreedhar et al. [33] present the only past approach to explicitly target dynamic class loading, although only for a specific client analysis (call specialization). Still, that work ends up making many statically unsound assumptions (requiring, at the very least, programmer intervention), illustrating well the difficulty of the problem, if not addressed defensively. The approach assumes that only the public API of a “closed world” is callable, thus ignoring



■ **Figure 6** Virtual call sites (percentage of 2objH) that are found to have receiver objects of a single type. Updates Figure 5, this time with soundness under a relaxed memory model.

many uses of reflection. (With reflection, any method is callable from unknown code, and any field is accessible.) It “[does] not address the Java features of reloading and the Java Native Interface”. It “optimistically assumes” that “[the extant state of statically known objects] remains unchanged when they become reachable from static reference variables”. It is not clear whether the technique is conservative relative to adversarial native code (in system libraries, since the JNI is ignored). Finally, the approach assumes the existence of a sound may-point-to analysis, even though none exists in practice!

Traditional conservative call-graph construction (*Class Hierarchy Analysis (CHA)* [9] or *Rapid Type Analysis (RTA)* [1]) is unsound. Such algorithms explore the entire class hierarchy for matching (overriding) methods and consider all of them to be potential virtual call targets. However, even this is not sufficient for a sound static analysis of opaque code: classes can be generated and loaded dynamically during program execution. CHA cannot find target methods that do not even exist statically, yet modeling them is precisely what is needed for soundness in real-world conditions. For instance, Java applications, especially in the enterprise (server-side) space, employ dynamic loading heavily, and patterns such as *dynamic proxies* have been standardized and used widely since the early Java days.

Furthermore, such heuristic “best-effort” over-approximation is detrimental to analysis precision and performance. CHA is an example of a loose over-approximation in an effort to capture most dynamic behaviors. Loose over-approximations compute many more possible targets than those that realistically arise. This yields vast points-to sets that render the analysis heavyweight and useless due to imprecision. (Avoiding such costs is exactly why past analyses have often opted for glaringly unsound handling of opaque code features.) Our lazy representation of “don’t know”/“cannot bound” values as empty sets addresses the problem, by keeping all points-to sets compact.

The conventional handling of reflection in may-point-to analysis algorithms for Java [12, 18, 22, 20, 29, 19] is unsound, instead relying on a “best-effort” approach. Such past analyses attempt to statically model the result of reflection operations, e.g., by computing a superset of the strings that can be used as arguments to a `Class.forName` operation (which accepts a name string and returns a reflection object representing the class with that name). The analyses are unsound when faced with a completely unknown string: instead of assuming

Benchmark		Average points-to over same vars defensive	2objH
DaCapo 2006-10-MR2	antlr	1.01	1.10
	bloat	1.02	2.12
	chart	1.09	1.09
	eclipse	1.06	1.31
	fop	1.00	1.03
	hsqldb	1.01	1.04
	kython*	1.01	6.05
	luindex	1.02	1.02
	lusearch	1.04	1.06
	pmd	1.01	1.05
	xalan	1.05	1.12
	jFlex	1.01	1.02
	NTI	1.03	1.03
DaCapo 9.12-Bach	avroa	1.05	3.04
	batik	1.04	1.05
	eclipse	1.07	1.53
	h2*	1.04	2.07
	luindex	1.01	1.04
	lusearch	1.03	1.08
	pmd	1.01	1.04
	sunflow	1.05	1.08
	xalan^	1.04	1.19
mean	1.03	1.51	

■ **Figure 7** Average number of abstract objects per variable, for variables for which both analyses compute results.

that *any* class object can be returned, the analysis assumes that *none* can. The reason is that over-approximation (assuming any object is returned) would be detrimental to the analysis performance and precision. Even with an unsound approach, current algorithms are heavily burdened by the use of reflection analysis. For instance, the documentation of the WALA library directly blames reflection analysis for scalability shortcomings [12],⁸ and enabling reflection on the DOOP framework slows it down by an order of magnitude on standard benchmarks [29]. Furthermore, none of these approaches attempt to model dynamic loading – a ubiquitous feature in Java enterprise applications.

8 Conclusions

Static analysis has long suffered from unsoundness for perfectly realistic language features, such as reflection, native code, or dynamic loading. We presented a new analysis architecture that achieves soundness by being *defensive*. Despite its conservative nature, the analysis

⁸ The WALA documentation is explicit: “Reflection usage and the size of modern libraries/frameworks make it very difficult to scale flow-insensitive points-to analysis to modern Java programs. For example, with default settings, WALA’s pointer analyses cannot handle any program linked against the Java 6 standard libraries, due to extensive reflection in the libraries.” [12]

manages to yield useful results for a large subset of the code in realistic Java programs, while being efficient and scalable. Additionally, the analysis is modular, as it can be applied to any subset of a program and will yield sound results.

We expect this approach to open significant avenues for further work. The analysis architecture can serve as the basis of other sound analysis designs. The defensive analysis itself can be combined with several other analyses (may-escape, must-alias) that have so far been hindered by the lack of a sound substrate.

References

- 1 David F. Bacon and Peter F. Sweeney. Fast static analysis of C++ virtual function calls. In *Proc. of the 11th Annual ACM SIGPLAN Conf. on Object Oriented Programming, Systems, Languages, and Applications*, OOPSLA '96, pages 324–341, New York, NY, USA, 1996. ACM.
- 2 Sandip K. Biswas. A demand-driven set-based analysis. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 372–385, 1997.
- 3 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 *Proceedings of the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006, Portland, Oregon, USA*, OOPSLA '06, pages 169–190, New York, NY, USA, 2006. ACM. doi:10.1145/1167473.1167488.
- 4 Bruno Blanchet. Escape analysis: Correctness proof, implementation and experimental results. In *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 25–37, 1998.
- 5 Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *Proc. of the 24th Annual ACM SIGPLAN Conf. on Object Oriented Programming, Systems, Languages, and Applications*, OOPSLA '09, New York, NY, USA, 2009. ACM.
- 6 Cristiano Calcagno, Dino Distefano, Peter O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, pages 289–300, New York, NY, USA, 2009. ACM. doi:10.1145/1480881.1480917.
- 7 Jong D. Choi, Michael Burke, and Paul Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 232–245, 1993.
- 8 Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM. doi:10.1145/512950.512973.
- 9 Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *ECOOP'95 - Object-Oriented Programming, 9th European Conference, Århus, Denmark, August 7-11, 1995, Proceedings*, ECOOP '95, pages 77–101. Springer, 1995. doi:10.1007/3-540-49538-X_5.

- 10 David Delmas and Jean Souyris. *Astrée: From Research to Industry*, pages 437–451. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007. doi:10.1007/978-3-540-74061-2_27.
- 11 Alain Deutsch. On the complexity of escape analysis. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, pages 358–371, New York, NY, USA, 1997. ACM. doi:10.1145/263699.263750.
- 12 Stephen J. Fink et al. WALA UserGuide: PointerAnalysis. <http://wala.sourceforge.net/wiki/index.php/UserGuide:PointerAnalysis>, 2013.
- 13 Martin Hirzel, Amer Diwan, and Michael Hind. Pointer analysis in the presence of dynamic class loading. In *ECOOP 2004 - Object-Oriented Programming, 18th European Conference, Oslo, Norway, June 14-18, 2004, Proceedings*, ECOOP '04, pages 96–122. Springer, 2004. doi:10.1007/978-3-540-24851-4_5.
- 14 Martin Hirzel, Daniel von Dincklage, Amer Diwan, and Michael Hind. Fast online pointer analysis. *ACM Trans. Program. Lang. Syst.*, 29(2), 2007. doi:10.1145/1216374.1216379.
- 15 Suresh Jagannathan, Peter Thiemann, Stephen Weeks, and Andrew Wright. Single and loving it: must-alias analysis for higher-order languages. In *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 329–341, 1998.
- 16 Davy Landman, Alexander Serebrenik, and Jurgen J. Vinju. Challenges for static analysis of Java reflection – literature review and empirical study. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, 2017.
- 17 Chris Lattner, Andrew Lenharth, and Vikram Adve. Making Context-Sensitive Points-to Analysis with Heap Cloning Practical For The Real World. In *Proc. of the 2007 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, PLDI '07, New York, NY, USA, 2007. ACM.
- 18 Yue Li, Tian Tan, Yulei Sui, and Jingling Xue. Self-inferencing reflection resolution for Java. In *Proc. of the 28th European Conf. on Object-Oriented Programming*, ECOOP '14, pages 27–53. Springer, 2014. doi:10.1007/978-3-662-44202-9.
- 19 Yue Li, Tian Tan, and Jingling Xue. Effective soundness-guided reflection analysis. In Sandrine Blazy and Thomas Jensen, editors, *Static Analysis - 22nd International Symposium, SAS 2015, Saint-Malo, France, September 9-11, 2015, Proceedings*, volume 9291 of *Lecture Notes in Computer Science*, pages 162–180. Springer, 2015. doi:10.1007/978-3-662-48288-9_10.
- 20 Benjamin Livshits. *Improving Software Security with Precise Static and Runtime Analysis*. PhD thesis, Stanford University, December 2006.
- 21 Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. In defense of soundness: A manifesto. *Commun. ACM*, 58(2):44–46, 2015. doi:10.1145/2644805.
- 22 Benjamin Livshits, John Whaley, and Monica S. Lam. Reflection analysis for Java. In *Proc. of the 3rd Asian Symp. on Programming Languages and Systems*, pages 139–160. Springer, 2005. doi:10.1007/11575467_11.
- 23 Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM Trans. Softw. Eng. Methodol.*, 14(1):1–41, 2005. doi:10.1145/1044834.1044835.
- 24 Durica Nikolić and Fausto Spoto. Definite expression aliasing analysis for Java bytecode. In *Theoretical Aspects of Computing - ICTAC 2012 - 9th International Colloquium, Bangalore, India, September 24-27, 2012. Proceedings*, volume 7521 of *ICTAC '12*, pages 74–89. Springer, 2012. doi:10.1007/978-3-642-32943-2_6.

- 25 Xavier Rival. Comment on “what is soundness in static analysis post”, in the PL Enthusiast blog. <http://www.pl-enthusiast.net/2017/10/23/what-is-soundness-in-static-analysis/#comment-1265>, 2017.
- 26 Bernhard Scholz, Herbert Jordan, Pavle Subotic, and Till Westmann. On fast large-scale program analysis in datalog. In *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*, pages 196–206, 2016. doi:10.1145/2892208.2892226.
- 27 Micha Sharir and Amir Pnueli. Two approaches to interprocedural data flow analysis. In Steven S. Muchnick and Neil D. Jones, editors, *Program flow analysis: theory and applications*, chapter 7, pages 189–233. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1981.
- 28 Olin Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie Mellon University, may 1991.
- 29 Yannis Smaragdakis, George Balatsouras, George Kastrinis, and Martin Bravenboer. More sound static handling of Java reflection. In *Proc. of the Asian Symp. on Programming Languages and Systems, APLAS ’15*. Springer, 2015.
- 30 Yannis Smaragdakis, Martin Bravenboer, and Ondřej Lhoták. Pick your contexts well: Understanding object-sensitivity. In *Proc. of the 38th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, POPL ’11*, pages 17–30, New York, NY, USA, 2011. ACM.
- 31 Yannis Smaragdakis, George Kastrinis, and George Balatsouras. Introspective analysis: Context-sensitivity, across the board. In *Proc. of the 2014 ACM SIGPLAN Conf. on Programming Language Design and Implementation, PLDI ’14*, pages 485–495, New York, NY, USA, 2014. ACM. doi:10.1145/2594291.2594320.
- 32 Johannes Späth, Lisa Nguyen Quang Do, Karim Ali, and Eric Bodden. Boomerang: Demand-driven flow- and context-sensitive pointer analysis for java. 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 22:1–22:26. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016. URL: <http://www.dagstuhl.de/dagpub/978-3-95977-014-9>, doi:10.4230/LIPICs.ECOOP.2016.22.
- 33 Vugranam C. Sreedhar, Michael Burke, and Jong-Deok Choi. A framework for interprocedural optimization in the presence of dynamic class loading. In *Proc. of the 2000 ACM SIGPLAN Conf. on Programming Language Design and Implementation, PLDI ’00*, pages 196–207, New York, NY, USA, 2000. ACM.
- 34 Bjarne Steensgaard. Points-to analysis in almost linear time. In *POPL ’96: Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 32–41, 1996.
- 35 Xin Zheng and Radu Rugina. Demand-driven alias analysis for C. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, POPL ’08, pages 197–208, New York, NY, USA, 2008. ACM. doi:10.1145/1328438.1328464.

Legato: An At-Most-Once Analysis with Applications to Dynamic Configuration Updates

John Toman

Paul G. Allen School of Computer Science & Engineering, University of Washington, USA
jtoman@cs.washington.edu

Dan Grossman

Paul G. Allen School of Computer Science & Engineering, University of Washington, USA
djg@cs.washington.edu

Abstract

Modern software increasingly relies on external resources whose location or content can change during program execution. Examples of such resources include remote network hosts, database entries, dynamically updated configuration options, etc. Long running, adaptable programs must handle these changes gracefully and correctly. Dealing with all possible resource update scenarios is difficult to get right, especially if, as is common, external resources can be modified without prior warning by code and/or users outside of the application's direct control. If a resource unexpectedly changes during a computation, an application may observe multiple, inconsistent states of the resource, leading to incorrect program behavior.

This paper presents a sound and precise static analysis, LEGATO, that verifies programs correctly handle changes in external resources. Our analysis ensures that every value computed by an application reflects a single, consistent version of every external resource's state. Although consistent computation in the presence of concurrent resource updates is fundamentally a concurrency issue, our analysis relies on the novel *at-most-once* condition to avoid explicitly reasoning about concurrency. The at-most-once condition requires that all values depend on at most one access of each resource. Our analysis is flow-, field-, and context-sensitive. It scales to real-world Java programs while producing a moderate number of false positives. We applied LEGATO to 10 applications with dynamically updated configurations, and found several non-trivial consistency bugs.

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

Keywords and phrases Static Analysis, Dynamic Configuration Updates

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2018.24

Supplement Material ECOOP Artifact Evaluation approved artifact available at
<http://dx.doi.org/10.4230/DARTS.4.3.2>

Funding This paper is based upon work sponsored in part by DARPA under agreement number FA8750-16-2-0032.

Acknowledgements The authors would like to thank Doug Woos, Chandrakana Nandi, Emina Torlak, Manuel Fahndrich, Francesco Logozzo, and Christian Kästner for their comments on early drafts of this work. We would also like to thank the anonymous reviewers for their feedback.

1 Introduction

Programs are no longer monolithic collections of code. In addition to source code, modern applications consist of configuration files, databases, network resources, and more. Treating these *external resources* as static inputs to the program is infeasible for adaptable, long



© John Toman and Dan Grossman;

licensed under Creative Commons License CC-BY

32nd European Conference on Object-Oriented Programming (ECOOP 2018).

Editor: Todd Millstein; Article No. 24; pp. 24:1–24:32

Leibniz International Proceedings in Informatics



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



```

1 if(hasReadPermission("harmless_file")) {
2     open("harmless_file").read();
3 }

```

■ **Figure 1** Example time-of-check-to-time-of-use bug caused by a dynamic resource update: if "harmless_file" is replaced with a symlink to another user's file after the permissions check but before the `open()` call, a leak of another user's private information will occur.

running programs. Remote hosts may become unavailable or change their APIs, database entries may be changed by other threads or programs, the filesystem may be changed by other tenants on the program's host, and users may update the configuration options of the program. We refer to these changing, evolving resources as *dynamic* external resources; together, these dynamic external resources form an application's view of the dynamic environment in which it executes.

Correctly handling changes in these external resources is challenging. If a dynamic resource is changed between two accesses used in the same computation, a program can observe two or more inconsistent versions of the resource state, which can lead to arbitrary and often incorrect behavior. For example, Figure 1 contains a program fragment exhibiting a well-known time-of-check-to-time-of-use defect [36, 6, 31] that can lead to a malicious user circumventing filesystem permissions. The attack is possible precisely because the code in question can observe two versions of the filesystem state: specifically, two different versions of the read permission. Such issues are not restricted to filesystems; similar problems can be found in applications that interact with databases with multiple users [1] or that support instantaneous configuration updates [43].

Further complicating matters, unlike state under sole control of the program, external resources are often mutated by other programs or users of the system without warning and it is often impossible for the application to prevent such changes. Errors due to dynamic resource updates are difficult to anticipate ahead of time, and (like concurrency errors) require difficult-to-write functional tests to manually uncover. Further, although the example shown in Figure 1 can be detected with a simple syntactic analysis, the dynamic resource errors we have found in practice often involve multiple levels of indirection through the heap and flows through multiple method calls. There has been extensive work to help programmers contend with and correctly handle these changes [4, 31, 9, 5]. However, existing techniques take a piecemeal approach tailored to a specific resource type (e.g., files [36, 6], configuration options [43], etc.).

This paper presents a unified approach to verify that programs always observe consistent versions of external resource state. Key to our approach is the *at-most-once* condition. The at-most-once condition states that a value may depend on at most one access of each external resource. Intuitively, programs observe inconsistent resource states when a resource changes between two or more related accesses of a resource. By restricting all computations to at most one access per resource, the condition guarantees that every value computed by the program always reflects some consistent snapshot of each resource's state.

We efficiently check this condition for complex, real-world programs using a novel static analysis. Conceptually, our analysis versions the external resources accessed by a program so that each read of a resource is assigned a unique version. Our analysis tracks these versioned values as they flow through the program and reports when two or more distinct versions flow to the same value. Although our analysis focuses on errors caused by concurrent changes it does *not* explicitly reason about concurrency involving external updates. Our


```

1 int getDoubled() {
2   return Config.get("number") +
3     Config.get("number");
4 }

```

■ **Figure 2** Example of inconsistency due to dynamically updated configuration options. If the "number" configuration option changes between the two calls to `Config.get()`, a non-even number may be returned.

```

1 int a = Config.get("number");
2 int b = 0;
3 while(★) {
4   b += a;
5 }

```

■ **Figure 3** Example of a resource used multiple times after being read. ★ represents a side-effect free, uninterpreted loop condition. This use pattern is correct because the "number" resource is accessed only once in computing `b`.

analysis is interprocedural and scales to large programs. The analysis is flow-, field-, and context-sensitive, and can accurately model dynamic dispatch.

We implemented the LEGATO¹ analysis as a prototype tool for Java programs. We evaluated LEGATO on 10 real-world Java applications that use dynamic resources. These applications were non-trivial: one application in our evaluation contains over 10,000 methods. LEGATO found 65 bugs, some of which caused serious errors in these applications. Further, we found that the at-most-once condition is a good fit for real applications that use external resources: violations of the at-most-once condition reported by our analysis often corresponded to bugs in the program. LEGATO had a manageable ratio of true and false positives. Our tool is also efficient: it has moderate memory requirements, and completed in less than one minute for 6 out the of 10 applications in our benchmark suite.

In summary, this paper makes the following contributions:

- We define the at-most-once condition, a novel condition to ensure consistent usage of external resources (Section 2).
- We present a novel static analysis for efficiently checking the at-most-once condition (Sections 3 and 4).
- We describe LEGATO, an implementation of this analysis for Java programs (Section 5).
- We show that LEGATO can find real bugs in applications that use dynamic resources (Section 6).

2 At-Most-Once Problems

LEGATO targets programs that use *dynamic external resources*. Unlike static program resources (e.g., program code, constant pools, etc.) dynamic resources are statically identifiable entities that may be changed without warning by code or programs outside of an application's control. In the presence of external changes, programs may observe inconsistent versions of an external resource's state.

For example, Figure 2 shows a (contrived) example of an error due to dynamically updated configuration options. Although callers of `getDoubled()` would reasonably expect the function to always produce an even number, an update of the "number" option between the two calls to `Config.get()` may result in an odd number being returned. This unexpected behavior occurs because the application observes inconsistent versions of "number". The time-of-check-to-time-of-use error in Figure 1 from the introduction is another example.

One possible technique for detecting these errors is to concretely model dynamic resource updates and reason explicitly about update/access interleavings. Unfortunately, explicitly

¹ LEGATO is open-source, available at <https://github.com/uwplse/legato>

modeling concurrency, e.g., [11, 3, 10], is intractably expensive on large programs or requires specific access patterns [25, 27].

LEGATO instead verifies consistent usage of dynamic resources without explicitly reasoning about concurrent updates and reads. In the worst case, a resource may change between *every* access; i.e., every access may yield a unique version of the resource. For example, suppose that the configuration accessed in Figure 2 is updated by another thread in response to user input. In the presence of non-deterministic thread scheduling and without prior synchronization between the two accesses of "number" on lines 2 and 3, the option may be updated some arbitrary number of times. The current implementation of `getDoubled` correctly handles updates that occur before or after the two accesses: only interleaved updates are problematic.

A key insight of LEGATO is that a program that is correct under the worst-case resource update pattern described above will necessarily be correct under *any* update pattern. Further, under the assumption that every access yields a distinct version of the underlying resource, values from two or more different accesses of the same resource can never be combined without potentially yielding an inconsistent result. It is therefore sufficient to verify that a value depends on *at most one* access to each resource. Verifying this condition for all values in a program is the *at-most-once problem*.

The at-most-once problem places no restrictions on the number of times a resource may be *used* once read, nor how many times a resource may be accessed, only on how many times the resource may be accessed in computing a single value. For example, the code in Figure 3 is correct according to our definition of at-most-once. When the "number" option is read on line 1 it reflect a single, consistent version of the option at the time of read. Although "number" may be updated an arbitrary number of times as the loop executes, the value of `a` is unaffected by these updates and remains consistent as it is used multiple times during the execution of the loop. As a result, after the loop finishes, the value of `b` will reflect a consistent version of the "number" option. If the body of the loop was `b += Config.get("number")`, the at-most-once requirement would be violated.

3 The Legato Analysis

LEGATO is a whole-program dataflow analysis for detecting at-most-once violations in programs that use dynamic resources. For ease of presentation, throughout the rest of this paper, we assume that there is only one resource of interest that is accessed with the function `get()`. The analysis described here naturally extends pointwise to handle multi-resource scenarios. Conceptually, the analysis operates by assigning a globally unique, abstract version to the values returned from each resource access. If two or more unique versions flow to the same value, this indicates that a resource was accessed multiple times, thus violating at-most-once.

In a dynamic setting, every read of a resource can be tagged with an automatically incrementing version number. With this approach, detecting violations of at-most-once is straightforward: when two or more different version numbers reach the same value, at-most-once *must* have been violated. This is the approach taken by Staccato [43], which finds inconsistent usage of dynamic configuration options. However, concrete version numbers do not translate to the static setting.

In place of concrete numbers, resource versions can be *abstractly* represented by the site at which a resource was accessed and the point in the program execution that the resource occurred. The presence of uninterpreted branch and loop conditions makes it impossible

to determine the absolute point in a program execution at which a resource access occurs. Instead, LEGATO uses *abstract resource versions* (ARVs) to encode accesses *relative* to the current point in the program execution. For example, an ARV can represent “the value returned from the 2nd most recent execution of the statement s ”, which precisely identifies a single access while remaining agnostic about the *absolute* point in the program execution the access occurred.

The LEGATO analysis combines a reachability analysis with the abstract domain of ARVs to discover which resource versions flow to a value. The ARV lattice is designed such that the meet of two ARV representing different accesses (and therefore versions) yields \perp , which indicates a possible violation of at-most-once.

We first present a simple intraprocedural analysis that does not support loops, heap accesses, or method calls (Section 3.2). We then extend the approach to handle loops (Section 3.3). The transformers defined by these two sections illustrate the core LEGATO analysis. In principle, this basic analysis could be extended to extremely conservatively handle language features, such as the heap or methods. However, in practice, doing so would result in enormous precision loss. We therefore show how we extend the analysis to field- and flow-sensitively handle information flow through the heap (Section 3.5). Extending the analysis to precisely handle method calls is non-trivial, and is discussed in Section 4. Other program features (e.g., exceptions, arrays, etc.) are straightforward extensions of the ideas presented here.

Abstract Resource Versions. As mentioned above, an abstract resource version (ARV) represents a resource version by the access site and the point in time at which the access was performed. To ensure soundness, values returned from different resource accesses must be assigned unique ARVs (we expand on this point further in Section 3.4). In a simple language with no loops or methods, ARVs are simply expression labels: each label represents the unique value produced by the execution of the labeled expression. In the presence of loops, we augment these labels with a priming mechanism to differentiate between multiple executions of the same expression. To precisely handle methods, in Section 4.1 we generalize to strings of primed labels, which identify an access by the sequence of method calls taken to reach an access site (similar to the approach taken by [48]). Finally, in Section 4.2, we further generalize ARVs to sets of strings (represented as tries) to encode multiple possible accesses that may reach a program value. However, even with this representation, our analysis always maintains the invariant that each ARV abstracts a single, unique resource access.

3.1 Preliminaries

Before describing the analysis, we first briefly review some relevant background information.

IDE. The LEGATO analysis uses the IDE (Interprocedural Distributive Environment) program analysis framework [40]. The IDE framework can efficiently solve program analysis problems stated in terms of *environment transformers*. An environment is a mapping from dataflow symbols (e.g., variables) to values. The domain of symbols must be finite, but the domain of values may be infinite, provided the values form a finite-height, complete lattice. The meet of environments is performed pointwise by symbol. IDE analyses assign environment transformers to edges in the program control-flow graph. However, to aid exposition, throughout the remainder of this section we will instead denote *statements* into

```

(const)  c ::= 0 | 1 | ...
(var)    v ::= a | b | ...
(atom)   a ::= c | v
(expr)   e ::= a + a | a | getℓ()
(stmt)   s ::= v = e | s; s | skip
          | if a then s1 else s2

```

■ **Figure 4** Grammar for the loop-, and method-free language.

Statement	Transformer
$\llbracket v_1 = v_2 \rrbracket$	$\triangleq \lambda e. e[v_1 \mapsto e(v_2)]$
$\llbracket v = c \rrbracket$	$\triangleq \lambda e. e[v \mapsto \top]$
$\llbracket v_1 = v_2 + v_3 \rrbracket$	$\triangleq \lambda e. e[v_1 \mapsto e(v_2) \sqcap e(v_3)]$
$\llbracket v = \text{get}^\ell() \rrbracket$	$\triangleq \lambda e. e[v \mapsto \widehat{\ell}]$
$\llbracket \text{skip} \rrbracket$	$\triangleq \lambda e. e$
$\llbracket \text{if } a \text{ then } s_1 \text{ else } s_2 \rrbracket$	$\triangleq \lambda e. (\llbracket s_1 \rrbracket e) \sqcap (\llbracket s_2 \rrbracket e)$
$\llbracket s_1; s_2 \rrbracket$	$\triangleq \lambda e. \llbracket s_2 \rrbracket (\llbracket s_1 \rrbracket e)$

■ **Figure 5** Environment transformers of the basic analysis.

environment transformers.²

The IDE framework targets a specific subclass of analyses where the environment transformers distribute over the meet operator on environments. That is, for all transformers $t : Env \rightarrow Env$ and all environments e_1, e_2, \dots, e_n , $\prod_i t(e_i) = t(\prod_i e_i)$, where equality on environments is defined pointwise. Given a set of distributive environment transformers, the IDE framework produces a flow and context-sensitive analysis with polynomial time complexity.

Access Paths. An *access path* [15, 19] is an abstract description of a heap location. An access path consists of a local variable v , and a (potentially empty) sequence of field names $f.g.h\dots$. Together, these two elements name the location reachable from v through fields f, g, h, \dots . We will write ϵ to represent an empty sequence of fields, π to refer to an arbitrary (potentially empty) sequence of fields, and $v.\pi$ to denote an arbitrary access path.

3.2 The Basic Analysis

We first present our analysis on a limited language described by the grammar in Figure 4. Every call to `get()` is uniquely labeled with ℓ : we will write concrete labels as 1, 2, etc. Our basic language contains no looping constructs, as a result every `get()` expression is executed at most once. Thus, every access can be uniquely identified by the label of a `get()` expression. For this language, the abstract resource versions are `get()` expression labels: $\widehat{\ell}$ represents the

² This change in presentation does not change the behavior of the analysis; the denotation of a statement is a simplification of the meet of the composition of the edge transformers for all paths through a statement.

unique version of the resource returned by the corresponding $\text{get}^\ell()$ expression. Further, at this point the analysis operates on access-paths with no field sequences: we abbreviate $v.\epsilon$ as v .

The basic LEGATO analysis is expressed using the environment transformers in Figure 5. Conceptually, for a program s , the analysis applies the empty environment (i.e., all facts map to \top) to the transformer associated with statement s . Thus, the analysis result is given by $\llbracket s \rrbracket (\lambda_.\top)$. The analysis is standard in its handling of several language features. For instance, sequential composition of statements is modeled by composing environment transformers, and conditional statements are modeled by taking the meet of the environments yielded from both branches.

The interesting portion of the analysis lies in the handling of variable assignments. Assignments overwrite previous mappings in the environment of the left-hand side with the abstract value of the right-hand side. Integer constants are never derived from resources, and therefore have the abstract value \top , which represents any value not derived from a resource. The statement $v_1 = v_2$ associates v_1 with the abstract version contained in v_2 . Resource accesses have the abstract value $\hat{\ell}$ which, as discussed above, is sufficient to uniquely identify the value returned from the access $\text{get}^\ell()$. This simplified version of the LEGATO analysis is very similar in style to a constant propagation analysis, where in place of integers or booleans, the constants of interest are abstract resource versions.

Values may become inconsistent for two reasons. The first is due to the addition operator. The expression $v_1 + v_2$ is given the abstract value $e(v_1) \sqcap e(v_2)$. The meet operator for these ARVs is derived from a flat lattice:

$$\top \sqcap x = x \quad x \sqcap \perp = \perp \quad \hat{i} \sqcap \hat{i} = \hat{i} \quad \hat{i} \sqcap \hat{j} = \perp, \text{ if } i \neq j$$

where i and j are two arbitrary labels. If $e(v_1) = \hat{i}$ and $e(v_2) = \hat{i}$, then the result of the addition still depends only on the resource accessed at $\text{get}^i()$. In this case, at-most-once is not violated, and the meet yields \hat{i} as the abstract value of the overall expression. However, if $e(v_1) = \hat{i}$ and $e(v_2) = \hat{j}$ then the program is combining two unique versions of the resource, which violates at-most-once. The meet of these two incompatible versions yields \perp , which is the “inconsistent” value in the lattice.

Finally, a variable may be assigned \perp due to LEGATO’s conservative handling of conditional statements. Recall that the environments produced by the two branches of a `if` statement are met at the control-flow join point of the conditional. Thus, if a variable x is mapped to two distinct, non- \perp values in environments produced by different branches of a conditional, those values will be met yielding \perp . In this case, the result of \perp does not correspond to a violation of at-most-once, and is a false positive. The alternative, full path-sensitivity, is unacceptably expensive. We do support a limited form of path-sensitivity to precisely model dynamic dispatch (Section 4.2).

3.3 Loops

The simple analysis presented so far is no longer sound if we extend the language with loop statements:

```
stmt ::= ... | while a do s end
```

If a $\text{get}^\ell()$ expression is in a loop, each evaluation of $\text{get}^\ell()$ must be treated as returning a unique version. However, the transformers presented in the previous section effectively assume $\text{get}^\ell()$ always returns the same version. We therefore extend the transformers and lattice to distinguish resource accesses from distinct iterations of an enclosing loop.

```

1 a = 1; // a:  $\top$ 
2 b = get1(); // b:  $\hat{1}$ 
3 c = get2(); // c:  $\hat{2}$ 
4 d = a + b; // d:  $\hat{1}$ 
5 e = c + d; // e:  $\perp$ 

```

```

1 while * do
2   b = a; // a:  $\hat{1}$ , b:  $\hat{1}$ 
3   a = get1() // a:  $\hat{1}$ , b:  $\hat{1}'$ 
4 end
5 c = a + b; // c:  $\perp$ 

```

■ **Figure 6** Results of the basic analysis. The comments on each line show the abstract value assigned to the variable assigned on that line.

■ **Figure 7** Example of priming due to loops. The abstract values shown in comments are derived after executing the loop once. On line 3 LEGATO primes the abstract value of `b` to distinguish it from the fresh value returned by `get1()`.

In a dynamic setting, we could associate every resource access `getℓ()` with a concrete counter c_ℓ incremented on every execution of `getℓ()`. In this (hypothetical) scenario, `getℓ()` yields the abstract version, $\langle \ell, c_\ell \rangle$: by auto-incrementing c_ℓ the analysis ensures executions of `getℓ()` from different iterations are given unique abstract versions.

This straightforward approach fails in the static setting: without *a priori* knowledge about how many times each loop executes, the analysis would fail to terminate. We introduce *priming* to address this issue. A primed `get()` label $\hat{\ell}^n$ represents the $n+1^{\text{th}}$ most recent access of the resource at `getℓ()`. For example, $\hat{1}''$ (i.e., $\hat{1}^2$) represents the unique value produced by the third most recent evaluation of `get1()`, and $\hat{2}$ (i.e., $\hat{2}^0$) is the value returned from the most recent evaluation of `get2()`. Abstract versions with the same base label but differing primes are considered unique from one another in the lattice, i.e., $\hat{i}^n \sqcap \hat{j}^m = \perp \iff i \neq j \vee m \neq n$. Thus, this domain distinguishes between accesses due to different `get()` expressions as well as different invocations of the *same* `get()` expression.

The syntactic structure of loops are handled using a standard fixpoint technique. The addition of loops changes how $v = \text{get}^\ell()$ statements are handled in the analysis. As before, the variable v is assigned the abstract value $\hat{\ell}$. In addition, a prime is added to all *existing* abstract values with the base label ℓ . We extend the environment transformer for the $v = \text{get}^\ell()$ case in Figure 5 as follows:

$$\lambda e. \lambda v'. \begin{cases} \hat{\ell} & \text{if } v \equiv v' & (1) \\ \hat{\ell}^{n+1} & \text{if } e(v') \equiv \hat{\ell}^n & (2) \\ e(v') & \text{o.w.} & (3) \end{cases}$$

In other words, the $v = \text{get}^\ell()$ statement creates a new environment³ such that:

1. v maps to $\hat{\ell}$, i.e., the most recent version returned from `getℓ()`
2. Variables besides v that map to the base label ℓ have a prime added, indicating these values originate one more invocation of `getℓ()` in the past
3. All other variables retain their value from *env*

A program illustrating this behavior is shown in Figure 7.

Termination. It is not obvious that the above environment transformer will not add primes forever. We therefore informally argue for termination.

Let us consider the simple case with a single loop and one call to `get()` labeled ℓ . Variables that are definitely not assigned a value from `getℓ()` will not be primed and therefore do not

³ Recall that an environment is a mapping of symbols (in this case, variables) to abstract values: the function term $\lambda v'. \dots$ is such an environment.

affect achieving fixpoint. For variables to which $\text{get}^\ell()$ may flow, the flow occurs along some single chain of assignments, e.g. $a = \text{get}^\ell(); b = a; c = b; \dots$. If instead the assignment occurred along multiple possible chains, the conservative handling of conditionals will yield \perp , ensuring the analysis achieves fixpoint.

Consider now the case where some assignments in the chain occur *conditionally*, e.g.:

```

1 while * do
2   if * then b = a else skip;
3   a = get1()
4 end

```

where $*$ represents uninterpreted loop and branch conditions. In this example, b receives the value of some arbitrary previous invocation of $\text{get}^1()$. Our domain of primed labels cannot precisely represent this value, but the analysis will conservatively derive \perp for b , again ensuring the analysis achieves fixpoint. After two iterations of the analysis, two possible values for b , $\widehat{1}$ and $\widehat{1}'$, will flow to line 3 from the two branches of the conditional on the previous line. The meet of these two values is \perp .

The last case to consider in the single loop case is a chain of definite assignments from $\text{get}^\ell()$ to some variable v . For some chain of length k , it is easy to show that the resource will propagate along the chain in at most k analysis iterations. Thus, the resource will flow over the $\text{get}()$ expression at most k times, and receive at most k primes. After fully propagating along the chain, the value in v will not receive further primes: on further iterations of the analysis the value in v is killed by the previous definite assignment in the chain.

Finally, we consider nested loops. As a representative case, consider the following scenario:

```

1 while * do
2   b = a;
3   while * do a = get1() end
4 end

```

Other possible combinations of assignments and nesting generalize straightforwardly from this example. After the first pass through the outer-loop, the environment produced is $[a \mapsto \widehat{1}]$. On the second pass, the environment that reaches line 3 is $[a \mapsto \widehat{1}, b \mapsto \widehat{1}]$. One further iteration of the inner-loop produces $[a \mapsto \widehat{1}, b \mapsto \widehat{1}']$. The meet of this environment with the previous input environment on line 3 assigns b the value \perp , ensuring a fixpoint is reached.

An alternative approach would be to artificially limit the number of primes on a label to some small constant k . However, we decided against choosing an *a priori* bound for the number of primes lest this bound introduce false positives. However, we found in practice we needed at most 2 primes for the programs in our evaluation set. This finding is consistent with Naik's experience with abstract loop vectors [35], which are similar to our priming approach.

3.4 Soundness

We have proved that the core analysis presented is sound. We first defined an instrumented concrete semantics that: 1) assigns to each value returned from $\text{get}()$ a unique, concrete version number, and 2) for each value, collects the set of concrete resource versions used to construct that value. The concrete semantics only considers direct data dependencies when collecting the versions used to construct a given value. We define soundness in relation to these concrete semantics. The LEGATO analysis is sound if, whenever variable is derived from multiple concrete versions in any execution of the instrumented semantics, the analysis derives \perp for that variable. As our concrete semantics uses only direct dependencies for

Statement	Transformer
$\llbracket v.f = c \mid \text{null} \rrbracket$	$\triangleq \lambda env.env[pref(v.f) \mapsto \top]$
$\llbracket v = \text{null} \mid \text{new } T \mid c \rrbracket$	$\triangleq \lambda env.env[pref(v) \mapsto \top]$
$\llbracket v_1 = v_2.f \rrbracket$	$\triangleq \lambda env.env[pref(v_1) \mapsto \top, v_1.\pi \mapsto env(v_2.f.\pi)]$
$\llbracket v_1 = v_2 \rrbracket$	$\triangleq \lambda env[pref(v_1) \mapsto \top, v_1.\pi \mapsto env(v_2.\pi)]$

■ **Figure 8** New environment transformers for the heap. The $pref(x)$ function yield the set of all access paths in e with x as a prefix. In addition, all references π are implicitly universally quantified.

collecting version numbers, our soundness claim is only with respect to such dependencies and ignores information propagated via control-flow. We discuss this reasons for this choice further in Section 5.4.

Our proof of soundness relies on a *distinctness invariant*: two variables have different abstract resource versions if they have different concrete version numbers under the concrete semantics. In other words, when two variables have the same abstract version, they *must* be derived from the same resource access in all possible program executions. Thus, the invariant ensures that when values derived from different concrete resource versions are combined by a program, the analysis will take the meet of distinct abstract resource versions yielding \perp . The converse is also true: if two values with the same abstract resource version are combined, then no program execution will combine two values derived from distinct resource accesses.

The justifications given above for the environment transformers and analysis domain provide intuitive arguments for why this invariant is maintained. The full proofs and concrete semantics are omitted for space reasons: they are included in the appendix of the paper. Although our proof is stated only for the simple intraprocedural analysis presented so far, when we extend the analysis to support methods in Section 4 we provide an argument for the preservation of the distinctness invariant.

3.5 Fields and the Heap

We now consider a language with objects and fields.

$$\begin{aligned}
 expr & ::= \dots \mid \text{new } T \mid v.f \\
 atom & ::= \dots \mid \text{null} \\
 stmt & ::= \dots \mid v.f = a
 \end{aligned}$$

A subset of the new environment transformers for the heap language are given in Figure 8. In this version of the language, our transformers operate on access paths with non-empty field sequences as opposed to plain variables. These environment transformers encode the effect of each statement on the heap: for example, constants, `null`, and `new` expressions on the right hand side of an assignment “kill” access-paths reachable from the left-hand side.

There are two statement forms that require special care that do not appear in Figure 8. First, LEGATO handles assignments with a `get()` right-hand side with the environment transformer from Section 3.3 extended to support access paths instead of variables. For an

assignment of the form $v = \text{get}^\ell()$, LEGATO uses the following transformer:⁴

$$\lambda e. \lambda \langle v'.\pi \rangle. \begin{cases} \widehat{\ell} & \text{if } v' \equiv v \\ \widehat{\ell}^{n+1} & \text{if } e(v'.\pi) \equiv \widehat{\ell}^n \wedge v' \not\equiv v \\ e(v'.\pi) & \text{o.w.} \end{cases}$$

The second statement form, heap *writes* such as $v_1.f = v_2$, is handled conservatively. LEGATO uses strong updates only for access paths with the $v_1.f$ prefix. After the heap-write, the abstract value reachable from some access path $v_1.f.\pi$ is precisely the value reachable from $v_2.\pi$. However, an access path that only *may* alias with $v_1.f$ is weakly updated. A weak update of the access path $v_3.\pi'$ to the abstract value $\widehat{\ell}^n$ takes the meet of the current value of $v_3.\pi'$ with $\widehat{\ell}^n$. Given the definition of the label lattice, this treatment of weak updates means that an access-path $v.\pi$ “updated” via aliasing cannot be updated at all: the new value must exactly match the existing value of the access-path or the access-path may have no value at all, represented by \top .

Formally, LEGATO assigns a heap write statement $v_1.f = v_2$ the environment transformer:

$$\lambda e. \lambda \langle v.\pi \rangle. \begin{cases} e(v_2.\pi') & \text{if } v.\pi \equiv v_1.f.\pi' \\ e(v_2.\pi') \sqcap e(v.\pi) & \text{if } v.\pi \not\equiv v_1.f.\pi' \wedge \text{mayAlias}(v.\pi, v_1.f.\pi') \\ e(v.\pi) & \text{o.w.} \end{cases}$$

Resolving the *mayAlias* query is an orthogonal concern to the LEGATO analysis. In our implementation we use an off-the-shelf, interprocedural, flow- and context-sensitive may alias analysis (Section 5.1).

4 Interprocedural Analysis

The interprocedural version of LEGATO is a non-trivial extension of the intraprocedural analysis from the previous section. There are two main extensions to the core analysis. First, LEGATO soundly accounts for *transitive* resource accesses. A transitive resource access refers to when a method $m()$ returns the result of an invocation of $\text{get}^\ell()$; the analysis must distinguish between abstract values produced by separate invocations of $m()$. In addition, to analyze realistic Java code, LEGATO precisely models dynamic dispatch. If a method call $m()$ may dynamically dispatch to one of several possible implementations, LEGATO soundly combines the unique abstract values returned by each implementation without sacrificing precision.

Definitions. For presentation purposes only, we begin by making the simplifying assumption that all methods are static (i.e., all call sites have one unique callee), a method has a single formal parameter p , all methods end with a single `return` statement, and method calls are always on the right hand side of an assignment. We extend the grammar for expressions and statements as follows:

$$\text{expr} ::= \dots \mid m^k(a) \qquad \text{stmt} ::= \dots \mid \text{return } a$$

⁴ In our formalism, we assume that $\text{get}()$ returns a primitive value, and thus the environment will only contain mappings for $v.\epsilon$.

Transformers for a method invocation: $v_1 = m^k(v_2) \rightarrow m(p)\{\dots; \text{return } r\}$		
Call-to-start	Exit-to-return	Call-to-return
$\lambda e.\lambda\langle v' . \pi \rangle \begin{cases} e(v_2.\pi) & \text{if } v' \equiv p \\ \top & \text{o.w.} \end{cases}$	$\lambda e.\lambda\langle v' . \pi \rangle \begin{cases} \rho(e(p.\pi)) & \text{if } v' \equiv v_2 \\ \quad \wedge \pi \neq \epsilon & \\ \rho(e(r.\pi)) & \text{if } v' \equiv v_1 \\ \top & \text{o.w.} \end{cases}$	$\lambda e.\lambda\langle v' . \pi \rangle \begin{cases} \top & \text{if } v' \equiv v_1 \\ \top & \text{if } v' \equiv v_2 \\ \quad \wedge \pi \neq \epsilon & \\ \tau(e(v' . \pi)) & \text{o.w.} \end{cases}$

■ **Figure 9** Interprocedural environment transformers. The names of the columns correspond to the transformer names in the original IDE paper. ρ is a function that transforms values that flow out of a method. τ transforms values propagated over method calls. We will define these methods later in the section. The analysis allows for strong updates to heap locations reachable from the argument of a method, although the base variable retains its value from the caller.

```

1 m () {
2   while * do a = get1() end;
3   return a
4 }
5 b = m2();
6 c = m3()

```

■ **Figure 10** A non-trivial interprocedural resource access.

All method calls are labeled: these sets of labels do not overlap with `get()` expression labels. We will continue to use ℓ to denote an arbitrary `get()` expression label, and k to denote a call site label. We will use the same notation for method call labels used in ARVs (i.e., $\hat{1}$) as we did for `get()` labels in Section 3: context will make clear which type of label we mean.

The interprocedural environment transformers used by LEGATO are mostly standard in the mapping of dataflow symbols into and out of methods. For a method call $v_1 = m(v_2)$ to $m(p)\{\dots\}$, the access-path $v_2.\pi$ in the calling context is mapped to $p.\pi$ in the callee method. Dataflow symbols that flow out of a method call (via heap locations reachable from formal arguments, or return statements) are mapped back into the caller environment. Finally, information local to the caller that does not flow through the method call to m is propagated over the method call.⁵ LEGATO’s analysis is non-standard only in how values are transformed across method boundaries. Values that flow out of a method are transformed by the function ρ and values propagated over a method call are transformed by τ . We define these functions in this section. The full environment transformers are given in Figure 9.

4.1 Transitive Resource Accesses

In a language with methods, a single primed `get()` label is no longer sufficient to uniquely identify a resource access at some point in time. Consider the code sample in Figure 10. After line 5, the value in `b` comes from the most recent invocation of `get1()`. However, after `m` is called again on line 6, the value in `b` comes from an execution of `get1()` at some arbitrary point in the past. A single-primed label is unable to represent this situation. Leaving the value of $\hat{1}$ in `b` after the second call to `m` is unsound, and using the \perp value is imprecise. In general, transitive resource access may occur any arbitrary depth in the call-graph.

⁵ For readers familiar with the IDE framework, these three components correspond to the call-to-start, exit-to-return-site, and call-to-return-site transformers respectively.

To precisely handle scenarios like the one in Figure 10, LEGATO generalizes the primed label ARV into *strings* of such labels. Unlike a single `get()` label, which identifies resource accesses relative to the current point in a programs execution, call-strings encode resource accesses relative to *other* program events: specifically, method invocations. For example, in the above example, the abstract resource version stored in `b` can be precisely identified by “the most recent invocation of `get1()` that occurred during the most recent invocation of `m` at call site 2”. The call-strings used as ARVs can precisely encode statements of this form.

A call-string takes the form $\widehat{k}_1^p \cdot \widehat{k}_2^q \cdots \widehat{k}_m^r \cdot \widehat{\ell}^n$, where $\widehat{\ell}$ is a primed `get()` label and each \widehat{k}_i is a primed call site label. Call-strings are interpreted recursively; $s \cdot \widehat{k}^n$ represents the $(n + 1)^{th}$ most recent invocation of $m^k()$ relative to the program point encoded in the prefix s . The string $s \cdot \widehat{\ell}^n$ has an analogous interpretation. If s is the empty string, the label is interpreted relative to the current point of execution. For example, the resource stored in `b` from Figure 10 can be represented by the ARV $\widehat{2} \cdot \widehat{1}$, which has the interpretation given above. As in the intraprocedural analysis, two distinct call-strings encode different invocations of a resource access, and thus their meet returns bottom. The lattice on call-strings is a constant, flat lattice on call-strings, which is a natural generalization of the lattice on individual labels.

When a value with call-string s flows out of a method m from the invocation $m^k()$, \widehat{k} is prepended onto the string s . In other words, for a method call $v = m^k()$, $\rho \triangleq \lambda s. \widehat{k} \cdot s$. The prepended label encodes that the access represented by s occurs relative to the most recent invocation of m at k . Prepending \widehat{k} also distinguishes transitive accesses that occurred while executing $m^k()$ from those resulting from other calls of $m()$.

The intraprocedural fragment of LEGATO remains primarily unchanged. Transitive resource accesses within a loop are handled with a priming mechanism similar to the one used for `get()` expressions. A string with \widehat{k} at the head that is propagated over the method call $v = m^k()$ has a prime added to \widehat{k} . We define propagate over method transformer as:

$$\tau \triangleq \lambda s. \begin{cases} \widehat{k}^{n+1} \cdot s' & \text{if } s \equiv \widehat{k}^n \cdot s' \\ s & \text{o.w.} \end{cases}$$

The justification for this transformation is identical to the one provided for values that flow over `get()` invocations. The added prime indicates any accesses that occurred relative to $m^k()$ now originate one more invocation of $m^k()$ in the past. Recursion is treated conservatively but does not require special handling in our analysis. Two iterations of the analysis through a recursive cycle will generate two strings, s and $s \cdot s$, the meet of which is \perp , ensuring fixpoint.

Soundness. We now informally argue for the soundness of the above approach. Recall from Section 3.4 that the soundness of LEGATO relies on a distinctness invariant, which states that if two values are derived from distinct resource accesses LEGATO must assign different abstract resource versions to those values. To simplify the following argument, we will assume that only a single value is returned from the callee via a `return` statement (the argument for values returned via the heap generalizes naturally from the following).

Let us assume the distinctness invariant holds for all values in the caller and callee environments, i.e., values from different invocations of `get()` are assigned different ARVs. Let us then show the invariant holds after the callee returns to the caller. First, it is immediate that the call-to-return transformer τ preserves distinctness for values in the caller environment. Next, suppose the value returned from the callee is derived from some resource access that occurred during the execution of the callee. To preserve the invariant, we must then show that the returned value is given a distinct ARV in the caller. By prepending the label of the call site and priming all ARVs that already contain that label, distinctness is ensured.

4.2 Dynamic Dispatch and Path-Sensitivity

LEGATO is not path-sensitive in general; as mentioned in Section 3.2 the abstract value of a variable from multiple branches are met at control-flow join points potentially yielding false positives. A key exception is LEGATO’s handling of dynamic dispatch. In Java and other object-oriented languages, a method call m may dispatch to different implementations depending on the runtime type of the receiver object. In general, it is impossible to predict the precise runtime type of the receiver object for every call site, so a program’s static call-graph has edges to every possible implementation m_1, m_2, \dots, m_n of m at the call site $m^k()$. If LEGATO treated multiple return flows like control-flow constructs such as `if` and `while`, the analysis would be sound but unacceptably imprecise.

LEGATO handles dynamic dispatch path-sensitively by aggregating results from each distinct concrete callee into a single, non- \perp ARV. Although the resulting ARV encodes multiple, potentially incompatible resource accesses, LEGATO ensures that all accesses represented by the ARV come from different concrete callees of a single virtual call site. As only one concrete callee is invoked per execution of a virtual call site, only one access represented in an ARV may be realized at runtime. Thus, combining results from different concrete implementations into a single ARV does not allow for violations of at-most-once.

Multiple resource accesses are represented by generalizing the call-string representation from the previous subsection into tries, which encode *sets* of call-strings. Leaf nodes of the trie are labeled with primed `get()` labels, and interior nodes with primed call site labels. The children of a call site node labeled \widehat{k}^n represent the possible results returned from the $(n+1)^{th}$ most recent invocation of the call site with label k . A path through the trie implicitly defines a call-string with the same interpretation as given in Section 4.1. The call-string representation of the previous subsection is a degenerate case of the trie representation where each node has only one child.

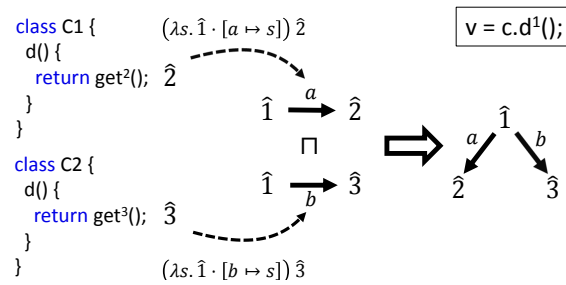
Formally, we write $\widehat{k}^n \cdot [b_1 \mapsto t_1, b_2 \mapsto t_2, \dots]$ to represent a call site node \widehat{k}^n with children t_1, t_2, \dots reachable along branches with ids b_1, b_2, \dots . The branch ids are unique within each call site node and correspond to a potential callee. We call the branch id to child mapping the *branch map*, and write \mathcal{M} to denote an arbitrary mapping.

We extend the return transformer ρ as follows. On return from a concrete implementation m_p to the call site $m^k()$, $\rho \triangleq \lambda s. \widehat{k} \cdot [p \mapsto s]$. That is, the ARV s is extended with a new call site root node labeled \widehat{k} that has a single child with branch id p . In the caller, these single-child ARVs are aggregated into a single node that represents all possible results from each callee. Similarly, we update the function τ as follows:

$$\tau \triangleq \lambda s. \begin{cases} \widehat{k}^{n+1} \cdot \mathcal{M} & \text{if } s \equiv \widehat{k}^n \cdot \mathcal{M} \\ s & \text{o.w.} \end{cases}$$

Combining ARVs from *different* invocations of the same virtual call site or different call sites yields \perp . To combine ARVs representing results from the *same* invocation of a call site, the branch maps of the ARVs are met pointwise by branch id. As is standard, unmapped branch ids in either map are assumed to have the value \top . However, if the meet of any branch is \perp then the entire meet operator yields \perp . That is, a violation of at-most-once in one possible callee yields an overall inconsistent result. An example return flow and meet is shown in Figure 11. Formally, the full meet operator for trie ARVs is as follows:

$$\widehat{i}^n \cdot \mathcal{M}_1 \sqcap \widehat{j}^p \cdot \mathcal{M}_2 = \begin{cases} \widehat{i}^n \cdot \mathcal{M}' & \text{if } i = j \wedge n = p \wedge \mathcal{M}' \neq \perp \text{ where } \mathcal{M}' \triangleq \mathcal{M}_1 \sqcap \mathcal{M}_2 \\ \perp & \text{o.w.} \end{cases}$$



■ **Figure 11** Example of LEGATO’s handling of dynamic dispatch. $v = c.d^1()$ may dispatch to either implementation in C1 or C2. The dashed lines illustrate the return flows, and are annotated with the return flow function applied by the analysis. The two single-child ARVs are met to produce the trie on the right. a and b are the branch ids assigned to the callees C1.d and C2.d respectively.

4.3 Effectively Identity Flows

Prepending labeled nodes on *all* return flows can cause imprecision. For example, consider:

```

1 idA(i) { return i }
2 idB(j) { return j }
3 x = get1();
4 y = id2(x)

```

where `id` may dispatch to one of `idA` or `idB`. In this example, x is assigned $\hat{1}$ and y is assigned $\hat{2} \cdot [a \mapsto \hat{1}, b \mapsto \hat{1}]$. According to the lattice, these two values are distinct and may not be safely combined, despite being identical. This issue arises because the invocation of `id` is unnecessary to identify the resource access that flows to y , nor does the behavior of the two possible callees of `id` differ. We call a scenario like the above an *effectively identity flow*.

LEGATO handles effectively identity flows by detecting when the standard meet operator would produce \perp , and refining the ARVs to eliminate any effectively identity flows. Call-site nodes are added on return from a method invocation $m()$ to either identify transitive resource accesses (Section 4.1) or to differentiate behavior of multiple callees at $m()$ (Section 4.2). Conversely, if all callees exhibit the same behavior and no transitive resource accesses occur within the call $m()$, call site nodes added on return flow from $m()$ are, by definition, redundant. LEGATO *cannot* add labels on return only when necessary to disambiguate different resource accesses. Such an approach would require non-distributive environment transformers, which are unsuitable for use with the IDE framework upon which LEGATO is built.

Based on this intuitive definition of effectively identity flows, we define a refinement operation \mathcal{R} , which traverses the ARV trie, and iteratively removes redundant nodes. After the operation is complete, only the nodes and corresponding labels necessary to either distinguish a resource access or differentiate multiple callees’ behavior are left in the trie. We first formally define effectively identity flows (EIF) and initial refinement operation \mathcal{R}_0 for the single dispatch case (Section 4.3.1). The definitions of EIFs and the full refinement operation, \mathcal{R} , for dynamic dispatch (Section 4.3.2) build upon these definitions.

4.3.1 Effectively Identity Flows and Single Dispatch

As a simplification, we consider call-strings with no primes: the operations and sets defined here can be easily extended to ignore primes on call labels. For every method m , let $\mathcal{AS}(m)$ denote the set of transitively reachable, unprimed, call site and `get()` labels of m . Further, for each call site label k we denote the method invoked at k as \mathcal{CS}_k . A call-string s contains

an EIF if there exists a suffix $\widehat{k} \cdot s'$ such that there exists a \widehat{j} in s' such that $j \notin \mathcal{AS}(\mathcal{CS}_k)$. The existence of \widehat{j} indicates that the ARV must have been returned out of some method other than those called by \mathcal{CS}_k , and, by definition, the access represented by the ARV must therefore have occurred in some method other than those called by \widehat{k} . Thus, \widehat{k} is irrelevant for the purposes of identifying the resource access encoded in the ARV.

The initial refinement operation, \mathcal{R}_0 , follows from this definition. Let s be a call-string, \widehat{k} the first label in s involved in an effectively identity flow, and \widehat{j} be defined as above. Finally, let s'' be the suffix of s that starts with \widehat{j} (inclusive). Given these definitions: $\mathcal{R}_0(s) \triangleq \mathcal{R}_0(s'')$. The refinement operation is defined inductively: in the base case where s contains no identity flows the refinement operation is defined to be $\mathcal{R}_0(s) \triangleq s$. Intuitively, the refinement operation iteratively strips off substrings of labels that form effectively identity flows until reaching the suffix of labels that are necessary to distinguish the resource access.

4.3.2 Effectively Identity Flows and Path-Sensitivity

In the presence of ARV branching, we must extend the definition of effectively identity flows presented above. In the single-dispatch case, call site nodes were necessary only to precisely represent transitive accesses; nodes that did not fulfill this purpose could be removed. In the presence of branching, a call site node may also be required to precisely combine otherwise incompatible method call results. Thus, a call site node is part of an effectively identity flow *iff* it is not required to identify accesses within a method call (as before) *and* it does not differentiate two or more otherwise incompatible method results.

We define an effectively identity flow in the presence of branching as follows. Each node encodes a finite set of strings, with each string corresponding to labels on a path from the node to the leaves of the ARV trie. Passing through a call site node \widehat{i} along branch b corresponds to \widehat{i}_b . We will denote the set of call-strings for a node n with n^\natural . Similarly a call-string ARV can be trivially converted into a trie ARV as follows:

$$\llbracket \widehat{i}_b \cdot s \rrbracket \triangleq \widehat{i} \cdot [b \mapsto \llbracket s \rrbracket] \quad \llbracket \widehat{i} \rrbracket \triangleq \widehat{i}$$

Given these definitions, an ARV contains an effectively identity flow if there exists a call site node $n \equiv \widehat{k} \cdot \mathcal{M}$ that satisfies two conditions. First, every call-string $\widehat{k}_b \cdot s \in n^\natural$ contains an effectively identity flow according to the definition in Section 4.3.1 originating at k . In other words, the call site node \widehat{k} is unnecessary to identify any resource accesses within the call at k . The second condition is $\prod_{s \in n^\natural} \llbracket \mathcal{R}_0(s) \rrbracket \neq \perp$. That is, after removing the call site node \widehat{k} , it must be possible to meet the resulting ARVs without producing a violation of at-most-once. For nodes that satisfy this condition, the full refinement operation is: $\mathcal{R}(n) \triangleq \mathcal{R}(\prod_{s \in n^\natural} \llbracket \mathcal{R}_0(s) \rrbracket)$. The base case for nodes that cannot be refined is $\mathcal{R}(n) \triangleq n$. Similarly to the single-dispatch case, the refinement operation traverses the ARV trie, stripping redundant nodes and collapsing redundant branches.

4.4 Application Level Concurrency

The at-most-once condition obviates reasoning about concurrent resource updates, but LEGATO must still account for concurrency within an application. LEGATO is not sound in the presence of data races: we assume that all mutable, shared state is accessed within a lock protected region. Thus, outside of synchronized regions, each thread reads only values previously written by that thread. However, within a synchronization region, a thread may observe values written by *any* other thread. LEGATO conservatively assigns heap locations read in synchronization regions the abstract version $\widehat{\ell}$, where ℓ is a fresh, distinct label.

In other words, synchronization primitives *havoc* the abstract resource versions potentially shared among threads.

5 Implementation and Challenges

We implemented LEGATO as a prototype tool for Java programs. We used the Soot framework [47] for parsing bytecode and call-graph construction. We built the LEGATO analysis on an extended version of the Heros framework [7]. Although we state our analysis in terms of access paths for simplicity of presentation, we actually operate on *access graphs* [21] a generalization of access paths. Access paths can only represent heap locations accessible via a finite number of field references. In contrast, access graphs compactly encode a potentially infinite set of paths through the heap. The analysis presented here extends naturally from access paths to access graphs.

To resolve uses of the Java reflection API, we relied on the heuristics present in the underlying Soot framework. However, we also found all of the applications in our evaluation suite provided a mechanism for one method to invoke another based on an application-specific URL recorded in a static configuration file. We found that, like many uses of Java reflection [46, 2], these mechanisms are almost always used with static strings. Following the technique outlined in [46], where possible we use these strings to statically resolve these implicit calls to a direct call to a single method. When these heuristics fail, we soundly resolve to all possible callees. Unlike the Java reflection API, which must consider all methods/constructors as possible targets, the set of potential callees was small enough that this over-approximate approach was feasible in practice.

We do not include the full Java Class Library (JCL) in our analysis for performance reasons. This exclusion is only a source of imprecision in our analysis. For certain methods (e.g., members of the collections framework) we provide highly precise summaries. For unsummarized methods, LEGATO conservatively propagates information from arguments to return values/receiver objects similar to TaintDroid [16].

5.1 Alias Queries

To resolve the *mayAlias* queries on heap writes (see Section 3.5), we use a demand-driven, context and flow-sensitive alias resolution [41]. A single alias query must complete within a user-configurable time limit; if this budget is exceeded, LEGATO reports the configuration value as lost into the heap similar to the approach taken by Torlak and Chandra [44]. This was not a source of any false positives in our evaluation. We take a similar approach on flows of resources into static fields. Static fields are global references that persist throughout the entire lifetime of the program. We conservatively flag any write of a resource derived value that flows into a static field. This dramatically improved our alias resolution time and did not lead to many false positives.

5.2 Resource Model

The analysis described in Sections 3 and 4 is stated in terms of only one external resource. Our implementation handles multiple resources by operating over maps from resource names to individual ARVs. For generality, our implementation is parameterized over the resource access model of an application. A model defines the resource access sites in an application, and for each site returns the set of resource names potentially accessed at that site. The soundness and precision of LEGATO depends on the choice resource model: a model that

omits some access sites may cause LEGATO to miss potential bugs. Similarly, an overly coarse model will be sound but likely imprecise in practice. However, in our evaluation we found that resource access sites are easy to identify in practice; we describe the resource models used in for our evaluation in Section 6.

The resource model used with LEGATO is unconstrained in the choice of resource names. This flexibility enables the use of an imprecise model when resources may alias, or when the exact name of resources cannot be determined precisely at analysis time. Under an imprecise resource model, all access sites that may access the same concrete external resource are mapped to a common abstract resource name. For example, all accesses to files with the extension `.txt` may be mapped to the logical resource name `*.txt`. A similar approach may be used when two or more resources interact or share state, i.e., resources with distinct names that share state may be given the same abstract resource name.

5.3 Context-Sensitivity

Each call site of a method m may call the method with different abstract input values. However, the IDE framework computes the values within m by taking the meet over all abstract inputs. This leads to imprecision in the following scenario:

```

1 do_print(a) {
2   print(a);
3 }
4 do_print(get1());
5 do_print(get2());

```

The standard value computation within `do_print` would assign `a` the value $\hat{1} \sqcap \hat{2} = \perp$ which is imprecise. Initial versions of LEGATO used the context-insensitive value computation provided in Heros [7], but our results were impractically imprecise.

To overcome this imprecision, the LEGATO implementation extends the value computation phase of Heros to make it context-sensitive. We require an initial context and a context extension operator. At a call site to method m , the context of the call site C is extended with the extension operator, yielding the context C' for values computed within m originating from context C . The original value computation pass of the IDE framework is then executed for the method body with respect to the new context.

In our instantiation, we use an adaptive, k -limited call-string context scheme similar to that in [35]. To trade-off precision and scalability, we initially run the value analysis pass with all contexts limited to length 1. If LEGATO derives the value \perp for some method parameter p in context C , it consults the corresponding argument values in all incoming contexts. If the argument in each incoming contexts is non- \perp , LEGATO infers that the \perp value computed for p was due to insufficient context-sensitivity. LEGATO then adaptively increases the context-sensitivity for all such call sites, and then re-runs the value computation phase. This process is repeated until no \perp parameter values arise due to insufficient context-sensitivity, although we impose an configurable artificial maximum length (6 in our experiments) to ensure termination. In our experiments, this limiting was the source of only 3 false positives.

The approach described above is necessarily more expensive than the original IDE framework, which runs only one value computation phase. In practice, the context-sensitive value computation phase does not significantly contribute to analysis time for two reasons. First, LEGATO needs only a handful of value computation phases to either rule out false positives from insufficient context-sensitivity or reach the configured limit. Second, within each value computation phase, values are computed within a method using context-*insensitive* summary functions, which are generated in an initial pass of the IDE analysis. These summary

■ **Table 1** Measures of application complexity in the evaluation suite. # IR Statements is the count across all methods of statements in the intermediate representation used by Soot.

Program	Classes	Methods	Call Graph Edges	# IR Statements	Options
snipsnap	643	3,318	20,079	68,841	19
vqwiki	506	5,019	43,211	145,891	73
jforum	528	3,075	15,607	41,319	48
subsonic	886	4,578	20,768	67,615	44
mvnforum	938	10,548	132,712	409,847	90
personalblog	371	1,427	8,186	25,514	16
ginp	205	1,011	8,100	26,448	7
pebble	576	2,989	20,646	66,477	7
roller	853	4,735	30,229	95,439	29
blojsom	471	1,782	15,846	26,786	67

functions are symbolic abstractions of the method behavior on all possible input values. As a result, there is no need to re-analyze a method under each new context, which keeps recomputing values under new contexts relatively inexpensive.

5.4 Limitations

A fundamental limitation of our analysis is that we do not consider any possible synchronization between resource updates and resource accesses or between multiple resource accesses. This limitation will only yield false positives, as this means our analysis may be overly conservative in considering a program’s resource accesses. Our prototype could, with modest effort, include annotations to indicate an access always returns the same abstract version or multiple access sites return the same abstract version.

Our analysis soundness is stated only in terms of direct information flow, i.e., we ignore the effects of implicit flow. Thus, LEGATO will fail to detect when two or more accesses of the same resource indirectly flow to a program value. We experimented with a version of the analysis that considered implicit flow but, as is common [22], the ratio of false positives to true positives was overwhelming.

As mentioned above, LEGATO relies on the Soot analysis framework for call-graph construction, reflection resolution, type hierarchy construction, etc. Thus, LEGATO is sound modulo the soundness of the underlying Soot framework implementation.

6 Evaluation

To evaluate LEGATO, we focused on the issue of consistency in the presence of dynamic configuration updates. A dynamic configuration update (DCU) is a configuration change that occurs at run time that takes effect without program restart. We chose this problem as representative of the broader problem of consistent dynamic resource usage, as we are unaware of any existing static analysis that is capable of effectively addressing this problem. The only tool we are aware of in this area is our prior work on Staccato [43], which is a dynamic analysis that may yield false negatives. In addition, Staccato LEGATO is parameterized over the dynamic resource being analyzed.

We are interested in the following questions:

- Does LEGATO find dynamic resource consistency errors in the analyzed applications with a reasonable ratio of true to false positives?
- Are the time and memory requirements to run LEGATO reasonable?

Experimental Setup. We evaluated LEGATO on 10 Java server applications. A summary of the applications and metrics related to code base and call graph size (as measures of application complexity) can be found in Table 1. We selected these applications from three sources. Subsonic and JForum come from our prior work on Staccato we include them for comparison with prior results.⁶ Personalblog, Snipsnap, Roller, and Pebble are from the Stanford SecuriBench suite [28], a set of commonly analyzed web apps [29, 23].⁷ Finally, we also used applications from prior work by Tripp et al. on TAJ [46], a taint analysis for web applications. We used all projects from TAJ’s evaluation that satisfied the following conditions: a) the source code is publicly available, b) the project is a single, self-contained application, and c) the application supports dynamic configuration updates. The applications satisfying these conditions are VQWiki, MVNForum, Ginp, and Blojsom. Where possible, we used the same versions of the projects as those used in the original TAJ paper.

The dynamically configurable options of every application may be changed by an administrator at any point while processing a request. Across all our applications, applications accessed the configuration by reading from a global, in-memory map. When the configuration is changed by an administrator (either via the web interface or editing the on-disk configuration file) a thread in the application updates the in-memory configuration map. This thread runs concurrently with request handler threads that read from the configuration map.

Given this implementation pattern, we treated each individual option as a separate resource that can change at any moment. Every application accessed configuration options by either passing static strings to a key-value API (e.g., `Config.getValue("db-password")`) or calling option-specific getter methods (e.g., `Config.getDBPassword()`). We implemented generic resource models for these two access patterns. When analyzing an application, we specialized the appropriate model with an application-specific configuration YAML file which described the application’s configuration API. The longest such file was only 195 lines. The number of options tracked for each application are included in Table 1.

All of the applications in our evaluation were written to run in a Java Servlet container [32]. To soundly model these applications, we generated driver programs based on the servlet container specification and used sound stub implementations of the servlet API. For heavily used parts of the Java Class Library, such as the collection and database APIs, we used hand written summaries. For other methods without implementations, we used the over-approximation of method behavior discussed in Section 5.

We performed two experiments. To measure the effectiveness of LEGATO, we ran the analysis on each evaluation program, and recorded all at-most-once violations reported by the analysis. We then manually classified these reports as either a true bug or false positive. (Where possible, we reported any true bugs we found to the original developers.)

To measure the performance of LEGATO, we ran the analysis 5 times for each application while collecting timing and memory usage information. We break down the time of the analysis into three components: call-graph construction time, alias query resolution time,

⁶ Staccato was also applied to Openfire, but was used only to detect out-of-date configurations, an orthogonal issue to consistency.

⁷ The SecuriBench suite contains 9 applications, but the remaining 5 do not support DCU.

■ **Table 2** Bug reports from LEGATO. **TP** and **FP** are the numbers of true and false positives respectively. The last four columns record sources of false positives: **PS** is path-insensitivity, **SYN** is the conservative handling of synchronization, and **SF** is the conservative handling of static fields discussed in Section 5. **O** counts causes not included in the above categories, and includes imprecision due lack of application-, library-, or framework-knowledge. $t\backslash o$ indicates no reports due to timeout.

Project	TP	FP	PS	SYN	SF	O
jforum	4	14	2	1	3	8
ginp	7	1	0	0	1	0
vqwiki	12	8	2	4	1	1
snipsnap	2	2	1	0	1	0
pebble	0	4	3	0	0	1
subsonic	31	12	1	9	2	0
personalblog	1	3	3	0	0	0
roller	6	5	1	0	1	3
mvnforum	2	27	19	0	0	8
blojsom	$t\backslash o$	$t\backslash o$	$t\backslash o$	$t\backslash o$	$t\backslash o$	$t\backslash o$

and core analysis time, and report the average of these times. To measure the memory requirements of LEGATO, we sampled the heap size of the JVM every second. We intentionally avoid garbage collection before sampling the heap size. We found that excessive garbage collection caused an artificially high number of alias query timeouts, which ultimately skewed the analysis results and reported memory requirements.

All experiments were run on AWS EC2 m4.xlarge instances with 4 virtual CPUs at 2.4GHz, using the OpenJDK VM version 1.7.0_131, with 10GB of memory allocated to the JVM. We limited all aliasing queries to ten seconds, and set a 15 minute timeout for each run of the analysis.

6.1 Analysis Effectiveness

The results of running LEGATO on programs in our evaluation suite are shown in Table 2. LEGATO successfully completed within the 15 minute budget on 9 of the 10 applications in our evaluation suite (we discuss the reason for Blojsom’s timeout below). Of the 9 applications on which LEGATO completed, the analysis found bugs in 8. Although the false positive *ratio* is relatively high, we were able to classify the results with minimal effort as many of the false positives were obvious. In many cases (84.6% of column **PS**) LEGATO detected that it lost precision due to control-flow join and automatically flagged the result as a potential false positive. We also exploited that ARVs are traces of flows from access to report sites to help interpret errors reported by our tool. We were able to find these bugs with a simple resource model (Section 5.2) and without being experts in the programs.

There are potentially two sources of false positives: imprecision in the analysis and the at-most-once condition being too strong for application specific reasons. In practice, we found that all false positives were the result of imprecision in the analysis. The primary source of imprecision was the lack of general path-sensitivity in the analysis (column **PS**). For example, almost all of the path-sensitivity false positives in MVNForum (16) were the result of identical code being cloned across different branches of conditional statements. The second largest source of false positives was the conservative handling of code that required application-, library-, or framework-specific domain knowledge to precisely model (included in column **O**). For example, 8 false positives in the **O** column of JForum are due to imprecise

```

1 // Instance (1)
2 request.setAttribute("url", config.getUrl());
3 request.setAttribute("baseurl", config.getUrl());
4 // Instance (2)
5 String url = "/space/" + encode(snip.getName());
6 url += "/" + encode(att.getName());
7 // ...
8 String encode(String toEncode) {
9     String encodedSpace = config.getEncodedSpace();
10    return toEncode.replace(" ", encodedSpace);
11 }

```

■ **Figure 12** Two simplified examples of the “double read” pattern found in Snipsnap.

```

1 List<String> getPodcastUrls() {
2     List<String> toReturn = new List<>();
3     for(...) {
4         String baseUrl = // ...
5         int port = config.getStreamPort();
6         toReturn.add(rewriteWithPort(baseUrl, port));
7     }
8     return toReturn;
9 }

```

■ **Figure 13** A correlated access found in Subsonic, where the "streamPort" option is aggregated into the toReturn variable.

models of Java’s reflection API. Our control-flow graph contained a control-flow edge from the return-site of a `Method.invoke` reflective invocation to a `MethodNotFoundException` exception handler, when the represented control-flow path is actually unrealizable.

6.1.1 Sample Bugs

We now highlight some of the bugs found and discuss broad patterns we noticed in our results. Many bugs arose from three patterns: 1) two sequential accesses to the same configuration option, 2) using a configuration option in a loop, and 3) storing configuration derived data in a global cache that was not cleared on update.

Double Reads. We found 4 instances of applications immediately combining two successive reads of the same option. Two simplified instances we found in the Snipsnap program are shown in Figure 12. In the first instance, `config.getUrl()` returns a URL based on the dynamically configurable option specifying the location of the web application. If this option changes between the two accesses, the `request` object’s attributes will contain URLs pointing to two different locations. This could cause confusion for the user as only a subset of links on the page returned by Snipsnap would be valid.

The second instance is similar as the two invocations of `encode` both access the dynamically configured `encodedSpace` option. In this instance, the URL returned to the user will contain a mix of incorrectly and correctly encoded spaces. As with the first instance, this bug can cause links in the returned page to mysteriously fail to work.

The author of Snipsnap confirmed that these two instances corresponded to true bugs, but declined to fix them due to age of the project, lack of active deployments, and the author’s judgment that the bugs were not serious enough to warrant a fix [20].

Correlated Accesses within Loops. Out of the 65 true reports, 21 were instances of correlated accesses of configuration options within a loop. We counted instances where a value derived from a configuration option read within a loop is aggregated with configuration-derived values from previous iterations of the same loop. The aggregated value is therefore derived from multiple accesses of the same option, violating our at-most-once condition. The priming approach described in Section 3.3 was crucial to detect these bugs.

A simplified example of this pattern, found in Subsonic, is shown in Figure 13. The URLs computed by the method are used to generate an XML file served to podcast subscription clients. If some of the URLs generated by the method have inconsistent port numbers, the subscription client end-user would be presented with a handful of podcasts that fail to work. Further, unlike broken links on a webpage, the generated XML file is likely never seen by the end-user and thus it may not be obvious that a refresh may solve the problem.

We also found this pattern in other applications in our benchmark suite. For example, in MVNForum, a web forum application, the email module may send messages to multiple recipients, but constructs each message in different iterations of a loop. During each loop iteration, MVNForum reads configuration options that specify the message's sender name and address, which may yield a batch of messages with inconsistent sender information.

Finally, we found an example in VQWiki, a wiki web application, that potentially led to a corrupted search index. While constructing the index, VQWiki executes a loop to generate the set of documents to add to the index. Each loop iteration reads a configuration option that controls the location of the application's data files; this value is then stored in the indexed document. If the value of the option were to change between loop iterations, the index would be corrupted and only recover on the next complete index rebuild.

Caching in Static Fields. As explained in Section 5.1, to avoid expensive alias queries for static fields while retaining soundness, we issue a report for each static field to which resource-derived information flows. This rough heuristic identified 4 instances where the at-most-once condition was violated due to caching.

For example, JForum (another forum application) can replace tokens in user text with embedded images of emojis. The URLs for these emojis, as with all URLs generated by JForum, are computed based on the dynamically configurable location of the forum application. JForum lazily computes the URL for every available emoji, then caches the results in a static field. However, if the administrator changes the base location of the application, this cache is not cleared. As a result, all links and images post-update will use the new location except for the emojis, which will be broken. Refreshing the page will not fix this issue as it requires the administrator to manually clear the emoji URL cache or restart the application.

In another more serious example, we found an instance in Roller where the login component cached whether password encryption was enabled in a static field populated at startup. However, user administration actions (e.g., update user, create user, etc.) always read the most up-to-date version of this flag, and encrypt passwords as appropriate. Thus, after changing this flag, any new users created by the administrator would be unable to log in until the entire application was restarted.

Other Patterns. We found multiple cases where configuration derived values were stored into the heap in one method, and then later combined with another configuration derived value in another method. A minimized example of this pattern, found in Ginp, is shown in Figure 14. Like most of the web applications in our evaluation, Ginp uses Java Servlet

```

1 // in doStartTag
2 this.cols = horiz / Config.getThumbSize();
3 this.rows = vert / Config.getThumbSize();
4 // in doAfterBody
5 if(count - start >= this.rows * this.cols)
6   showPicture = false
7 // in _jspService (autogenerated)
8 int _j_0 = _jspx_getpictures.doStartTag();
9 // 41 lines of auto-generated code
10 int _e = _jspx_getpictures.doAfterBody();

```

■ **Figure 14** Inconsistency bug found in Ginp. Detecting this bug requires precisely modeling framework code, and handling flows through method calls and the heap.

Pages (JSP), a dialect of HTML which allows mixing arbitrary Java code and user defined tags (such as `<ginp:getpictures.../>`). At page rendering time, JSP pages are transpiled into Java code and compiled. User defined tags are transformed into a sequence of calls to programmer defined callbacks. However, programmers generally only interact with the JSP source code and do not see the intermediate code containing the callback invocations.

The bug found by LEGATO involved one such user-defined tag. In one callback (`doStartTag`, lines 2 and 3), the same configuration option is read twice and stored into two seemingly unrelated heap locations. However, in a second callback (`doAfterBody`, lines 5 and 6) these two values are incorrectly combined to decide a loop condition. Finding this bug required precisely tracing the two abstract resource versions interprocedurally through the heap.

In another example, found in JForum, an SMTP mail session is constructed using the value of the dynamically configured mail host and then stored into an object field. In another method, this session is used to construct a transport, again using the value of the mail host option. If the mail host option changes between these two calls, the transport may try to connect to a mail host different from that of the mail session, which could cause the mail sending process to fail. Further, we confirmed that if the mail sending process failed with an exception, the messages to be sent were dropped and never resent.

Finally, we found a bug in Subsonic that relied on the application level concurrency approach described in Section 4.4. In this instance, a web request would initiate an update of an in-memory list of remote clients. However, this list was protected by a synchronized block. LEGATO concluded that a configuration-derived value placed in the list could be mixed with other configuration-derived values that originated from other threads.

Comparison with Staccato. To validate the effectiveness of our analysis, we compared the bugs found by LEGATO with those found by Staccato. A direct comparison is impossible, as Staccato uses slightly different correctness conditions, unsound heuristics not present in LEGATO, and also detects different types of errors orthogonal to the at-most-once condition. However, the 4 bugs found by Staccato in JForum and Subsonic that correspond to our at-most-once condition were detected by LEGATO. This finding partially validates that the bugs found by LEGATO correspond to true DCU bugs.

6.2 Performance

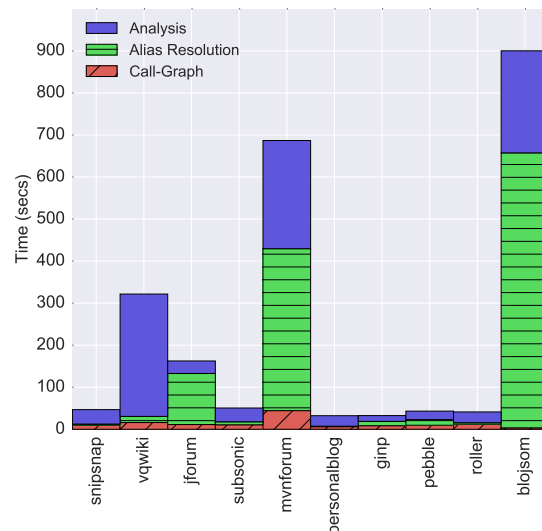
The results of our performance experiments are shown in Figure 16. Of the 10 applications, 9 finished within the 15 minute time limit, and 6 took less than a minute. For all applications in our evaluation suite, the 10GB heap limit was sufficient: the smallest peak heap size we

```

1 Request req = ...;
2 Response resp = ...;
3 HashMap context = new HashMap();
4 for(Plugin p : plugins) {
5   p.process(req, resp, context);
6 }
7 sendResponse(resp);

```

■ **Figure 15** Sketch of code pattern that caused LEGATO to time out while analyzing Blojsom.



■ **Figure 16** Analysis times for the evaluation targets.

observed was 0.5GB while analyzing Ginp and the largest was 7.5GB on MVNForum.

We now discuss the cause of Blojsom’s timeout. The vast majority of Blojsom’s 15 minute analysis budget was spent resolving alias queries. We found these expensive alias queries were caused by a problematic code pattern, which we sketch in Figure 15. Blojsom delegates the majority of request processing and application logic to 79 different plugins which are called via interface methods in a for loop during request processing (lines 4–6). To track per-request state, a shared `HashMap context` is also passed to each plugin; many plugins write configuration information into this map. To find all aliases of `context`, the alias resolver must explore all backwards paths of execution through the loop. Unfortunately, the megamorphic callsite on line 5 causes an explosion in the paths that must be explored, which quickly overwhelms the alias resolver. We could potentially address this issue by using a less precise approach to aliases, at the cost of overall analysis result quality.

7 Related Work

Typestate Analysis and Affine Type Systems. The phrase at-most-once often evokes linear (or more accurately, affine) type systems [49, 18, 45, 8, 13]. Both linear and affine type systems restrict how often a value may be used. Linear type systems guarantee that values may not be duplicated or destroyed, which enforces an exactly-once use discipline. Affine type systems allow destruction, which enforces an at-most-once use discipline. In contrast, under the at-most-once condition resources may be accessed multiple times, and may copied and re-used by the program. The at-most-once restriction only requires that each value depends only on at most one resource access.

Similar to linear and affine types, typestate analyses [42, 14, 50, 12, 17, 34] focus on verifying that the use of some object or resource follows a specific protocol. For example, the motivating example given in the original typestate paper by Strom et al. [42] is to verify that file handles are not written to after being closed. These access protocols are generally expressed in terms of an abstract state assigned to each object, and a set of methods or operations that cause transitions of object state according to some automaton.

The at-most-once condition is difficult to accurately capture using this framework. Although it would be possible to design an automaton to enforce that each resource was *used* exactly once during a value’s computation, this condition is stricter than LEGATO’s.

External Resources. There has been considerable effort into analyzing and understanding the external resources used by an application. For example, in the database community, recent work by Linares-Vásquez et al. [26] has looked at generating descriptions of how applications interact with databases. In a related work, Maule et al. among others [37, 30] have looked at evaluating the impact of database changes on applications. For configurable software understanding how software behaves under different configurations remains an active area of research [39, 24, 38]. Existing research on software configuration consistency has primarily focused on ensuring consistency between related configuration options. For example, Nadi et al. [33] examined constraints between compile-time configuration options for the Linux kernel. We consider this orthogonal to the consistency issues discussed here.

In addition to the above work on static external resources, verifying consistent behavior in the presence of *dynamic*, external resources has also been an active area of research. There has been considerable work in the security field to prevent vulnerabilities due to malicious, concurrent changes of the filesystem [36, 6, 31, 9].

Several decades of database research on transactions and isolation has focused on ensuring that applications interact consistently with the database. For example, serializeable isolation [5] can prevent check-then-act errors within a transaction by determining when a concurrent update has invalidated a previous read. Although this isolation can prevent consistency errors due to concurrent updates, empirical research performed by Bailis et al. [1] has shown that applications that eschew database level transactions (specifically Ruby on Rails applications) struggle to maintain consistency in the presence of concurrent writers.

To find errors in dynamic configuration update implementations (the instantiation considered in Section 5), our prior work on the Staccato dynamic analysis checks for correctness violations in applications with configuration changes at runtime [43]. One of the two correctness conditions checked by Staccato closely mirrors our at-most-once condition. However, Staccato does not consider multiple reads of the same option to be an error provided the same value is returned on each access. Thus, when considering multiple accesses on the same value, the at-most-once condition of LEGATO can be stricter than that checked by Staccato.

8 Conclusion

We presented LEGATO, a novel static analysis for detecting consistency violations in applications that use external resources. LEGATO verifies the at-most-once condition, which requires that all values depend on at most one access to each external resource. LEGATO efficiently checks this condition without explicitly modeling concurrency by using abstract resource versions. We demonstrated the effectiveness of this approach on 10 real-world Java applications that utilize dynamically changing configuration options.

References

- 1 Peter Bailis, Alan Fekete, Michael J Franklin, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica. Coordination avoidance in database systems. *Proceedings of the VLDB Endowment*, 8(3), 2014.

- 2 Paulo Barros, Suzanne Just, Renéand Millstein, Paul Vines, Werner Dietl, and Michael D Ernst. Static analysis of implicit control flow: Resolving java reflection and android intents. In *ASE*, 2015.
- 3 Tom Bergan, Dan Grossman, and Luis Ceze. Symbolic execution of multithreaded programs from arbitrary program contexts. In *OOPSLA*, 2014.
- 4 Arthur J Bernstein, Philip M Lewis, and Shiyong Lu. Semantic conditions for correctness at different isolation levels. In *Data Engineering*, 2000.
- 5 Philip A Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency control and recovery in database systems*. Addison-Wesley Pub. Co. Inc., Reading, MA, 1987.
- 6 Matt Bishop and Michael Dilger. Checking for race conditions in file accesses. *Computing systems*, 2(2), 1996.
- 7 Eric Bodden. Inter-procedural data-flow analysis with ifds/ide and soot. In *State of the Art in Java Program analysis*, 2012.
- 8 Chandrasekhar Boyapati, Barbara Liskov, and Liuba Shrira. Ownership types for object encapsulation. In *POPL*, 2003.
- 9 Xiang Cai, Rucha Lale, Xincheng Zhang, and Robert Johnson. Fixing races for good: Portable and reliable unix file-system race detection. In *Information, Computer and Communications Security*, 2015.
- 10 Sagar Chaki, Edmund Clarke, Alex Groce, Joël Ouaknine, Ofer Strichman, and Karen Yorav. Efficient verification of sequential and concurrent c programs. *Formal Methods in System Design*, 25(2-3):129–166, 2004.
- 11 Sagar Chaki, Edmund Clarke, Nicholas Kidd, Thomas Reps, and Tayssir Touili. Verifying concurrent message-passing c programs with recursive calls. In *TACAS*, 2006.
- 12 Manuvir Das, Sorin Lerner, and Mark Seigle. Esp: Path-sensitive program verification in polynomial time. In *PLDI*, 2002.
- 13 Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *PLDI*, 2001.
- 14 Robert DeLine and Manuel Fähndrich. Typestates for objects. In *ECOOP*, 2004.
- 15 Alain Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *PLDI*, 1994.
- 16 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. *TOCS*, 32(2), 2014.
- 17 Stephen J Fink, Eran Yahav, Nurit Dor, G Ramalingam, and Emmanuel Geay. Effective typestate verification in the presence of aliasing. *TOSEM*, 17(2), 2008.
- 18 Jean-Yves Girard. Linear logic. *Theoretical computer science*, 50(1), 1987.
- 19 Neil D. Jones and Steven S. Muchnick. Flow analysis and optimization of lisp-like structures. In *POPL*, 1979.
- 20 Matthias L. Jugel. Personal Communication, 2017.
- 21 Uday P Khedker, Amitabha Sanyal, and Amey Karkare. Heap reference analysis using access graphs. *TOPLAS*, 30(1), 2007.
- 22 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*, 2008.
- 23 Johannes Lerch, Johannes Späth, Eric Bodden, and Mira Mezini. Access-path abstraction: Scaling field-sensitive data-flow analysis with unbounded access paths. In *ASE*, 2015.
- 24 Max Lillack, Christian Kästner, and Eric Bodden. Tracking load-time configuration options. In *ASE*, 2014.
- 25 Yu Lin. *Automated refactoring for Java concurrency*. PhD thesis, University of Illinois at Urbana-Champaign, 2015.

- 26 Mario Linares-Vásquez, Boyang Li, Christopher Vendome, and Denys Poshyvanyk. Documenting database usages and schema constraints in database-centric applications. In *ISSTA*, 2016.
- 27 Peng Liu, Omer Tripp, and Xiangyu Zhang. Flint: fixing linearizability violations. In *OOPSLA*, 2014.
- 28 Ben Livshits. Stanford securibench suite. <http://suif.stanford.edu/~livshits/securibench/>, 2017.
- 29 Benjamin Livshits. *Improving software security with precise static and runtime analysis*. PhD thesis, Stanford University, 2006.
- 30 Andy Maule, Wolfgang Emmerich, and David S Rosenblum. Impact analysis of database schema changes. In *ICSE*, 2008.
- 31 William S. McPhee. Operating system integrity in os/vs2. *IBM Systems Journal*, 13(3), 1974.
- 32 Rajiv Mordani and Shing Wai Chan. Java servlet specification, 2009.
- 33 Sarah Nadi, Thorsten Berger, Christian Kästner, and Krzysztof Czarnecki. Mining configuration constraints: Static analyses and empirical results. In *ICSE*, 2014.
- 34 Nomair A. Naeem and Ondrej Lhotak. Tpestate-like analysis of multiple interacting objects. In *OOPSLA*, 2008.
- 35 Mayur Hiru Naik. *Effective Static Race Detection For Java*. PhD thesis, Stanford, 2008.
- 36 Mathias Payer and Thomas R. Gross. Protecting applications against toctou races by user-space caching of file metadata. In *VEE*, 2012.
- 37 Dong Qiu, Bixin Li, and Zhendong Su. An empirical analysis of the co-evolution of schema and code in database applications. In *FSE*, 2013.
- 38 Ariel Rabkin and Randy Katz. Static extraction of program configuration options. In *ICSE*, 2011.
- 39 Elnatan Reisner, Charles Song, Kin-Keung Ma, Jeffrey S Foster, and Adam Porter. Using symbolic evaluation to understand behavior in configurable software systems. In *ICSE*, 2010.
- 40 Mooly Sagiv, Thomas Reps, and Susan Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Theor. Comput. Sci.*, 167(1-2), 1996.
- 41 Johannes Späth, Lisa Nguyen Quang Do, Karim Ali, and Eric Bodden. Boomerang: Demand-driven flow-and context-sensitive pointer analysis for java. In Shriram Krishnamurthi and Benjamin S. Lerner, editors, *30th European Conference on Object-Oriented Programming (ECOOP 2016)*, volume 56 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 22:1–22:26, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. URL: <http://drops.dagstuhl.de/opus/volltexte/2016/6116>, doi: 10.4230/LIPIcs.ECOOP.2016.22.
- 42 Robert E Strom and Shaula Yemini. Tpestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, 12, 1986.
- 43 John Toman and Dan Grossman. Staccato: A Bug Finder for Dynamic Configuration Updates. In *ECOOP*, 2016.
- 44 Emina Torlak and Satish Chandra. Effective interprocedural resource leak detection. In *ICSE*, 2010.
- 45 Jesse A. Tov and Riccardo Pucella. Practical affine types. In *POPL*, 2011.
- 46 Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. Taj: Effective taint analysis of web applications. In *PLDI*, 2009.
- 47 Raja Vallée-Rai, Etienne Gagnon, Laurie Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. Optimizing java bytecode using the soot framework: Is it feasible? In *Compiler Construction*, 2000.
- 48 David Van Horn and Matthew Might. Abstracting abstract machines. In *ICFP*, 2010.

- 49 Philip Wadler. Linear types can change the world. In *IFIP TC*, 1990.
- 50 Daniel M Yellin and Robert E Strom. Protocol specifications and component adaptors. *TOPLAS*, 19(2), 1997.

A Appendix: Soundness

A.1 Preliminaries

Although the environment transformers presented in the main paper gave semantics as denotations from statements to environment transformers, the IDE framework of Sagiv et al. assigns transformers to edges in the program control flow graph. Following the notation of Sagiv et al. in [40], assume we have a function $M : E^* \rightarrow (Env \rightarrow Env)$, which maps an edge in the program control-flow graph to an environment transformer. This function naturally extends to paths of edges by composing the environment transformers for each successive edge in a path.

The solution computed by the IDE framework is the meet-over-all-paths solution,⁸ defined for a distinguished start node s_0 and start environment Ω as:

$$MOP(n) \triangleq \bigsqcap_{p \in path(s_0, n)} M(p)(\Omega)$$

In other words, the meet-over-all-paths the meet of applying the transformers for every path from s_0 to n to the start environment Ω .

Our proofs exploit this path-based paradigm: we give the abstract and concrete instrumented semantics as assignments of transformers to edges. It is easy to see the correspondence to the transformers presented in the paper.

A.2 Concrete Instrumented Semantics

We first define the domain of concrete instrumented states as: $S = (X \rightarrow \mathbb{P}(\mathbb{N})) \times \mathbb{N}$, where X is the finite domain of variables that appear in a given program. We denote a concrete instrumented state of type S with $\langle env, c \rangle$. The instrumented semantics are given by the following assignment of transformers of type $S \rightarrow S$ to edges in the program supergraph:

$$\text{if } e \text{ then } s_1 \text{ else } s_2 \rightarrow s_1 \triangleq id \tag{1}$$

$$\text{if } e \text{ then } s_1 \text{ else } s_2 \rightarrow s_2 \triangleq id \tag{2}$$

$$\text{while } e \text{ do } s \text{ end } \rightarrow s \triangleq id \tag{3}$$

$$\text{while } e \text{ do } s \text{ end } \rightarrow s' \triangleq id \tag{4}$$

$$\text{skip} \rightarrow s \triangleq id \tag{5}$$

$$x = y \rightarrow s \triangleq \lambda \langle env, c \rangle. \langle env[x \mapsto env[y]], c \rangle \tag{6}$$

$$x = c \rightarrow s \triangleq \lambda \langle env, c \rangle. \langle env[x \mapsto \emptyset], c \rangle \tag{7}$$

$$x = y + z \rightarrow s \triangleq \lambda \langle env, c \rangle. \langle env[x \mapsto env[y] \cup env[z]], c \rangle \tag{8}$$

$$x = \text{get}^\ell() \rightarrow s \triangleq \lambda \langle env, c \rangle. \langle env[x \mapsto \{c\}], c + 1 \rangle \tag{9}$$

⁸ Technically, when considering interprocedural programs, the IDE framework computes the meet-over-all-valid-paths solution. As we do not consider methods in this section, we instead state our proofs using the simpler notion of meet-over-all-paths.

Where the edge in Equation (1) refers to the edge from the conditional header to the node corresponding to the branch statement s_1 , and similarly for Equation (2) and the false branch s_2 . The edge in Equation (3) corresponds to when the loop condition is true, and the loop body executed, whereas the edge in Equation (4) is when the loop condition is false and the loop is skipped. All other edges refer to the (unique) edge from a statement to its successor in the supergraph.

Define $C(p)$ as the composition of the transformers corresponding to each edge in the path p . Let Ω be the initial instrumented state, defined to be: $\langle \lambda_.\emptyset, 0 \rangle$.

A.3 Abstract Semantics

Let the domain of primed labels presented in the paper be denoted by $L = \widehat{\ell}^n \cup \{\top, \perp\}$. The environments used in the paper are of type $\widehat{S} = X \rightarrow L$. We will denote environments of type \widehat{S} with \widehat{env} . The distributive environment transformers in the paper are equivalent to the transformers of type $\widehat{S} \rightarrow \widehat{S}$ assigned to the edges in the supergraph:

$$\text{if } e \text{ then } s_1 \text{ else } s_2 \rightarrow s_1 \triangleq id \quad (10)$$

$$\text{if } e \text{ then } s_1 \text{ else } s_2 \rightarrow s_2 \triangleq id \quad (11)$$

$$\text{while } e \text{ do } s \text{ end} \rightarrow s \triangleq id \quad (12)$$

$$\text{while } e \text{ do } s \text{ end} \rightarrow s' \triangleq id \quad (13)$$

$$\text{skip} \rightarrow s \triangleq id \quad (14)$$

$$x = y \rightarrow s \triangleq \lambda \widehat{env}. \widehat{env}[x \mapsto \widehat{env}[y]] \quad (15)$$

$$x = c \rightarrow s \triangleq \lambda \widehat{env}. \widehat{env}[x \mapsto \top] \quad (16)$$

$$x = y + z \rightarrow s \triangleq \lambda \widehat{env}. \widehat{env}[x \mapsto \widehat{env}[y] \sqcap \widehat{env}[z]] \quad (17)$$

$$x = \text{get}^\ell() \rightarrow s \triangleq \lambda env. \lambda v. \begin{cases} \widehat{\ell} & \text{if } v \equiv v' \\ \widehat{\ell}^{n+1} & \text{if } env(v') \equiv \widehat{\ell}^n \\ env(v') & \text{o.w.} \end{cases} \quad (18)$$

Where the edges have the same interpretation as those given for the concrete semantics. At first glance, the use of id for loops and conditionals may appear incorrect. However, because the IDE framework computes the meet over all paths solution, the final result of the analysis takes the meet of all paths through a conditional, giving us the same effect. A similar observation applies for computing loop fixpoints.

Let $A(p)$ be the composition of the environment transformers corresponding to each edge in the path p , and let the initial abstract state Ω be defined to be $\top_{\widehat{S}}$, i.e., an environment that maps all variables to \top .

A.4 Proof

Define the invariant relation for two states as follows, $\langle env, c \rangle \sim \widehat{env}$ iff the following conditions hold:

$$\forall x. |env[x]| > 1 \Rightarrow \widehat{env}[x] = \perp \quad (\text{Invariant 1})$$

$$\forall x, y, m, n. m \neq n \wedge env[x] = \{m\} \wedge env[y] = \{n\} \Rightarrow \widehat{env}[x] \neq \widehat{env}[y] \vee \widehat{env}[x] = \perp \vee \widehat{env}[y] = \perp \quad (\text{Invariant 2})$$

$$\forall x. env[x] \neq \emptyset \Leftrightarrow \widehat{env}[x] \neq \top \quad (\text{Invariant 3})$$

We now show that:

► **Theorem 1.**

$$\begin{aligned} \forall n, n', p, p'. p' \equiv p \circ n \circ n' \wedge p' \in \text{path}(s, n') \wedge C(p \circ n)(\Omega) \sim A(p \circ n)(\Omega) \\ \Rightarrow C(p')(\Omega) \sim A(p')(\Omega) \end{aligned}$$

Theorem 1 states that if the invariant holds for the two environments yielded by the transformers along the path $p \circ n$, the invariant still holds after applying the respective transformers for the edge $n \rightarrow n'$.

Proof. Let $C(p \circ n)(\Omega) = \langle env, c \rangle$ and $A(p \circ n)(\Omega) = \widehat{env}$, and let $C(p')(\Omega) = \langle env', c' \rangle$ and $A(p')(\Omega) = \widehat{env}'$. We assume $\langle env, c \rangle \sim \widehat{env}$ and must show that $\langle env', c' \rangle \sim \widehat{env}'$. We proceed on the type of edge $n \rightarrow n'$ that makes up the final component of the path p' .

Cases (1), (2), (3), (4), (5), (6), (7): Trivial

Case (8): It suffices to show that after executing the environment transformer all invariants hold for the variable x on the left-hand side of the assignment.

Invariant 1 If $|env[y]| > 1$ or $|env[z]| > 1$ then by definition of \sim , $\widehat{env}[y] = \perp$ or $\widehat{env}[z] = \perp$, and by the definition of meet, $\widehat{env}'[x] = \widehat{env}[y] \sqcap \widehat{env}[z] = \perp$, preserving the invariant. Consider the case now where $|env[y]| = 1 \wedge |env[z]| = 1 \wedge env[y] \neq env[z]$. Then by the definition of \sim , $\widehat{env}[y] \neq \widehat{env}[z]$ or one or both of $\widehat{env}[y]$ and $\widehat{env}[z]$ is \perp . In either case, $\widehat{env}'[x] = \widehat{env}[y] \sqcap \widehat{env}[z] = \perp$, again preserving the invariant.

Invariant 2 If $env'[x] = \{m\} = env[y] \cup env[z]$, then either:

1. $env[y] = \{m\}$ and $env[z] = \{m\}$. Then by invariant 3, we have that $\widehat{env}[y] \neq \top$ and $\widehat{env}[z] \neq \top$. If either $\widehat{env}[y]$ or $\widehat{env}[z]$ is \perp , then $\widehat{env}'[x] = \perp$ and the condition is trivially satisfied. Similarly, if $\widehat{env}[y]$ and $\widehat{env}[z]$ are distinct, non- \perp values, then $\widehat{env}'[x]$ will be \perp and again the invariant is trivially satisfied. Finally, consider the case where $\widehat{env}[y] = \widehat{env}[z]$. Then $\widehat{env}'[x] = \widehat{env}[y] = \widehat{env}[z]$, and thus the invariant must hold by transitivity of equality and the invariant relation on the input environments.
2. $env[y] = \{m\}$ and $env[z] = \emptyset$. Then invariant 3 implies that $\widehat{env}[y] \neq \top$ and $\widehat{env}[z] = \top$, whence $\widehat{env}'[x] = \widehat{env}[y]$. If $\widehat{env}[y] = \perp$ then the invariant is trivially satisfied, otherwise the invariant holds from the transitivity of equality and the invariant on the input environments.
3. $env[y] = \emptyset$ and $env[z] = \{m\}$ follows from symmetric reasoning to the above.

Invariant 3 If $env'[x] \neq \emptyset$, then $env[y] \neq \emptyset \vee env[z] \neq \emptyset$. By invariant 3 on the input environments, this implies that $\widehat{env}[y] \neq \top \vee \widehat{env}[z] \neq \top$. By the definition of meet, we must have $\widehat{env}'[x] \neq \top$ as required.

To establish the other direction of the bi-implication, it suffices to show that $env'[x] = \emptyset \Rightarrow \widehat{env}'[x] = \top$. If $env'[x] = \emptyset$, then $env[y] = \emptyset \wedge env[z] = \emptyset$, whence by the invariant on the input environments, we have $\widehat{env}[y] = \top \wedge \widehat{env}[z] = \top$. As $\top \sqcap \top = \top$, we have the desired result.

Case (9): We again establish the invariants post assignment.

Invariants 1 and 3 Trivial.

Invariant 2 By simple proof by contradiction, it can be shown that c is greater than any version number that appears in env . Thus, as $env'[x] = \{c\}$ is distinct from all other singleton version sets, it suffices to show that $\widehat{env}'[x]$ is likewise distinct from all other abstract versions. As the environment transformer in (18) adds a prime to existing

values of the form $\widehat{\ell}^n$, this ensures that $\widehat{env}'[x] = \widehat{\ell}$ is unique within \widehat{env}' . Finally, for $y \neq x$, the priming process preserves inequality between abstract resource versions, ensuring the invariant holds. ◀

► **Corollary 2.** $\forall n, p. p \in \text{path}(s, n) \Rightarrow C(p)(\Omega) \sim A(p)(\Omega)$

Proof. By straightforward induction on path length and application of Theorem 1. ◀

We can now state the main soundness result:

► **Theorem 3.** $\forall p, n, x. p \in \text{path}(s, n) \wedge |C(p)(\Omega)[x]| > 1 \Rightarrow [\prod_{q \in \text{path}(s, n)} A(q)(\Omega)][x] = \perp$

In other words, Theorem 3 states that if any execution, at some point in the program a variable is derived from multiple versions of the resource, the analysis derives \perp for that variable at that point.

Proof. Observe that if, for some x , $|C(p)(\Omega)[x]| > 1$, then $A(p)(\Omega)[x] = \perp$ by Theorem 2, and by the definition of meet, $\prod_{q \in \text{path}(s, n)} A(q)(\Omega)[x] = \perp$ ◀

Definite Reference Mutability

Ana Milanova¹

Dept. of Computer Science, Rensselaer Polytechnic Institute, 110 8th Street, Troy NY, USA
milanova@cs.rpi.edu

Abstract

Reference immutability type systems such as Javari and ReIm ensure that a given reference cannot be used to mutate the referenced object. These systems are conservative in the sense that a mutable reference may be mutable due to approximation.

In this paper, we present ReM (for definite Re[ference] M[utability]). It separates mutable references into (1) definitely mutable, and (2) maybe mutable, i.e., references whose mutability is due to inherent approximation. In addition, we propose a CFL-reachability system for reference immutability, and prove that it is equivalent to ReIm/ReM, thus building a novel framework for reasoning about correctness of reference immutability type systems. We have implemented ReM and applied it on a large benchmark suite. Our results show that approximately 86.5% of all mutable references are definitely mutable.

2012 ACM Subject Classification Theory of computation → Program analysis, Software and its engineering → General programming languages

Keywords and phrases reference immutability, type inference, CFL-reachability, precision

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2018.25

Supplement Material ECOOP Artifact Evaluation approved artifact available at <http://dx.doi.org/10.4230/DARTS.4.3.7>

Acknowledgements We thank the ECOOP 2018 PC and AEC for valuable suggestions, and the National Science Foundation for supporting our work through NSF grant 1319384.

1 Introduction

Reference immutability ensures that a *readonly* reference cannot be used to modify the state of the object, including its transitively reachable state. For example, in the code below

```
1 Date md = new Date();
2 readonly Date rd = md;
3 rd.setTime(1);
4 md.setTime(1);
```

the `Date` object cannot be modified through the readonly reference `rd`, however, the same object can be modified through the mutable reference `md`.

Reference immutability has a wide variety of applications. It can enrich method specifications. It can help prevent errors due to unwanted aliasing and unwanted object mutation, as well as errors due to concurrency. It can enable compiler and runtime optimizations as well as reasoning about more complex properties such as method purity and object immutability. One application that has not received attention (to the best of our knowledge), is the impact

¹ This work was partially supported by NSF grant 1319384.



25:2 Definite Reference Mutability

of reference immutability on “flow systems”. Flow systems track and prevent flow from *positive* references to *negative* ones:

```
1 a = b;  
2 positive X x = ... ;  
3 a.f = x;  
4 negative X y = b.f;
```

Many interesting analyses fall into this category, most notably approximate computing systems (e.g., EnerJ [28]), which prevent flow of approximate values into precise ones, and taint systems, which prevent flow from sensitive sources to untrusted sinks (e.g., [29, 17]). Unfortunately, the natural subtyping `negative <: positive` is unsound in the presence of mutable references [3]. (In the above example, had we allowed for such subtyping, reference `a` could have been positive, `b` could have been negative, and the program would have type checked.) Therefore, flow systems disallow subtyping for reference types [29, 28, 12], forcing equality constraints at reference type assignments instead of the more precise subtyping constraints. Reference immutability can alleviate the imprecision arising from equality constraints – if the left-hand-side of the assignment is readonly, then subtyping is safe – allowing for more correct programs to type check. In summary, because of its many applications, reference immutability has been studied extensively [34, 39, 2, 40, 18, 13, 22], and it remains important to continue research in the area.

Javari [34] is the state-of-the art in reference immutability. ReIm [18] has similar core semantics but is less expressive and therefore simpler. In this paper we focus on ReIm because of its simplicity and clarity; we believe that our treatment extends to other reference immutability systems. Standard reference immutability systems, like Javari and ReIm capture what we call *definite immutability* – a `readonly` reference is truly immutable. However, a `mutable` reference may be truly mutable, or it may be mutable because of inherent approximation. ReIm (and Javari) approximate in the handling of structure-transmitted dependences [25] (i.e., flow through heap objects). For example, in the code below

```
x.f = y; ... w = z.f; w.g = ...
```

reference `y` is `mutable`. However, it is not necessarily mutable: if `x` and `z` refer to the same object `o`, then `y` is indeed mutable; if they refer to different objects, then it is not mutable (at least not because of the update to `w`). The system does not reason about aliasing, and errs on the safe side marking `y` `mutable`. ReIm (and Javari) handle call-transmitted dependences [25] precisely. In the code below, `id` is the standard “identity” function that returns its argument.

```
x = id(y); // x is readonly  
z = id(w); // z is mutable
```

Reference `y` is `readonly`, and `w` is `mutable`. The system properly transmits mutability without mixing the two call sites.

The key contribution of our paper is reasoning about approximation. We propose a new type system ReM (for definite Re[ference] M[utability]). ReM captures definite immutability, and in addition it captures definite mutability – a `mutable` reference is now *definitely mutable*. We note that our use of “definitely mutable” is somewhat inaccurate. Of course, whether a given reference is ever mutated is undecidable for various reasons, e.g., it is undecidable whether a given statement is executed, or whether a given path is executed. We use it in the sense of *definitely mutable according to CFL-reachability*, which is a highly precise

model of data dependence [25] and analyses are unlikely to improve upon it. ReM captures approximation explicitly by introducing the `maybe` qualifier. In the earlier example

$$x.f = y; \quad \dots \quad w = z.f; w.g = \dots$$

`y` is now `maybe` mutable. A key result is that empirically, approximation has limited impact – only about 13% of all ReIm-mutable references (about 6% of all references) are `maybe` mutable, leading to a conclusion that ReIm and ReM are precise, and therefore can be used to power client analyses.

Another contribution of our paper is the interpretation of reference immutability in terms of Context Free Language reachability, commonly referred to as CFL-reachability [27, 26, 25]. We propose a CFL-reachability system for inference of reference immutability, and prove that it is equivalent to the ReIm/ReM inference system, thus building a framework for reasoning about correctness, and proving ReIm and ReM correct. To the best of our knowledge, ReIm has not been proven correct, even though it has been used to power client analyses [17, 32]. We plan to extend our system for reasoning about approximation and correctness to flow systems [29, 28, 18, 17]. A CFL-reachability interpretation is beneficial for several reasons: (1) it defines the semantics of reference immutability type systems in terms of intuitive and well-known concepts, which may lead to wider applicability of reference immutability type systems in software engineering, and (2) it provides a framework for reasoning about approximation and correctness, not only for reference immutability type systems, but for the larger class of flow type systems as well.

This paper makes the following contributions.

- We present ReM, a novel type system for reference immutability. ReM captures explicitly definite mutability (in the CFL-reachability sense), and approximation.
- We interpret reference immutability in terms of CFL-reachability, and prove ReIm and ReM correct.
- We present an implementation and evaluation. We show that ReIm and ReM are precise – only 13% of mutable references (6% of all references) are `maybe` mutable. The implementation is publicly available online and has been evaluated and accepted by the ECOOP Artifact Evaluation committee.

The rest of the paper is organized as follows. Sect. 2 presents the mutability semantics based on CFL-reachability. Sect. 3 interprets ReIm in terms of the mutability semantics, and presents the novel system ReM. Sect. 4 establishes equivalence between the systems in Sects. 2 and 3. Sect. 5 presents the empirical evaluation, Sect. 6 discusses related work, and Sect. 7 concludes.

2 Mutability Semantics

2.1 Flow Graph

The mutability semantics builds a *flow graph* G that represents flow (data) dependences between variables. The nodes in the graph are program variables, e.g., `x`, `y`, `this`, and field access expressions, e.g., `x.f`, `y.f`, `this.f`. The edges capture flow from one variable/field access expression, to another. The goal is to capture deep reference (im)mutability with data dependence paths in G . For example, in

$$x = y; z = x; z.f = w$$

25:4 Definite Reference Mutability

$cd ::= \text{class } C \text{ extends } D \{ \overline{fd} \overline{md} \}$	<i>class</i>
$fd ::= t f$	<i>field</i>
$md ::= t m(t \text{ this}, t x) \{ \overline{t y} s; \text{return } y \}$	<i>method</i>
$s ::= s; s \mid x = \text{new } t \mid x = y \mid x.f = y \mid y = x.f \mid x = y.m(z)$	<i>statement</i>
$t ::= q C$	<i>qualified type</i>

■ **Figure 1** Syntax. C and D are class names, f is a field name, m is a method name, and x , y , and z are names of local variables, formal parameters, or parameter **this**. As in the code examples, **this** is explicit. Qualifiers q range over ReIm/ReM qualifiers (defined in Sect. 3).

y is mutable, because there is a path in G from y to z , which is the receiver of the update at field write $z.f = w$. Throughout the paper we refer to receivers at field writes as *updates*.

We restrict our core language to a “named form” in the style of Vaziri et al. [35, 10]. The language models Java with the syntax in Fig. 1, where the results of instantiations, field accesses, and method calls, are immediately stored in a variable. Without loss of generality, we assume that methods have parameter **this**, and exactly one other formal parameter. Features not strictly necessary are omitted from the formalism, but they are handled correctly in the implementation.

An assignment statement contributes a *direct* (i.e., intraprocedural) edge as follows:

$$x = y \quad \Rightarrow \quad y \xrightarrow{d} x$$

It represents flow from variable y to variable x . Therefore, if x is an update, i.e., there is field write $x.g = z$, the direct edge propagates mutability to reference y .

A field write statement $x.f = y$ contributes a direct edge from y to the field access node $x.f$, and an *approximate* edge from $x.f$ to every $x'.f \in G$, where $x'.f$ is the right-hand-side of a field read $y' = x'.f$. (Without loss of generality we may assume $x' \neq x$.)

$$x.f = y \quad \Rightarrow \quad y \xrightarrow{d} x.f \xrightarrow{a} x'.f$$

We elaborate upon approximate edges shortly. A field read statement $y' = x'.f$ contributes direct edges as follows:

$$y' = x'.f \quad \Rightarrow \quad x' \xrightarrow{d} x'.f \xrightarrow{d} y'$$

Edge $x' \xrightarrow{d} x'.f$ accounts for deep (im)mutability. It links x' to y' , propagating mutability back to x' when y' is update.

Therefore, together a pair of field write $x.f = y$ and field read $y' = x'.f$ contribute a triple

$$x.f = y, y' = x'.f \quad \Rightarrow \quad y \xrightarrow{d} x.f \xrightarrow{a} x'.f \xrightarrow{d} y'$$

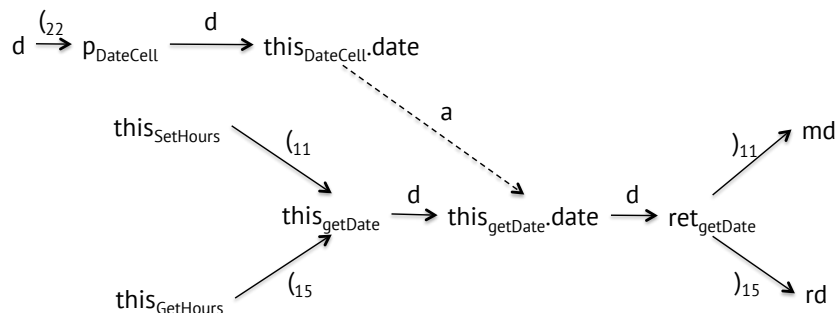
creating a path from y to y' . It models flow through heap objects while completely avoiding heap objects. In terms of ReMs' terminology [25], our mutability semantics, like ReIm, models structure (i.e., heap)-transmitted dependences approximately.

The approximate edge makes approximation explicit. The approximate path from y to y' propagates mutability from y' back to y , but “with an asterisk”. This is *maybe* mutability – if $x.f$ and $x'.f$ are aliases because x and x' point to the same object, then y is truly mutable, however, if they are not aliases, y is not mutable due to this path. ReIm (and other reference immutability systems) overapproximate, and mark y mutable. A key insight of our work is that the impact of approximate paths is quite muted. Generally, when there is an approximate

```

1  class DateCell {
2    Date date;
3
4    DateCell(DateCell this, Date p) {
5      this.date = p;
6    }
7    Date getDate(DateCell this) {
8      return this.date;
9    }
10   void cellSetHours(DateCell this) {
11     Date md = this.getDate();
12     md.setHours(1);
13   }
14   int cellGetHours(DateCell this) {
15     Date rd = this.getDate();
16     int hour = rd.getHours();
17     return hour;
18   }
19
20   public static void main(String[] args) {
21     Date d = new Date();
22     DateCell dc = new DateCell(d);
23     ...
24   }
25 }

```



■ **Figure 2** Running example. Code (adapted from Huang et al. [18]) and corresponding graph.

path from a reference y to an update, there is also a direct path from y to an update, and y would have become mutable regardless of the approximate path.

A method call (method entry) creates the expected *call* edges from actual arguments to formal parameters:

$$i: x = y.m(z) \Rightarrow y \xrightarrow{(i)} \text{this} \quad z \xrightarrow{(i)} p$$

Here *this* and *p* are the parameters of the compile-time target of the call. The standard CFL-reachability annotation $(_i$ marks call entry at call site i . A method return (method exit) creates a *return* edge from the return value to the left-hand-side of the call assignment:

$$i: x = y.m(z) \Rightarrow \text{ret} \xrightarrow{)_i} x$$

The standard CFL-reachability annotation $)_i$ marks a return at site i . In terms of Reps' terminology, the semantics models call-transmitted dependences precisely.

$$\begin{aligned}
E & ::= R' \mid R \mid C \mid M \\
R' & ::= RC \\
R & ::=)_i \mid)_i M \mid)_i R \mid MR \\
C & ::= (_i \mid (_i M \mid (_i C \mid MC \\
M & ::= d \mid (_i M)_i \mid MM
\end{aligned}$$

■ **Figure 3** A context-free grammar for exact paths, i.e., paths that account (solely) for call-transmitted dependences. M captures matched-parentheses strings, e.g., $(_i d)_i$, C captures strings with one or more outstanding calls, e.g., $(_i d (_j d)_j$, and R and R' capture strings with one or more outstanding returns, e.g., $d)_j$.

Since the goal is to capture dependences between variables, the semantics eschews objects and object creation. Fig. 2 shows an example including all kinds of statements and their corresponding edges.

2.2 Paths in Flow Graph

We classify paths in G into two categories: (1) *exact* paths, which do not contain approximate edges, and (2) *approximate* paths, which contain approximate edges. In our running example in Fig. 2, $\text{this}_{\text{getDate}} \rightsquigarrow \text{rd}$ is an exact path, while $d \rightsquigarrow \text{md}$ is an approximate path. (We use squiggle arrows \rightsquigarrow to denote multi-edge paths.) Not all paths in G are well-formed, and different well-formed paths have different meaning.

2.2.1 Exact Paths

Fig. 3 defines a context-free grammar that classifies exact paths into 3 categories. This grammar is standard in CFL-reachability theory. There is an M -path from node n to node u if and only if the edge annotations on the path form a string in the language described by M . M -paths are paths with matched parentheses. For example, path $\text{this}_{\text{GetHours}} \rightsquigarrow \text{rd}$ is an M -path. However, $\text{this}_{\text{GetHours}} \rightsquigarrow \text{md}$ is not a well-formed path because call edge $(_{15}$ and return edge $)_{11}$ do not match.

There is a C -path from n to u if and only if the edges from n to u form a string in the language described by C . More intuitively, these are paths with outstanding call edges. For example, $\text{this}_{\text{GetHours}} \rightsquigarrow \text{ret}_{\text{getDate}}$ is a C -path. With respect to reference immutability, if there is an M -path or a C -path from x (or from $x.f$) to an update, then x (or $x.f$) is definitely mutable, in the sense that analysis generally cannot improve from mutable. Because M -paths and C -paths have the same effect, from now on we refer to them as $M|C$ -paths.

The third category is R -paths. There is an R -path from n to u if and only if the edges form a string in R or R' . That is, the path starts with outgoing return edges, and it may or may not descend into a call path before reaching u . For example, $\text{this}_{\text{getDate}} \rightsquigarrow \text{md}$ is an R -path. With respect to reference immutability, if there is an R -path from x (or $x.f$) to an update, then x (or $x.f$) is *polymorphic*. It is mutable in some contexts of invocation of the enclosing method, and readonly in other. For example, $\text{this}_{\text{getDate}}$ and $\text{ret}_{\text{getDate}}$ are polymorphic. They are interpreted as mutable when getDate is called from cellSetHours , and they are interpreted as readonly when getDate is called from cellGetHours .

2.2.2 Approximate Paths

The following grammar rules capture approximate paths:

$$A ::= a \mid a E \mid a A \mid E A$$

There is an A -path from n to u if and only if the edges form a string in A . Pictorially, an A -path consists of exact paths and approximate edges. For example,

$$n \xrightarrow{E} \cdot \xrightarrow{a} \cdot \xrightarrow{E} \cdot \xrightarrow{a} \cdot \xrightarrow{E} u$$

is an A -path, and so is

$$n \xrightarrow{a} \cdot \xrightarrow{E} \cdot \xrightarrow{a} u$$

The only mandatory component of the A -path is the one approximate edge.

A -paths fall into two categories, $(M|C)A$ -paths, and RA -paths determined by the leading exact path:

- (1) if the leading exact path is an $M|C$ -path, then there is a $(M|C)A$ -path. For example,

$$d \xrightarrow{(20)} p_{\text{DateCell}} \xrightarrow{d} \text{this}_{\text{DateCell}}.\text{date} \xrightarrow{a} \text{this}_{\text{DateCell}}.\text{date} \xrightarrow{d} \text{ret}_{\text{getDate}} \xrightarrow{9} \text{md}$$

is a $(M|C)A$ -path.

- (2) if the leading exact path is an R -path, then there is an RA -path. For the rest of the paper we use the term R -path to denote both the exact R -path and the RA -path as they have the same effect for our purposes.

Standard reference immutability type systems (e.g., ReIm), conservatively mark **mutable** every reference x , such that there is a $(M|C)A$ -path from x to an update. As we mentioned earlier, an approximate path introduces uncertainty rather than definite mutability. The key observation of our work is the following. The majority of references x that exhibit an A -path from x to an update, also exhibit a “parallel” M -path or C -path to a (potentially different) update. Therefore, x is indeed definitely mutable and the A -path has no ill impact; an analysis that attempts to handle A -paths, i.e., structure-transmitted dependences, more precisely would not do better regarding x . Roughly speaking, our analysis separates the A -paths that do exhibit a “parallel” path to an update, from the A -paths that do not, thus separating references that are definitely mutable, from ones that are *maybe mutable*. If an analysis that treats structure-transmitted dependences more precisely is to realize precision improvement, the improvement is bounded by the number of maybe mutable references.

3 Type Systems

This section presents two reference immutability type systems, Huang et al.’s [18] ReIm, and our novel proposal ReM. ReIm captures *definite reference immutability*, that is, **readonly** references in ReIm are guaranteed immutable, however, **mutable** references are not necessarily mutable. ReM captures definite immutability and definite mutability – in ReM **readonly** references are still guaranteed immutable, and in addition, **mutable** references are guaranteed mutable (in the CFL-reachability sense).

The reader may wonder why one needs type-based reference immutability like ReIm and Javari, when one has a clear semantics expressed in terms of standard CFL-reachability. First, type-based reference immutability is studied extensively in the literature [34, 39, 40,

18, 13, 22]; its connection to CFL-reachability brings new insights. Second, type-based reference immutability allows programmers to specify immutability requirements with type qualifiers, e.g., `readonly x`, and take advantage of systems such as JSR 308 and the Checker Framework (<https://checkerframework.org/>) to check these immutability requirements; such requirements cannot be easily expressed or checked using CFL-reachability. Third, type systems promote modularity, while CFL-based systems are typically whole-program analyses. Yet another advantage comes when reasoning about complexity. While CFL-reachability is $O(N^3)$, ReIm/ReM inference is $O(N^2)$, where N is the program size.

Sect. 3.1 outlines ReIm, largely following Huang et al. [18]. We add a new interpretation in terms of our mutability semantics. Sect. 3.2 builds ReM upon the discussion in Sect. 3.1. Sect. 3.3 discusses type inference for ReIm and ReM.

3.1 ReIm

3.1.1 ReIm Qualifiers

The ReIm type system has three immutability qualifiers: `mutable`, `readonly`, and `poly`. We explain the qualifiers in terms of the mutability semantics defined in Sect. 2.

- **mutable**: A mutable reference `x` can be used to mutate the referenced object. This is the implicit and only option in standard object-oriented languages. In terms of our mutability semantics, a `mutable` reference denotes an $M|C$ -path, or a $(M|C)A$ -path from `x` to an update.
- **readonly**: `readonly` captures “deep” immutability. A `readonly` reference `x` cannot be used to mutate the referenced object nor anything it references. All of the following are forbidden:
 - `x.f = y`
 - `x.set(z)` where `set` sets a field of its receiver `x`
 - `z = id(x); z.f = w`
 - `y = x.f; y.g = z`

In terms of the mutability semantics, a `readonly` reference means that there does not exist either an exact or an approximate path to an update.

- **poly**: This qualifier expresses polymorphism over immutability. `poly` denotes that a reference is interpreted as mutable in some contexts, and it is interpreted as immutable in other contexts. The enclosing method *does not* mutate the reference, however, mutation to the reference or one of its components may happen after return. In terms of the mutability semantics, a `poly` reference denotes that there is an R -path from `x` to an update – the reference “flows” out of its enclosing method where it is mutated in some caller context.

The subtyping relation between the qualifiers is

`mutable <: poly <: readonly`

where $q_1 <: q_2$ denotes q_1 is a subtype of q_2 . For example, it is allowed to assign a `mutable` reference to a `poly` or `readonly` one, but it is not allowed to assign a `readonly` reference to a `poly` or `mutable` one.

$$\begin{array}{c}
\text{(TASSIGN)} \\
\frac{\Gamma(x) = q_x \quad \Gamma(y) = q_y \quad q_y <: q_x}{\Gamma \vdash x = y} \\
\\
\text{(TWRITE)} \\
\frac{\Gamma(x) = q_x \quad q_x = \text{mutable} \quad \Gamma(y) = q_y \quad \text{typeof}(f) = q_f \quad q_y <: q_x \triangleright q_f}{\Gamma \vdash x.f = y} \\
\\
\text{(TREAD)} \\
\frac{\Gamma(x) = q_x \quad \Gamma(y) = q_y \quad \text{typeof}(f) = q_f \quad q_x \triangleright q_f <: q_y}{\Gamma \vdash y = x.f} \\
\\
\text{(TCALL)} \\
\frac{\Gamma(x) = q_x \quad \Gamma(y) = q_y \quad \Gamma(z) = q_z \quad \text{typeof}(m) = q_{\text{this}}, q_p \rightarrow q_{\text{ret}} \quad q_y <: q_x \triangleright q_{\text{this}} \quad q_z <: q_x \triangleright q_p \quad q_x \triangleright q_{\text{ret}} <: q_x}{\Gamma \vdash x = y.m(z)}
\end{array}$$

■ **Figure 4** Typing rules. Function *typeof* retrieves the qualifiers of fields and methods. Γ is a type environment that maps variables to their immutability qualifiers.

3.1.2 Typing Rules

ReIm is independent of the Java type system, which allows us to specify typing rules solely over type qualifiers q . The typing rules, following [18] are presented in Fig. 4. Rule (TASSIGN) is straightforward. It requires that the left-hand-side is a supertype of the right-hand-side. The system does not enforce object immutability and only `mutable` objects are created. The object creation rule becomes redundant and we omit it, just as we did in Sect. 2.

Rules (TREAD), (TWRITE) and (TCALL) make use of *viewpoint adaptation*, a concept from Universe Types [8, 9, 7]. Viewpoint adaptation of a type q' from the point of view of another type q , results in the adapted type q'' . This is written as $q \triangleright q' = q''$.

Below, we explain viewpoint adaptation in terms of the mutability semantics. At field accesses (TREAD) and (TWRITE) \triangleright adapts the field f from the viewpoint of (context of) the receiver. Viewpoint adaptation at field access handles structure-transmitted dependences, *approximately*. At method calls \triangleright adapts formal parameters and the return value from the point of view of the *variable at the left-hand-side* of the call assignment. This variable captures the calling context i . Viewpoint adaptation at calls handles call-transmitted dependences, *precisely*.

Notably, ReIm restricts fields to `readonly` or `poly`. Javari [34] does allow for `mutable` fields, increasing expressiveness and allowing Javari to express common idioms such as caching. However, `mutable` fields complicates the system. Declaring a field `mutable` in Javari excludes it from the state of the enclosing object, and adaptation of a `mutable` field requires special

25:10 Definite Reference Mutability

treatment, as discussed in [34, 18]. One can similarly allow mutable fields in ReIm/ReM. However, we are interested in type inference, and allowing mutable fields would create ambiguity: if a field access expression $x.f$ is inferred **mutable**, do we infer that field f is **mutable** and is excluded from the state of a **readonly** x , or do we infer that f is just a “regular” field and a mutable $x.f$ signals deep mutation of x and x must be **mutable**? Restricting fields to $\{\text{readonly}, \text{poly}\}$ chooses the latter, as there is no way to know, without programmer annotations, which fields are caches and thus excluded from the object state. Javarifier [23], Javari’s inference tool, makes the same choice. Javari is more expressive than ReIm, but its “inferable” semantics appears to be the same as ReIm’s: Huang et al. [18] report essentially identical inference result for Javarifier and ReIm.

Following [18], we define \triangleright as follows:

$$\begin{aligned} _ \triangleright \text{mutable} &= \text{mutable} \\ _ \triangleright \text{readonly} &= \text{readonly} \\ q \triangleright \text{poly} &= q \end{aligned}$$

The underscore denotes a “don’t care” value. Qualifiers **mutable** and **readonly** do not depend on the viewpoint. Qualifier **poly** depends on the viewpoint (context), and is substituted by that viewpoint (context).

Let us take a closer look at rules (TWRITE) and (TREAD). For a pair of field write $x.f = y$ and field read $y' = x'.f$, the rules entail the following constraints:

$$q_y <: q_x \triangleright q_f \quad q_{x'} \triangleright q_f <: q_{y'}$$

Suppose y' is an update, i.e., there is statement $y'.f = z$, and $q_{y'}$ is thus **mutable**. Therefore, q_f must be **poly**. First, recall that $q_f \in \{\text{readonly}, \text{poly}\}$. Since **readonly** adapts to **readonly**, $q_{x'} \triangleright q_f <: \text{mutable}$ does not type check, locking $q_f = \text{poly}$. (TWRITE) sets q_x , the type of the receiver to **mutable**. This serves two purposes in ReIm, (1) to account for the update of x , and (2) to account for the structure-transmitted dependence, i.e, the approximate path from y to the (eventual) update y' . Thus, $q_x \triangleright q_f$ evaluates to **mutable**, forcing q_y to be **mutable** as well. As mentioned earlier, ReIm handles approximate paths conservatively. If there is an (M|C)A-path from a reference y to an update, then ReIm’s rules force y to be **mutable**, as is the case above, even though x and x' may refer to different runtime objects.

Now, consider rule (TCALL). Function *typeof* retrieves the type of compile-time target m . q_{this} is the type of parameter **this**, q_p is the type of the formal parameter, and q_{ret} is the type of the return. Rule (TCALL) requires $q_x \triangleright q_{\text{ret}} <: q_x$, which accounts for R -paths. The constraint disallows the return value of m from being **readonly** when there is a call to m , $x = y.m(z)$, where left-hand-side x is **mutable**. Only if the left-hand-sides of all call assignments to m are **readonly**, can the return type of m be **readonly**; otherwise, it is **poly**. A programmer can annotate the return type of m as **mutable**. However, this typing is pointless, as it unnecessarily forces local variables and parameters in m to become **mutable** when they may remain less restrictively **poly**. In Fig. 2, $\text{md} = \text{this.getDate}()$; entails constraint

$$q_{\text{md}} \triangleright q_{\text{ret}_{\text{getDate}}} <: q_{\text{md}} \quad \equiv \quad \text{mutable} \triangleright q_{\text{ret}_{\text{getDate}}} <: \text{mutable}$$

leading to $q_{\text{ret}_{\text{getDate}}} = \text{poly}$. This accounts for the R -path from **ret** to the update through **md**. Continuing with the example, the field read in `DateCell.getDate` (line 8) entails constraint

$$q_{\text{this}_{\text{getDate}}} \triangleright q_{\text{date}} <: q_{\text{ret}_{\text{getDate}}}$$

leading to $q_{\text{this}_{\text{getDate}}} = \text{poly}$, which accounts for the R -path from $\text{this}_{\text{getDate}}$ to the update **md**.

Additionally, rule (TCALL) requires $q_y <: q_x \triangleright q_{\text{this}}$. When q_{this} is *readonly* or *mutable*, its adapted value is the same. Thus, when q_{this} is *mutable* (e.g., due to `this.f = 0` in `m`),

$q_y <: q_x \triangleright q_{\text{this}}$ becomes $q_y <: \text{mutable}$

which disallows q_y from being anything but *mutable*, as expected. This accounts for *C*- and *CA*-paths. The interesting case is when q_{this} is *poly*. A *poly* parameter `this` reflects a dependence between `this` and `ret` of `m`, such as the one in Fig. 2:

```
Date getDate(Date this) { ret = this.date; }
```

It allows the `this` object (or some part of it, in our example the `date` part of it), to be modified in caller context, after `m`'s return. The type system entails that whenever there is intraprocedural dependence between `this` and `ret`, we have

$q_{\text{this}} <: q_{\text{ret}}$.

Recall that when there exists a context where the left-hand-side variable `x` is mutated, q_{ret} must be *poly*. Therefore, constraint $q_{\text{this}} <: q_{\text{ret}}$ forces q_{this} to be *poly* (assuming that `this` is not mutated in the context of its enclosing method). Rule (TCALL) adds the 2 constraints “around” $q_{\text{this}} <: q_{\text{ret}}$ to capture call-transmitted dependences:

$q_y <: q_x \triangleright q_{\text{this}}$ $q_{\text{this}} <: q_{\text{ret}}$ $q_x \triangleright q_{\text{ret}} <: q_x$

When `m` is called in a *mutable* context, i.e., q_x is *mutable*, q_y becomes *mutable*, as expected. Conversely, when `m` is called in a *readonly* context, i.e., q_x is *readonly*, $q_x \triangleright q_{\text{this}}$ evaluates to *readonly*, leaving q_y unchanged. In terms of our mutability semantics, this behavior captures *M*- and *MA*-paths.

3.2 ReM

We now present the ReM type system, which builds upon ReIm.

3.2.1 ReM Qualifiers

The ReM type system adds to the set of ReIm qualifiers, and changes the meaning of some of the ReIm qualifiers. There are 5 qualifiers in ReM: ReIm's *mutable*, *readonly* and *poly*, and two new, *maybe* and *polymaybe*. Again, we interpret the qualifiers in terms of the mutability semantics defined in Sect. 2.

- **mutable**: A *mutable* reference `x` is now definitely *mutable*. It denotes that there is an (*M|C*)-path from `x` to an update.
- **readonly**: A *readonly* reference `x` has the same meaning as in ReIm, i.e., there is neither an exact nor an approximate path to an update.
- **maybe**: A *maybe* reference denotes that there is a (*M|C*)*A*-path to an update, but there is no *R*-path to an update.
- **poly**: A *poly* reference now denotes that there is an *R*-path to an update, but there is no (*M|C*)*A*-path.
- **polymaybe**: A *polymaybe* reference denotes that there is an *R*-path to update, *and* a (*M|C*)*A*-path to update.

The subtyping hierarchy is as follows:

$\text{mutable} <: \text{polymaybe} <: \begin{array}{c} \text{maybe} \\ \text{poly} \end{array} <: \text{readonly}$

3.2.2 Typing Rules

ReM rules extend ReIm. There are two extensions: (1) viewpoint adaptation must account for new qualifiers `maybe` and `polymaybe`, and (2) rule (TWRITE) must account for approximate paths.

Viewpoint adaptation rules from Sect. 3.1 remain in effect. We add two new rules:

$$\begin{aligned} _ \triangleright \text{maybe} &= \text{maybe} \\ q \triangleright \text{polymaybe} &= (q \triangleright \text{poly}) \wedge \text{maybe} \end{aligned}$$

Notation \wedge stands for the standard meet operation: the result of $q_1 \wedge q_2$ is the greatest lower bound of q_1 and q_2 in the lattice of ReM types above.

Since field and return types are restricted to $\{\text{readonly}, \text{poly}\}$, adaptation of `maybe` or `polymaybe` happens *only when adapting parameters at method calls*.

Recall that a `maybe` parameter p denotes a $(M|C)A$ -path from p . Thus, call $x = y.m(z)$ creates an $(M|C)A$ path from z (a CA -path to be precise). Rule (TCALL) requires

$$q_z <: q_x \triangleright q_p \quad \equiv \quad q_z <: \text{maybe}$$

which accounts for the $(M|C)A$ -path from z .

Now recall that a `polymaybe` parameter p denotes an R -path to update, and an $(M|C)A$ -path to a (possibly different) update. These paths entail paths from z : one through `ret`, depending on the left-hand-side of the call assignment, and an $(M|C)A$ path. Rule (TCALL) applies viewpoint adaptation of q_p , essentially recording the more conservative choice at the caller. Consider (TCALL) constraint

$$q_z <: q_x \triangleright q_p.$$

Suppose that x is `readonly`, and there is no path from the left-hand-side of the call assignment x to update (that is, the R -path from p is due to a different call site). Then there is no new path from z to update through $p \rightsquigarrow \text{ret} \xrightarrow{i} x$. However, there is an $(M|C)A$ -path from z to update. Viewpoint adaptation accounts for it:

$$q_z <: (\text{readonly} \triangleright \text{poly}) \wedge \text{maybe} \quad \equiv \quad q_z <: \text{maybe}$$

Conversely, suppose x is `mutable`, and there is an $M|C$ -path from x to update. Then the R -path leads to an $M|C$ -path from z to update. Viewpoint adaptation accounts for this:

$$q_z <: (\text{mutable} \triangleright \text{poly}) \wedge \text{maybe} \quad \equiv \quad q_z <: \text{mutable}$$

Consider the more detailed example:

```

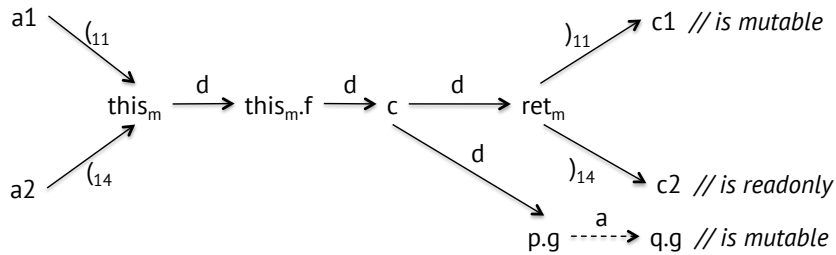
1 class A {
2   ...
3   C m(A this, B p) {
4     C c = this.f;
5     p.g = c;
6     return c;
7   }
8 }

```

```

9  public static void main(String[] args) {
10 ...
11  C c1 = a1.m(b);
12  c1.setField();
13  ...
14  C c2 = a2.m(b);
15  int i = c2.getField();
16  ....
17  }
18  }

```



The R -path from $this_m$ to $c1$, entails $q_{this} <: poly$, while the $(M|C)A$ -path through $p.g$ entails $q_{this_m} <: maybe$. Thus, $q_{this_m} = polymaybe$. At call 11 we have

$$q_{a1} <: q_{c1} \triangleright q_{this_m} \equiv q_{a1} <: q_{c1} \triangleright polymaybe.$$

Since q_{c1} is mutable, $polymaybe$ adapts to mutable, setting q_{a1} to mutable. On the other hand, at call 14 we have

$$q_{a2} <: q_{c2} \triangleright q_{this_m} \equiv q_{a2} <: q_{c2} \triangleright polymaybe \equiv q_{a2} <: maybe.$$

Thus, since q_{c2} is readonly, the meet is $maybe$, and q_{a2} is precisely $maybe$.

We now change rule (TWRITE) to account for approximate paths:

$$\frac{\begin{array}{l} \text{(TWRITE)} \\ \Gamma(x) = q_x \quad q_x = \text{mutable} \quad \Gamma(y) = q_y \\ \text{typeof}(f) = q_f \quad q_y <: \text{maybe} \triangleright q_f \end{array}}{\Gamma \vdash x.f = y}$$

q_x remains mutable to account for the direct update on x . However, instead of adapting by $mutable$ context as in ReIm, we adapt by $maybe$. This reflects the approximate path, which ReIm conservatively made mutable. If q_f is $readonly$, then the $maybe$ -mutability of x does not affect y . If q_f is $poly$, that reflects an update to some f , and $q_y <: maybe \triangleright poly$ propagates the “ $maybe$ ” update to y .

3.3 Type Inference

Type inference for both ReIm and ReM proceeds as outlined in [16, 18] and earlier in [19, 33]. We present novel treatment in terms of dataflow frameworks, and include necessary extensions for ReM.

The inference operates on mappings from keys to values S . The keys in the mapping are (1) local variables and parameters, (2) fields, and (3) method returns. The values in the

25:14 Definite Reference Mutability

mapping are *sets* of type qualifiers. For instance, $S(x) = \{\text{poly}, \text{mutable}\}$ in ReIm means the type of reference x can be *poly* or *mutable*.

S is initialized as follows. $S(\text{ret}) = S(f) = \{\text{readonly}, \text{poly}\}$ for all return values ret and fields f . The rest of the variables are initialized to the universal set of qualifiers U , which is $\{\text{readonly}, \text{poly}, \text{mutable}\}$ in ReIm, and $\{\text{readonly}, \text{maybe}, \text{poly}, \text{polymaybe}, \text{mutable}\}$ in ReM. For the rest of this paper we use U to refer to either ReIm or ReM. We denote the initial mapping by S_0 .

The inference iterates over all statements s in the program and removes qualifiers inconsistent with the typing rule for s from S . More precisely, let s consist of variables v_1, v_2, \dots, v_k and let s entail transfer function $c(s)$. Applying $c(s)$ removes each q_1 from $S(v_1)$ when there are no qualifiers $q_2 \in S(v_2), \dots, q_k \in S(v_k)$, such that q_1, q_2, \dots, q_k make s type check; then it removes all q_2 from $S(v_2)$, etc. For example, consider $s: y = x.f$ and corresponding rule (TREAD) triggering constraint $q_x \triangleright q_f <: q_y$. Let $S(x), S(f)$ and $S(y)$ be as follows:

$S(x)$	$S(f)$	$S(y)$
$\{\text{maybe}, \text{polymaybe}, \text{mutable}\}$	$\{\text{readonly}, \text{poly}\}$	$\{\text{poly}, \text{polymaybe}, \text{mutable}\}$

$c(s)$ removes *maybe* from $S(x)$ because there does not exist $q_f \in S(f)$ and $q_y \in S(y)$ that satisfy $\text{maybe} \triangleright q_f <: q_y$. Similarly, it removes *readonly* from $S(f)$. After application of $c(s)$:

$S'(x)$	$S'(f)$	$S'(y)$
$\{\text{polymaybe}, \text{mutable}\}$	$\{\text{poly}\}$	$\{\text{poly}, \text{polymaybe}, \text{mutable}\}$

One can easily prove that, given S_0 as shown, $c(s)$ only need look at the left-hand-side of the constraint, i.e., the right-hand-side always remains unchanged.

The inference analysis iterates over the statements in the program and removes qualifiers from the sets until it reaches a fixpoint. The problem fits into the standard monotone dataflow framework [1, 20]. The lattice L_v for variables v is

$$\{\text{mutable}\} > \{\text{polymaybe}, \text{mutable}\} > \{\text{maybe}, \text{polymaybe}, \text{mutable}\} > U > \{\text{poly}, \text{polymaybe}, \text{mutable}\}$$

and the dataflow lattice L is the product lattice of all L_v lattices, which is standard. Initializing all variables to U corresponds to initializing with the 0 of the lattice. The function space is $L \rightarrow L$ and is monotone. This is a theorem that one can easily show by case-by-case analysis of each $c(s)$. Therefore, the result of fixpoint iteration is the maximal fixpoint solution. Call this solution S_{Fix} . Yet the fixpoint solution is a mapping from references to sets. The actual mapping from references to types is derived as follows: for each reference x we pick the *maximal* element of $S_{Fix}(x)$ according to the following ranking, which mirrors the subtyping lattice:

$$\begin{array}{ccccccc} & & \text{maybe} & & & & \\ \text{readonly} & > & & > & \text{polymaybe} & > & \text{mutable} \\ & & \text{poly} & & & & \end{array}$$

Importantly, the maximal element exists because each $S_{Fix}(x)$ is an element of L_v . We denote this typing by $\text{max}(S_{Fix})$, and call it the *maximal typing*.

The following propositions state that (1) the maximal typing type checks, and (2) the maximal typing is the “best typing”. (Note that setting all references to *mutable* also type checks, but makes up a useless typing.)

► **Proposition 1.** *ReIm’s $\text{max}(S_{Fix})$ and ReM’s $\text{max}(S_{Fix})$ always type check.*

Proof Sketch. The proof for ReIm is given in [18]. The proof for ReM proceeds by case-by-case analysis. The most difficult case arises at $q_z <: q_x \triangleright q_p$. Let $S_{Fix}(\mathbf{p}) = \{\text{poly}, \text{polymaybe}, \text{mutable}\}$ and $S_{Fix}(\mathbf{z}) = \{\text{maybe}, \text{polymaybe}, \text{mutable}\}$. $S_{Fix}(\mathbf{x})$ must be either $\{\text{maybe}, \text{polymaybe}, \text{mutable}\}$ or $\{\text{readonly}, \text{maybe}, \text{poly}, \text{polymaybe}, \text{mutable}\}$. If it were any other set, then *maybe* would have been removed from $S(\mathbf{z})$ during fixpoint iteration. Combinations $q_z = \text{maybe}$, $q_x = \text{maybe}$, $q_p = \text{poly}$, and $q_z = \text{maybe}$, $q_x = \text{readonly}$, $q_p = \text{poly}$, maximal typings under the two cases, both typecheck.

A more general statement is true. For every S that satisfies the equations of the dataflow framework, typing $\text{max}(S)$ type checks. ◀

Previous work in [16] formalized the notion of “best typing” for ownership type systems, specifically Ownership types [5] and Universe Types [8], by using a heuristic ranking over typings. This formalization applies to ReIm/ReM, as well as other ownership-like type systems, e.g., AJ [35] and EnerJ [28]. Below we extend the treatment of [16] to ReM.

We say that T is a *valid typing* if T type checks. Objective function o ranks valid typings. o takes a valid typing T and returns a tuple of numbers. For ReIm, o is as follows:

$$o_{ReIm}(T) = (|T^{-1}(\text{readonly})|, |T^{-1}(\text{poly})|, |T^{-1}(\text{mutable})|)$$

The tuples are ordered lexicographically. We have $T_1 > T_2$ iff T_1 has more *readonly* references than T_2 , or T_1 and T_2 have the same number of *readonly* references, but T_1 has more *poly* references than T_2 . The preference ranking over typings is based on ranking over qualifiers: naturally, we prefer *readonly* over *poly* and *mutable*, and *poly* over *mutable*.

ReM’s objective function is the following:

$$o_{ReM}(T) = (|T^{-1}(\text{readonly})|, |T^{-1}(\text{poly})| + |T^{-1}(\text{maybe})|, |T^{-1}(\text{polymaybe})|, |T^{-1}(\text{mutable})|)$$

Again the tuples are ordered based on the natural ranking over qualifiers: *readonly* is the most preferred, followed by *poly* and *maybe*, which are equally preferred, and so on. The following proposition establishes that the maximal typing is the best typing.

► **Proposition 2.** *Let o be the objective function over valid typings (either o_{ReIm} over ReIm, or o_{ReM} over ReM). $o(\text{max}(S_{Fix})) > o(T)$ holds for every valid typing $T \neq \text{max}(S_{Fix})$.*

Proof Sketch. The fact that the maximal typing is the “best typing”, follows from the properties of monotone dataflow frameworks. Let S be another solution of the dataflow framework. (It is easy to see that if T is valid typing, it must be contained into a solution of the dataflow framework.) Since $S > S_{Fix}$ by virtue of S_{Fix} being the maximal fixpoint solution, for every variable x $S(x) \geq S_{Fix}(x)$, and there exist variables y such that $S(y) > S_{Fix}(y)$. Thus, $o_{ReM}(\text{max}(S_{Fix})) > o_{ReM}(\text{max}(S))$. ◀

4 Equivalence

This section formally links the mutability semantics and the maximal typing. Specifically, we establish equivalence between CFL-reachability, as outlined in Sect. 2, and the maximal typing as outlined in Sect. 3.3 and previous work [16, 18, 19, 33].

Fig. 5 states the algorithms for CFL-reachability and type inference explicitly. CFL initializes graph G to \emptyset , then iterates over the program statements adding paths to updates, until no more paths can be added. TYPES initializes S to the 0 of the lattice, then iterates over the statements, removing qualifiers from S until no more qualifiers can be removed. To

25:16 Definite Reference Mutability

<pre> 1: procedure CFL 2: $G = \emptyset$ 3: Add $u \overset{M}{\rightsquigarrow} u$ to G for all updates u 4: while G changes do 5: for each s in Program do 6: EDGE($e(s)$) 7: end for 8: end while 9: end procedure </pre>	<pre> 1: procedure TYPES 2: $S(n) = U$ 3: $S(n) = \{\text{mutable}\}$ for all updates n 4: while S changes do 5: for each s in Program do 6: CONSTRAINT($c(s)$) 7: end for 8: end while 9: end procedure </pre>
<pre> 1: procedure EDGE($n \overset{t}{\rightarrow} n'$) 2: for each $n' \overset{N}{\rightsquigarrow} u \in G$ do 3: Add $n \overset{t \oplus N}{\rightsquigarrow} u$ to G 4: end for 5: end procedure </pre>	<pre> 1: procedure CONSTRAINT($l <: r$) 2: Remove each q_l from $S(l)$ 3: if $\nexists q_r \in S(r)$ s.t. $q_l <: q_r$ 4: end procedure </pre>

■ **Figure 5** Algorithm CFL initializes G , then iterates over program statements s adding edges as specified in Sect. 2.1. Algorithm TYPES initializes S , then iterates over program statements s removing qualifies from S as specified in Sect. 3.3. The algorithms elide details to highlight the “parallel” structure of the two systems.

emphasize the parallel structure, Fig. 5 simplifies the presentation. Most notably, recall that according to Sect. 2 field read $y' = x'.f$ accounts for two edges:

$$y' = x'.f \quad \Rightarrow \quad x' \xrightarrow{d} x'.f \xrightarrow{d} y'$$

Even though Fig. 5 shows a single invocation of EDGE, in fact CFL processes two edges, first $x'.f \xrightarrow{d} y'$, followed by $x' \xrightarrow{d} x'.f$. Similarly, CFL processes multiple edges at field writes $x.f = y$: for each field read $y' = x'.f$, such that $x'.f \in G$, it processes $x.f \xrightarrow{a} x'.f$, followed by $y \xrightarrow{d} x.f$.

Another detail elided from Fig. 5 is the meaning of t , N and concatenation operator \oplus . t ranges over the terminals: $(i,)_i, d$, and a . N ranges over the kinds of paths: $M|C$, $(M|C)A$, and R . Concatenation $t \oplus N$ applies the grammar rules, and in all but one case, is straightforward:

$$\begin{aligned}
(i \oplus M|C) &= M|C \\
(i \oplus (M|C)A) &= (M|C)A \\
)_i \oplus _ &= R \\
d \oplus N &= N \\
a \oplus _ &= (M|C)A
\end{aligned}$$

$(i \oplus R$ is the difficult case, because it applies rule $M ::= (i M)_i$. EDGE applies concatenation $(i \oplus R$ when processing call edge $z \xrightarrow{(i)} p$ and $p \overset{R}{\rightsquigarrow} u \in G$. (Here p is the parameter of the compile-time target method of the call.) $p \overset{R}{\rightsquigarrow} u \in G$ entails

$$p \overset{M}{\rightsquigarrow} \text{ret} \xrightarrow{)_j} x \overset{N}{\rightsquigarrow} u$$

where ret is the return value of the target method, and x is some left-hand-side of a call assignment. Note that $p \overset{M}{\rightsquigarrow} \text{ret} \xrightarrow{)_j} x$ are not explicitly in G , but $x \overset{N}{\rightsquigarrow} u$ is in G . There are two cases. If there is no edge $)_j$ such that $j = i$, then $(i \oplus R$ adds no new paths; the R -path from p is due to a different call site. Otherwise, that is, when $j = i$, concatenation adds $z \overset{N}{\rightsquigarrow} u$ to G .

Let us illustrate CFL and EDGE, and TYPES and CONSTRAINT in parallel. Consider

$$y = x; z = y; z.f = w$$

Calling EDGE on $y \xrightarrow{d} z$, which corresponds to statement $z = y$, leads to path $y \xrightarrow{M} z$ in G . Subsequently calling EDGE on edge $x \xrightarrow{d} y$ leads to concatenation of d and M and path $x \xrightarrow{M} z$. There are two M -paths, x to z and y to z , as expected. Analogously, calling CONSTRAINT on $q_y <: q_z$, which corresponds to statement $z = y$, removes all qualifiers but mutable from $S(y)$. Subsequently calling CONSTRAINT on $q_x <: q_y$ removes all qualifiers but mutable from $S(x)$. $S(x) = \{\text{mutable}\}$, and $S(y) = \{\text{mutable}\}$ mirror the two M -paths that CFL finds.

The most interesting case arises (as it has been the case throughout the paper), when adding a call edge. Consider code

$$y = \text{id}(x); y = z; z.f = w$$

and assume $y \xrightarrow{M} z$ and $p \xrightarrow{R} z$ are already in G . Concatenation breaks the R -path $p \xrightarrow{R} z$ into $p \xrightarrow{M} \text{ret} \xrightarrow{i} y \xrightarrow{M} z$. Since $(i \text{ and })_i$ match, it adds path $x \xrightarrow{M} z$. Analogously, for TYPES assume $S(p) = \{\text{poly, polymaybe, mutable}\}$ and $S(y) = \{\text{mutable}\}$. Calling CONSTRAINT on $q_x <: q_y \triangleright q_p$ removes all qualifiers but mutable from $S(x)$, which directly corresponds to the M -path from x to z that CFL finds.

To formally establish equivalence we use the bisimulation methodology for proving equivalence between two systems A and B [36, 6]. The methodology requires that we establish a relation that relates states in A to those in B. In our case, A is constructed by CFL-reachability inference (Algorithm CFL), and B is constructed by type inference (Algorithm TYPES). Our approach defines an explicit equivalence relation between the states in A, captured by G , and those in B, captured by S . Intuitively, assume algorithms CFL and TYPES run in “parallel”. To show equivalence we must show that processing s in each system maintains equivalence.

We state two definitions that form the basis of equivalence. Informally, Def. 3 states that for every path from n to update u in G , n is correspondingly typed in S . In Def. 3 n stands for either a variable node x , or field access node $x.f$.

► **Definition 3.** (Soundness) $G \Rightarrow S$ if and only if

1. $n \xrightarrow{M|C} u \in G \Rightarrow \max(S(n)) <: \text{mutable}$
2. $n \xrightarrow{R} u \in G \Rightarrow \max(S(n)) <: \text{poly}$
3. $n \xrightarrow{(M|C)A} u \in G \Rightarrow \max(S(n)) <: \text{maybe}$

Def. 4 states that n 's maximal type in S implies a corresponding path in G . For example, maximal typing `polymaybe` must imply that there are both an R -path and a $(M|C)A$ -path in G , but there is no $M|C$ -path.

► **Definition 4.** (Precision) $S \Rightarrow G$ if and only if

1. $\max(S(x)) = \text{mutable} \Rightarrow \exists x \xrightarrow{M|C} u \in G$
2. $\max(S(x)) = \text{polymaybe} \Rightarrow \exists x \xrightarrow{R} u \in G \wedge \exists x \xrightarrow{(M|C)A} u \in G \wedge \neg \exists x \xrightarrow{M|C} u \in G$
3. $\max(S(x)) = \text{poly} \Rightarrow \exists x \xrightarrow{R} u \in G \wedge \neg \exists x \xrightarrow{(M|C)A} u \in G \wedge \neg \exists x \xrightarrow{M|C} u \in G$
4. $\max(S(x)) = \text{maybe} \Rightarrow \neg \exists x \xrightarrow{R} u \in G \wedge \exists x \xrightarrow{(M|C)A} u \in G \wedge \neg \exists x \xrightarrow{M|C} u \in G$
5. $\max(S(x)) = \text{readonly} \Rightarrow \text{no path from } x \text{ in } G$
6. $\max(S(x.f)) = \text{readonly} \Rightarrow \text{no path from } x.f \text{ in } G$

25:18 Definite Reference Mutability

► **Definition 5.** (Equivalence) $G \simeq S$ if and only if $G \Rightarrow S$ and $S \Rightarrow G$.

Let the following Hoare triple denote parallel execution of EDGE and CONSTRAINT on statement s :

$$\{G, S\} \text{ EDGE}(e(s)) \parallel \text{ CONSTRAINT}(c(s)) \{G', S'\}$$

Our key result is the following theorem:

► **Theorem 6.** *If $G \simeq S$ and $\{G, S\} \text{ EDGE}(e(s)) \parallel \text{ CONSTRAINT}(c(s)) \{G', S'\}$ then $G' \simeq S'$.*

Proof Sketch. As expected, the proof is by induction on the number of applications of

$$\text{EDGE}(e(s)) \parallel \text{ CONSTRAINT}(c(s))$$

Clearly, the statement holds after initialization, lines 2-3 in CFL and lines 2-3 in TYPES. The inductive step requires case-by-case analysis of each s .

To prove correctness, we must show that given $G \Rightarrow S$, after the execution of EDGE($e(s)$) and CONSTRAINT($c(s)$), $G' \Rightarrow S'$ still holds. We outline the most difficult case, EDGE($z \xrightarrow{(i)} p$) \parallel CONSTRAINT($q_z <: q_x \triangleright q_p$) (method call naturally brakes into three steps). Consider $x \xrightarrow{(i)} p \oplus p \xrightarrow{R} u$. Let x be the left-hand-side at call assignment i . If there does not exist a path $x \xrightarrow{N} u \in G$, then no new paths are added to G' and $G' \Rightarrow S'$ holds. If there exists $x \xrightarrow{N} u \in G$, then a new path $z \xrightarrow{N} u$ is added to G' . We must show that $S'(z)$ reflects N according to Def. 3 (e.g., if N is an $M|C$ -path, then z is mutable). By the inductive hypothesis $p \xrightarrow{R} u \in G \Rightarrow \max(S(p)) <: \text{poly}$. Similarly, the N -path entails appropriate $S(x)$: if N is an $M|C$ path, then $S(x) = \{\text{mutable}\}$, if N is an R path then $\max(S(x)) <: \text{poly}$, and if N is a $(M|C)A$ -path, then $\max(S(x)) <: \text{maybe}$. Constraint $q_z <: q_x \triangleright q_p$ removes qualifiers from $S(z)$. For example, if N is $M|C$, then $S(x) = \{\text{mutable}\}$, and it is easy to see that $\max(S(x)) \triangleright \max(S(p)) <: \text{mutable}$. Thus $\max(S'(z)) <: \text{mutable}$, as needed. We enumerate all cases in Sect. A (Proofs).

In the other direction, we must show that if $S \Rightarrow G$ holds, after the execution of EDGE and CONSTRAINT on s , $S' \Rightarrow G'$ still holds. Consider the analogous case, EDGE($z \xrightarrow{(i)} p$) \parallel CONSTRAINT($q_z <: q_x \triangleright q_p$), and the following values of $S(x)$, $S(p)$ and $S(z)$ (only the maximal element shown):

$S(x)$	$S(p)$	$S(z)$
{ readonly, ... }	{ maybe, ... }	{ readonly, ... }

Constraint $q_z <: q_x \triangleright q_p$ “lowers” $S(z)$ into $S'(z) = \{\text{maybe}, \dots\}$. By the inductive hypothesis, $S(x)$ and $S(p)$ entail that there are no paths from x in G , no paths from z in G , and only a $(M|C)A$ -path from p . Therefore, EDGE($z \xrightarrow{(i)} p$) adds an $(M|C)A$ -path from z in G' , and no other kind of path. Thus, $\max(S'(z)) = \text{maybe} \Rightarrow z \xrightarrow{(M|C)A} u$, as expected. We enumerate all cases in Sect. A (Proofs). ◀

For clarity, we omitted method overriding. It is handled in both the mutability semantics and type inference, and equivalence still holds. Concretely, if m' overrides m we add

$$q_{\text{this}_m} \xrightarrow{d} q_{\text{this}_{m'}} \quad q_{p_m} \xrightarrow{d} q_{p_{m'}} \quad q_{\text{ret}_m} \xrightarrow{d} q_{\text{ret}_{m'}}$$

to G . Analogously, we require

$$\text{typeof}(m') <: \text{typeof}(m)$$

which entails

$$(q_{\text{this}_{m'}}, q_{p_{m'}} \rightarrow q_{\text{ret}_{m'}}) <: (q_{\text{this}_m}, q_{p_m} \rightarrow q_{\text{ret}_m})$$

which leads to the standard function subtyping constraints:

$$q_{\text{this}_m} <: q_{\text{this}_{m'}} \quad q_{p_m} <: q_{p_{m'}} \quad q_{\text{ret}_{m'}} <: q_{\text{ret}_m}$$

Our implementation handles function subtyping.

5 Empirical Results

We implemented ReM on top of ReIm. (ReIm is publicly available.) Soot is the underlying platform, and Jimple is the underlying intermediate representation. We evaluate ReM on DaCapo, plus the benchmarks used in Javarifier [23] and ReIm [18]. There are 13 whole programs, and 8 libraries:

- **DaCapo** suite DaCapo-2006-10MR.
- **JOlden** is a classical suite of 10 small whole programs (Javarifier and ReIm).
- **ejc-3.2.0** is the Java Compiler for the Eclipse IDE (Javarifier and ReIm).
- **javad** is a Java disassembler program (ReIm).
- **tinySQL-1.1** is a database engine (Javarifier and ReIm).
- **htmlparser-1.4** is an HTML parser library (Javarifier and ReIm).
- **commons-pool-1.2** is an object pooling library (ReIm).
- **jtids-1.0** is a JDBC driver (ReIm).
- **jdbm-1.0** is a transactional engine (ReIm).
- **jdbf-0.0.1** is an object-oriented mapping system (ReIm).
- **java.lang** and **java.util** are the packages from JDK 1.7.0_75.

All benchmarks are analyzed with JDK 1.7. On whole programs, our analysis relies on the standard Class Hierarchy Analysis (CHA)-based reachability in Soot, which pulls in all relevant packages according to CHA. ReIm/ReM analyzes all these packages. All experiments are done on a MacBook Pro 2.8 GHz Intel Core i7 and 16GB of RAM using default VM settings for everything, including maximal heap size.

Tab. 1 presents the results of running ReM inference on the benchmarks. On average, only 6.4% of all references are inferred as *maybe* or *polymaybe*. They make up only about 13.6% of all ReIm-mutable references while the remaining 86.4% are *definitely mutable*. To assess the impact of the intermediate representation, Soot’s Jimple, which creates a significant number of temporary local variables, we computed statistics on parameter and returns (no local variables). The results show that 5.8% of all references are *maybe* or *polymaybe*, and approximately 84% of all ReIm-mutable parameters and returns are *definitely mutable*. These result is very similar, suggesting that the intermediate representation does not lead to an overestimation of the number of *definitely mutable* references. (In fact, our investigation suggests that it may lead to an underestimation, as we explain shortly.) Running times do not exceed 90 seconds, with most benchmarks completing in under 60 seconds on the commodity laptop described earlier.

In addition to the benchmarks from Tab. 1 we ran our analysis on *Avrora*, *Batik* and *Sunflow* from DaCapo-9.12-MR1-bach; these are whole-program benchmarks were added to

■ **Table 1** Inference results for ReM. **Annotatable References** includes all variables of reference type, including locals, parameters, returns, and fields. It does not include variables of primitive type. Column **#Readonly** shows the number of references inferred as **readonly**, **#Poly** shows the number of variables inferred as **poly**, and **#Maybe/#polymaybe** shows the number of **maybe** and **polymaybe** references, respectively. Column **#Mutable** shows the number of **mutable** references, which are now definitely mutable. In parentheses is the percentage of definitely mutable references over of all potentially mutable ones: $\#Mutable/(\#Maybe+\#Polymaybe+\#Mutable)$.

Benchmark	Annotatable References				Time (sec.)
	#Readonly	#Poly	#Maybe/ #Polymaybe	#Mutable	
antlr	15751	1029	1198/2	12552 (91.3%)	64.3
bloat	12567	2299	3880/56	18844 (82.7%)	36.1
chart	31640	4973	7570/139	32063 (80.6%)	79.3
eclipse	16945	4117	2613/36	12807 (82.9%)	39.1
fop	47411	5563	5578/66	35564 (86.3%)	89.0
hsqldb	29907	4282	3673/31	26054 (87.6%)	66.8
luindex	5465	686	750/11	4604 (85.8%)	30.1
lusearch	6296	945	1019/16	5344 (83.8%)	30.2
pmd	37758	4605	5248/96	34096 (86.5%)	78.4
xalan	21254	3337	2853/63	19771 (87.1%)	75.4
jolden	1208	281	206/0	1069 (83.8%)	29.0
javad	796	45	15/0	417 (96.5%)	26.5
ejc	29062	8202	3612/174	24768 (86.7%)	49.3
tinySQL	6115	665	638/7	3585 (84.8%)	28.1
htmlparser	7787	1215	1049/12	4904 (82.2%)	29.2
commons-pool	847	25	222/0	640 (74.2%)	27.1
jdbm	1134	317	322/9	1584 (82.7%)	28.8
jdbf	4017	312	289/2	2379 (89.1%)	58.6
jtds	8628	825	587/25	5738 (90.4%)	32.5
java.lang	3336	350	249/1	4721 (95.0%)	29.5
java.util	3216	376	309/0	4758 (93.9%)	30.9
Average				86.4%	

DaCapo 2006 for the 2009 suite. Our analysis reports that on average 84.5% of ReIm-mutable references are definitely mutable, which is in line with Tab. 1.²

The results demonstrate that ReM and ReIm are *precise* and *scalable*. They can be used to power inference for approximate computing (e.g. EnerJ [28] and Rely [4]), taint analysis (e.g., DroidInfer [17]), and method purity [18], as well as other client analyses. Even if one designed a more complex system that handled structure-transmitted dependences more precisely, by employing a powerful alias analysis for example, improvement would be at most 5-6% of all references being promoted from mutable (12-13% of ReIm’s mutable references). ReM/ReIm’s complexity is $O(N^2)$, which leads to fast running times.

Finally, to better understand the results, we examined all 15 **maybe** references from javad, and 15 randomly selected **maybe** references from ejc. We looked to identify definite paths to mutation, or more precisely, we examined $y \xrightarrow{d} x.f \xrightarrow{a} x'.f \xrightarrow{d} y'$ and attempted to prove

² We omitted Tomcat, H2 and the Treadsoap benchmarks from DaCapo-9.12-MR1-bach as these are complex client-server programs and we were unable to set the analysis in time for the publication deadline. Recent work in this space [14] omits these program as well. Also due to timing, DaCapo 2009 was not included in Artifact Evaluation.

that there exist a runtime object o such that x points to o , x' points to o , and the value of $x.f$ indeed flows to $x'.f$. We immediately identified such definite paths in 16 out of 30 cases. The remaining 14 cases exhibited difficult data and control flow, and we could not identify definite paths. A typical case of obvious definite paths was the following. Consider this typical code for initializing an array field f :

```

1  f = new X[10];
2  for (int i=0; i<cnt; i++)
3    f[i] = new X();
4  ...

```

This code snippet is translated into the following Jimple:

```

1  r1 = newarray (X)[10];
2  this.f = r1;
3  ...
4  r2 = this.f;
5  x = new X();
6  r2[i] = x;

```

Mutation of the array is captured by the approximate path, and the `maybe` typing of $r1$. This case leads to an overestimation of the number of `maybe` variables and the following simple optimization reduces the number of `maybe/polymaybe` references. Specifically, at each field write `this.f = r1` we add constraint $q_{r1} <: q_{m.this.f}$ where m is the enclosing method and $m.this.f$ is a dummy variable. Similarly, at each field read `r2 = this.f` we add constraint $q_{m.this.f} <: q_{r2}$. Thus, when $r2$ is `mutable` or `poly/polymaybe`, `mutable` or `poly/polymaybe` propagates through $q_{m.this.f}$, and the analysis demotes $r1$ to `mutable` or `polymaybe`. Tab. 2 shows the results of this optimization – on average 87.3% of mutable references are now definitely mutable. As with DaCapo 2009 the optimization was added for the final version and was not part of Artifact Evaluation.

Another source of `maybe` mutability is containers. E.g., in

```

1  class Container {
2    Data data;
3    void set(Container this, Data p) {
4      this.data = p;
5    }
6    Data get(Container this) {
7      return this.data;
8    }
9  }

```

parameter p of `set` is rightfully `maybe` mutable. p and the `data` object will be mutable in some clients of `Container` and `readonly` in others.

6 Related Work

The most closely related work is Huang et al.'s `ReIm` and `ReImInfer` [18]. Our work builds upon `ReIm` and `ReImInfer` but extends them in two directions. First, we build a theoretical framework that interprets `ReIm` and `ReM` in terms of CFL-reachability, and we prove them correct within this simple framework. To the best of our knowledge, `ReIm` has not been proven correct even though it has been used to power client analyses [17, 32]. Second, we propose `ReM` and definite mutability, which extends the expressive power of `ReIm`, and also,

■ **Table 2** Results after optimization. 87.3% of ReIm-mutable references are definitely mutable.

Benchmark	Annotatable References			#Mutable	Time (sec.)
	#ReadOnly	#Poly	#Maybe/ #PolyOrMaybe		
antlr	15751	1029	1093/6	12653 (92.0%)	128.2
bloat	12567	2299	3683/68	19029 (83.5%)	38.1
chart	31640	4973	7329/154	32289 (81.2%)	86.5
eclipse	16945	4118	2473/46	12937 (83.7%)	38.3
fop	47411	5563	5235/93	35880 (87.1%)	93.2
hsqldb	29907	4282	3303/37	26418 (88.8%)	86.8
luindex	5465	686	677/13	4675(87.1%)	31.6
lusearch	6296	945	932/17	5430 (85.1%)	35.9
pmd	37758	4605	4989/106	34345 (87.1%)	83.3
xalan	21254	3337	2634/71	19982 (88.1%)	79.8
jolden	1208	281	187/9	1080 (84.6%)	29.0
javad	796	45	3/0	429 (99.3%)	28.4
ejc	29062	8202	3146/194	25214 (88.3%)	51.7
tinySQL	6115	665	608/7	3615 (85.5%)	118.7
htmlparser	7787	1215	1023/12	4930(82.6%)	114.1
commons-pool	847	25	222/0	642 (74.5%)	25.6
jdbm	1134	317	293/9	1613(84.2%)	28.4
jdbf	4017	312	281/2	2387 (89.4%)	116.3
jtds	8628	825	552/25	5773(90.1%)	36.7
java.lang	3336	350	217/1	4753(95.6%)	34.1
java.util	3216	376	276/0	4791 (94.6%)	37.1
Average				87.3%	

establishes a bound on (im)precision. Our empirical results show that ReIm/ReM are highly precise and highly scalable.

Reference immutability has been an active area of research for many years. Tschantz et al. propose Javari [34] and Javari’s inference tool Javarifier [23]. Javari is more expressive and more complex than ReIm, but the inferable features are essentially the same. (Huang et al. report that Javarifier and ReImInfer produce essentially the same result.) Zibin et al.’s IGJ [39] and OIGJ [40] are type systems that support reference immutability and object immutability. Haack and Poll [15] propose a type system for object immutability as well. Gordon et al. [13] propose a reference immutability system for safe parallelism. Potanin et al. [22] survey work on reference and object immutability, and method purity. As it is standard, these systems support definite immutability (like ReIm). They do not attempt to estimate precision (imprecision), or connect reference immutability and CFL-reachability.

Artzi et al. [2] propose a hybrid static and dynamic analysis for inference of parameter reference immutability. In contrast, our work focuses on static analysis.

Salcianu and Rinard’s JPPA [31] and Pearce’s JPure [21] infer method purity for Java. ReIm/ReM is more general, in the sense that it enables reasoning about method purity, as well as other client analyses (e.g., EnerJ [28] and DroidInfer [17]). The fact that ReIm/ReM is precise, suggests that client analyses would be precise as well.

CFL-reachability is a standard program analysis framework [25]. Rehof and Fahndrich [24] connect type-based flow analysis and CFL-reachability. This is similar to our interpretation of type-based reference immutability in terms of CFL-reachability. However, Rehof and Fahndrich do not discuss mutable references and it is unclear how they handle such references or structure-transmitted dependences. Fahndrich et al. [11] apply the theory of [24] to build

a context-sensitive Steensgard-style points-to analysis for C, thus using equality constraints instead of subtyping constraints. (Equality constraints is the standard approach to the handling of mutable references [29, 28, 12], as we mention earlier.) Our work focuses specifically on reference immutability and reasoning about its precision. The result that ReIm/ReM is precise, indicates that they can be incorporated into flow analyses [29, 28, 12, 17].

Sridharan and Bodik [30] present refinement-based points-to analysis for Java using CFL-reachability. Xu et al. [37] improve the scalability of CFL-reachability-based points-to analysis. These works focus on points-to analysis and require heap abstraction. Therefore, they inherit known issues with reflection. Type-based reference immutability and the parallel CFL-reachability analysis avoid heap abstraction and thus, they completely avoid issues due to reflective object creation (`x = Class.forName("className").newInstance()`), for free. We still face issues with reflective method invocation (`getMethod`). However, reflective object creation is by far most common, and has been studied extensively in the points-to analysis community. Recent work by Zhang and Su [38] propose new approximation algorithms based on CFL-reachability that can handle both structure-transmitted and call transmitted dependences precisely. Our work focuses on type-based reference immutability, for which handling of structure-transmitted dependences approximately appears sufficient.

7 Conclusion

We presented ReM, a novel reference immutability type system. ReM separated potentially mutable references into definitely mutable, and maybe mutable, i.e., references that may be mutable due to inherent approximation. In addition, we proposed a CFL-reachability system for reference immutability, thus building a novel framework for reasoning about correctness of reference immutability type systems. We implemented ReM and showed that about 86.5% of all potentially mutable references were definitely mutable.

References

- 1 Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.
- 2 Shay Artzi, Adam Kiezun, David Glasser, and Michael D. Ernst. Combined static and dynamic mutability analysis. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering, ASE '07*, pages 104–113, New York, NY, USA, 2007. ACM. doi:10.1145/1321631.1321649.
- 3 Joseph A. Bank, Andrew C. Myers, and Barbara Liskov. Parameterized types for java. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '97*, pages 132–145, New York, NY, USA, 1997. ACM. doi:10.1145/263699.263714.
- 4 Michael Carbin, Sasa Misailovic, and Martin C. Rinard. Verifying quantitative reliability for programs that execute on unreliable hardware. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, pages 33–52, 2013.
- 5 David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, OOPSLA '98*, pages 48–64, New York, NY, USA, 1998. ACM. doi:10.1145/286936.286947.

- 6 Rance Cleaveland and Matthew Hennessy. Testing equivalence as a bisimulation equivalence. In *Automatic Verification Methods for Finite State Systems, International Workshop, Grenoble, France, June 12-14, 1989, Proceedings*, pages 11–23, 1989. doi:10.1007/3-540-52148-8_2.
- 7 Dave Cunningham, Werner Dietl, Sophia Drossopoulou, Adrian Francalanza, Peter Müller, and Alexander J. Summers. Formal methods for components and objects. In Frank S. Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul Roever, editors, *Formal Methods for Components and Objects, 6th International Symposium, FMCO 2007, Amsterdam, The Netherlands, October 24-26, 2007, Revised Lectures*, chapter Universe Types for Topology and Encapsulation, pages 72–112. Springer-Verlag, Berlin, Heidelberg, 2008. doi:10.1007/978-3-540-92188-2_4.
- 8 Werner Dietl and Peter Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology*, 4(8):5–32, 2005.
- 9 Werner Dietl and Peter Müller. Runtime universe type inference. In *IWACO*, pages 72–80, 2007. URL: http://sct.ethz.ch/projects/student_docs/Frank_Lyner/Frank_Lyner_MA_paper.pdf.
- 10 Julian Dolby, Christian Hammer, Daniel Marino, Frank Tip, Mandana Vaziri, and Jan Vitek. A data-centric approach to synchronization. *ACM Transactions on Programming Languages and Systems*, 34(1):1–48, apr 2012.
- 11 Manuel Fähndrich, Jakob Rehof, and Manuvir Das. Scalable context-sensitive flow analysis using instantiation constraints. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation, PLDI '00*, pages 253–263, New York, NY, USA, 2000. ACM. doi:10.1145/349299.349332.
- 12 Robert Fuhrer, Frank Tip, Adam Kiezun, Julian Dolby, and Markus Keller. Efficiently refactoring java applications to use generic libraries. In *Proceedings of the 19th European Conference on Object-Oriented Programming, ECOOP'05*, pages 71–96, Berlin, Heidelberg, 2005. Springer-Verlag. doi:10.1007/11531142_4.
- 13 Colin S. Gordon, Matthew J. Parkinson, Jared Parsons, Aleks Bromfield, and Joe Duffy. Uniqueness and reference immutability for safe parallelism. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '12*, pages 21–40, New York, NY, USA, 2012. ACM. doi:10.1145/2384616.2384619.
- 14 Neville Grech and Yannis Smaragdakis. P/taint: unified points-to and taint analysis. *PACMPL*, 1(OOPSLA):102:1–102:28, 2017. doi:10.1145/3133926.
- 15 Christian Haack and Erik Poll. Type-based object immutability with flexible initialization. In *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, Genoa, pages 520–545, Berlin, Heidelberg, 2009. Springer-Verlag. doi:10.1007/978-3-642-03013-0_24.
- 16 Wei Huang, Werner Dietl, Ana Milanova, and Michael D. Ernst. Inference and checking of object ownership. In *Proceedings of the 26th European Conference on Object-Oriented Programming, ECOOP'12*, pages 181–206, Berlin, Heidelberg, 2012. Springer-Verlag. doi:10.1007/978-3-642-31057-7_9.
- 17 Wei Huang, Yao Dong, Ana Milanova, and Julian Dolby. Scalable and precise taint analysis for android. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015*, pages 106–117, New York, NY, USA, 2015. ACM. doi:10.1145/2771783.2771803.
- 18 Wei Huang, Ana Milanova, Werner Dietl, and Michael D. Ernst. Reim & reiminfer: Checking and inference of reference immutability and method purity. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Lan-*

- guages and Applications*, OOPSLA '12, pages 879–896, New York, NY, USA, 2012. ACM. doi:10.1145/2384616.2384680.
- 19 Adam Kiezun, Michael D. Ernst, Frank Tip, and Robert M. Fuhrer. Refactoring for parameterizing java classes. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 437–446, Washington, DC, USA, 2007. IEEE Computer Society. doi:10.1109/ICSE.2007.70.
 - 20 Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of program analysis*. Springer-Verlag New York, Inc., 1999.
 - 21 David J. Pearce. Jpure:: A modular purity system for java. In *Proceedings of the 20th International Conference on Compiler Construction: Part of the Joint European Conferences on Theory and Practice of Software*, CC'11/ETAPS'11, pages 104–123, Berlin, Heidelberg, 2011. Springer-Verlag. URL: <http://dl.acm.org/citation.cfm?id=1987237.1987247>.
 - 22 Alex Potanin, Johan Östlund, Yoav Zibin, and Michael D. Ernst. Immutability. In *Aliasing in Object-Oriented Programming*, volume 7850 of *LNCS*, pages 233–269. Springer-Verlag, apr 2013.
 - 23 Jaime Quinonez, Matthew S. Tschantz, and Michael D. Ernst. Inference of reference immutability. In *Proceedings of the 22Nd European Conference on Object-Oriented Programming*, ECOOP '08, pages 616–641, Berlin, Heidelberg, 2008. Springer-Verlag. doi:10.1007/978-3-540-70592-5_26.
 - 24 Jakob Rehof and Manuel Fähndrich. Type-base flow analysis: From polymorphic subtyping to cfl-reachability. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '01, pages 54–66, New York, NY, USA, 2001. ACM. doi:10.1145/360204.360208.
 - 25 Thomas Reps. Undecidability of context-sensitive data-independence analysis. *ACM Transactions on Programming Languages and Systems*, 22(1):162—186, 2000.
 - 26 Thomas W. Reps. Program analysis via graph reachability. *Information & Software Technology*, 40(11-12):701–726, 1998. doi:10.1016/S0950-5849(98)00093-7.
 - 27 Thomas W. Reps, Susan Horwitz, and Shmuel Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995*, pages 49–61, 1995. doi:10.1145/199448.199462.
 - 28 Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. Enerj: approximate data types for safe and general low-power computation. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 164–174, 2011.
 - 29 Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting format string vulnerabilities with type qualifiers. In *Proceedings of the 10th Conference on USENIX Security Symposium - Volume 10*, SSYM'01, pages 16–16, Berkeley, CA, USA, 2001. USENIX Association. URL: <http://dl.acm.org/citation.cfm?id=1267612.1267628>.
 - 30 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*, PLDI '06, pages 387–400, New York, NY, USA, 2006. ACM. doi:10.1145/1133981.1134027.
 - 31 Alexandru Sălcianu and Martin Rinard. Purity and side effect analysis for java programs. In *Proceedings of the 6th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI'05, pages 199–215, Berlin, Heidelberg, 2005. Springer-Verlag. doi:10.1007/978-3-540-30579-8_14.
 - 32 Rishi Surendran and Vivek Sarkar. Automatic parallelization of pure method calls via conditional future synthesis. In *Proceedings of the 2016 ACM SIGPLAN International Con-*

- ference on *Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, pages 20–38, New York, NY, USA, 2016. ACM. doi:10.1145/2983990.2984035.
- 33 Frank Tip, Robert M. Fuhrer, Adam Kiezun, Michael D. Ernst, Ittai Balaban, and Bjorn De Sutter. Refactoring using type constraints. *ACM Transactions on Programming Languages and Systems*, 33(3):1–47, 2011. doi:10.1145/1961204.1961205.
 - 34 Matthew S. Tschantz and Michael D. Ernst. Javari: Adding reference immutability to java. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 211–230, New York, NY, USA, 2005. ACM. doi:10.1145/1094811.1094828.
 - 35 Mandana Vaziri, Frank Tip, Julian Dolby, Christian Hammer, and Jan Vitek. A type system for data-centric synchronization. In *Proceedings of the 24th European Conference on Object-oriented Programming*, ECOOP'10, pages 304–328, Berlin, Heidelberg, 2010. Springer-Verlag. URL: <http://dl.acm.org/citation.cfm?id=1883978.1884000>.
 - 36 Yuepeng Wang, Isil Dillig, Shuvendu K. Lahiri, and William R. Cook. Verifying equivalence of database-driven applications. *Proc. ACM Program. Lang.*, 2(POPL):56:1–56:29, dec 2017. doi:10.1145/3158144.
 - 37 Guoqing Xu, Atanas Rountev, and Manu Sridharan. Scaling cfl-reachability-based points-to analysis using context-sensitive must-not-alias analysis. In *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, Genoa, pages 98–122, Berlin, Heidelberg, 2009. Springer-Verlag. doi:10.1007/978-3-642-03013-0_6.
 - 38 Qirun Zhang and Zhendong Su. Context-sensitive data-dependence analysis via linear conjunctive language reachability. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 344–358, New York, NY, USA, 2017. ACM. doi:10.1145/3009837.3009848.
 - 39 Yoav Zibin, Alex Potanin, Mahmood Ali, Shay Artzi, Adam Kie, un, and Michael D. Ernst. Object and reference immutability using java generics. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, pages 75–84, New York, NY, USA, 2007. ACM. doi:10.1145/1287624.1287637.
 - 40 Yoav Zibin, Alex Potanin, Paley Li, Mahmood Ali, and Michael D. Ernst. Ownership and immutability in generic java. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 598–617, New York, NY, USA, 2010. ACM. doi:10.1145/1869459.1869509.

A Proofs

Our main theorem follows from the following two lemmas.

► **Lemma 7.** *If $G \Rightarrow S$ and $\{G, S\} \text{EDGE}(e(s)) \parallel \text{CONSTRAINT}(c(s)) \{G', S'\}$ then $G' \Rightarrow S'$.*

Proof. The proof relies on the fact that viewpoint adaptation preserves subtyping. That is, for each x, x' and p, p' , $x <: x' \Rightarrow x \triangleright p <: x' \triangleright p$. Also, for each x, p and $p', p' <: p \Rightarrow x \triangleright p <: x \triangleright p'$. Therefore, for each $x, x', p,$ and p' , $x <: x' \wedge p <: p' \Rightarrow x \triangleright p <: x' \triangleright p'$.

We proceed by induction on the number of applications of

$\text{EDGE}(e(s)) \parallel \text{CONSTRAINT}(c(s))$

and case by case analysis.

Consider the most difficult case, case 1: s is $x = y.m(z)$. Naturally, it breaks into 3 smaller cases, (1) $z \xrightarrow{(i)} p$, (2) $y \xrightarrow{(i)}$ this and (3) $\text{ret} \xrightarrow{(i)} x$. (2) is analogous to (1).

For (1), suppose EDGE adds $z \xrightarrow{i \oplus M|C} u$ to G' . By the inductive hypothesis, $p \xrightarrow{M|C} u$ implies $\max(S(p)) <: \text{mutable}$. The corresponding constraint $q_z <: q_x \triangleright q_p$ sets $\max(S'(z)) = \text{mutable}$. Similarly, $p \xrightarrow{(M|C)^A} u \Rightarrow \max(S(p)) <: \text{maybe}$ and constraint $q_z <: q_x \triangleright q_p$ leads to $q_z <: q_x \triangleright \text{maybe} <: \text{maybe}$ by the above theorem. Thus, $\max(S'(z)) <: \text{maybe}$, as needed. Now, suppose that (1) adds $z \xrightarrow{i \oplus R} u$ to G' . This entails $\max(S(\text{ret})) <: \text{poly}$. If $x \xrightarrow{M|C} u \in G$ then $\max(S(x)) <: \text{mutable}$, leading to $\max(S'(z)) <: \text{mutable}$. If $x \xrightarrow{(M|C)^A} u \in G$ then $\max(S(x)) <: \text{maybe}$, leading to $\max(S'(z)) = \text{maybe}$, as needed. If $x \xrightarrow{R} u \in G$ then $\max(S(x)) <: \text{poly}$; constraint $q_z <: q_x \triangleright q_p \equiv q_z <: \text{poly} \triangleright \text{poly}$ leads to $\max(S'(z)) <: \text{poly}$, as needed.

For (3), suppose EDGE adds $\text{ret} \xrightarrow{i \oplus N} u$ to G' . This happens only if there is $x \xrightarrow{N} u \in G$. Therefore by the inductive hypothesis $\max(S(x)) \neq \text{readonly}$, and therefore, constraint $q_x \triangleright \text{ret} <: q_x$ entails that $\max(S'(\text{ret})) <: \text{poly}$, as needed.

Case 2: s is $x = y$ is straightforward.

Case 3: s is $x.f = y$. Suppose EDGE processes approximate edge $x.f \xrightarrow{a} x'.f$ followed by direct edge $y \xrightarrow{d} x.f$. EDGE adds to G' only if $x'.f$ in G , which by the inductive hypothesis entails $S(f) = \{\text{poly}\}$. Thus, $\max(S(x.f))$ evaluates to mutable , and the desired subtyping is preserved (even though this typing is not precise). Furthermore, field write constraint $q_y <: q_x \triangleright q_f$ evaluates to $q_y <: \text{maybe} \triangleright \text{poly}$, meaning that $\max(S(y)) <: \text{maybe}$ as needed to account for the $(M|C)^A$ -path from y in G' .

Finally, consider case 4: s is $y = x.f$. There is new path in G' only if there is a path in G from y . If there is a path from y , then $\max(S(y)) \neq \text{readonly}$. The constraint for field write $q_x \triangleright q_f <: q_y$ entails $\max(S(f)) = \text{poly}$ and $\max(S(x)) <: \max(S(y))$. Therefore, $\max(S(x.f))$ and $\max(S(x))$ reflect the new paths through y . ◀

► **Lemma 8.** *If $S \Rightarrow G$ and $\{G, S\} \text{EDGE}(e(s)) \parallel \text{CONSTRAINT}(c(s)) \{G', S'\}$ then $S' \Rightarrow G'$.*

Proof. The proof is by induction on the number of applications of

$$\text{EDGE}(e(s)) \parallel \text{CONSTRAINT}(c(s))$$

We begin with the most difficult case, case 1: s is $x = y.m(z)$.

The tables below examine all cases for parameter constraint $q_z <: q_x \triangleright q_p$. Constraint $q_y <: q_x \triangleright q_{\text{this}}$ is completely analogous. For brevity, sets S show only the maximal element. For example, when

$$S(x) = \{\text{readonly}, \dots\} \quad S(p) = \{\text{maybe}, \dots\} \quad S(z) = \{\text{readonly}, \dots\}$$

the constraint $q_z <: q_x \triangleright q_p$ removes readonly and poly from $S(z)$ resulting in:

$$S'(z) = \{\text{maybe}, \dots\}$$

To preserve precision, we must show that after execution of the parallel EDGE operation, there exist only an $(M|C)^A$ -path from z to an update. The last column of the table enumerates the paths from z : added by $\text{EDGE}(z \xrightarrow{i} p)$ given $S(x)$ and $S(p)$, and existing ones in G . Continuing with the example, $(M|C)^A[p] + \text{none}[\text{ret}] + \text{none}[z]$ reads as follows: (1) an $(M|C)^A$ -path is added through p (the inductive hypothesis $S \Rightarrow G$ entails that the maybe typing of p implies an $(M|C)^A$ -path from p), and no paths are added through ret and x (again, the maybe typing implies that there is no path through ret and x), and (2) there are

25:28 Definite Reference Mutability

no existing paths from z (due to the readonly typing of z in S). Therefore, there is only an $(M|C)A$ -path from z , and this case preserves precision.

The cases of $S(z) = \{\text{mutable}\}$ and $S(p) = \{\text{readonly}, \dots\}$ are not shown because neither of these cases triggers change to S , and it is trivial to argue $S' \Rightarrow G'$.

The following table enumerates the cases for $S(x) = \{\text{readonly}, \dots\}$:

$S(x)$	$S(p)$	$S(z)$	$S'(z)$	$G, \text{EDGE}(z \xrightarrow{i} p)$
{ readonly,	{ maybe,			$(M C)A[p] + \text{none}[\text{ret}]$
		{ readonly, { maybe, { poly, { polymaybe,	{ maybe, NO CHANGE { polymaybe, NO CHANGE	$+\text{none}[z]$ $+(M C)A[z]$ $+R[z]$ $+(M C)A[z] + R[z]$
{ readonly,	{ poly,			$\text{none}[p] + \text{none}[x]$
		{ readonly, { maybe, { poly, { polymaybe,	NO CHANGE NO CHANGE NO CHANGE NO CHANGE	$+\text{none}[z]$ $+(M C)A[z]$ $+R[z]$ $+(M C)A[z] + R[z]$
{ readonly,	{ polymaybe,			$(M C)A[p] + \text{none}[x]$
		{ readonly, { maybe, { poly, { polymaybe,	{ maybe, NO CHANGE { polymaybe, NO CHANGE	$+\text{none}[z]$ $+(M C)A[z]$ $+R[z]$ $+(M C)A[z] + R[z]$

The table below enumerates the cases for $S(x) = \{\text{maybe}, \dots\}$:

$S(x)$	$S(p)$	$S(z)$	$S'(z)$	$G, \text{EDGE}(z \xrightarrow{i} p)$
{ maybe,	{ maybe,			$(M C)A[p] + \text{none}[\text{ret}]$
		{ readonly, { maybe, { poly, { polymaybe,	{ maybe, NO CHANGE { polymaybe, NO CHANGE	$+\text{none}[z]$ $+(M C)A[z]$ $+R[z]$ $+(M C)A[z] + R[z]$
{ maybe,	{ poly,			$\text{none}[p] + (M C)A[x]$
		{ readonly, { maybe, { poly, { polymaybe,	{ maybe, NO CHANGE { polymaybe NO CHANGE	$+\text{none}[z]$ $+(M C)A[z]$ $+R[z]$ $+(M C)A[z] + R[z]$
{ maybe,	{ polymaybe,			$(M C)A[p] + (M C)A[x]$
		{ readonly, { maybe, { poly, { polymaybe,	{ maybe, NO CHANGE { polymaybe NO CHANGE	$+\text{none}[z]$ $+(M C)A[z]$ $+R[z]$ $+(M C)A[z] + R[z]$

The table below enumerates the cases for $S(x) = \{\text{poly}, \dots\}$:

$S(x)$	$S(p)$	$S(z)$	$S'(z)$	$G, \text{EDGE}(z \xrightarrow{i} p)$
{ poly,	{ maybe,			$(M C)A[p] + \text{none}[\text{ret}]$
		{ readonly, { maybe, { poly, { polymaybe,	{ maybe, NO CHANGE { polymaybe, NO CHANGE	$+\text{none}[z]$ $+(M C)A[z]$ $+R[z]$ $+(M C)A[z] + R[z]$
{ poly,	{ poly,			$\text{none}[p] + R[x]$
		{ readonly, { maybe, { poly, { polymaybe,	{ poly, { polymaybe NO CHANGE NO CHANGE	$+\text{none}[z]$ $+(M C)A[z]$ $+R[z]$ $+(M C)A[z] + R[z]$
{ poly,	{ polymaybe,			$(M C)A[p] + R[x]$
		{ readonly, { maybe, { poly, { polymaybe,	{ polymaybe, { polymaybe, { polymaybe, NO CHANGE	$+\text{none}[z]$ $+(M C)A[z]$ $+R[z]$ $+(M C)A[z] + R[z]$

The table below enumerates the cases for $S(x) = \{\text{polymaybe}, \dots\}$:

$S(x)$	$S(p)$	$S(z)$	$S'(z)$	$G, \text{EDGE}(z \xrightarrow{i} p)$
{ polymaybe,	{ maybe,			$(M C)A[p] + \text{none}[\text{ret}]$
		{ readonly, { maybe, { poly, { polymaybe,	{ maybe, NO CHANGE { polymaybe, NO CHANGE	$+\text{none}[z]$ $+(M C)A[z]$ $+R[z]$ $+(M C)A[z] + R[z]$
{ polymaybe,	{ poly,			$\text{none}[p] + R[x] + (M C)A[x]$
		{ readonly, { maybe, { poly, { polymaybe,	{ polymaybe, { polymaybe, { polymaybe, NO CHANGE	$+\text{none}[z]$ $+(M C)A[z]$ $+R[z]$ $+(M C)A[z] + R[z]$
{ polymaybe,	{ polymaybe,			$(M C)A[p] + (M C)A[x] + R[x]$
		{ readonly, { maybe, { poly, { polymaybe,	{ polymaybe, { polymaybe, { polymaybe, NO CHANGE	$+\text{none}[z]$ $+(M C)A[z]$ $+R[z]$ $+(M C)A[z] + R[z]$

Consider constraint $q_x \triangleright q_{\text{ret}} <: q_x$. If $S(x)$ is $\{\text{readonly}, \dots\}$ then there is no change to S and no change to G , and the statement holds. If $S(\text{ret})$ is $\{\text{poly}\}$, then, there is no change to S and EDGE “adds” a path already in G , resulting in no change in G as well. Let $S(x)$ be any other value but $\{\text{readonly}, \dots\}$ and let $S(\text{ret})$ be $\{\text{readonly}, \dots\}$. $S'(\text{ret})$ becomes $\{\text{poly}\}$ implying an R -path. Since $S(x)$ is of any other value but $\{\text{readonly}, \dots\}$, this means that a path from x to update, $x \xrightarrow{N} u$ does exist in G , and $\text{EDGE}(\text{ret} \xrightarrow{i} x)$ results in R path from ret in G' . Therefore, the theorem holds.

Consider case 2, s is $x = y$. We enumerate all possibilities analogously. Again we omit the cases when $S(x) = \{\text{mutable}, \dots\}$ as well as the case when $S(y) = \{\text{readonly}, \dots\}$, as they are trivial.

25:30 Definite Reference Mutability

$S(x)$	$S(y)$	$S'(y)$	$G, \text{EDGE}(y \xrightarrow{d} x)$
{ maybe,	{ readonly, { maybe, { poly, { polymaybe,	{ maybe, NO CHANGE { polymaybe, NO CHANGE	$(M C)A[x]+none[y]$ $(M C)A[x]+(M C)A[y]$ $(M C)A[x]+R[y]$ $(M C)A[x]+(M C)A[y]+R[y]$
{ poly,	{ readonly, { maybe, { poly, { polymaybe,	{ poly, { polymaybe, NO CHANGE NO CHANGE	$R[x]+none[y]$ $R[x]+(M C)A[y]$ $R[x]+R[y]$ $R[x]+(M C)A[y]+R[y]$
{ polymaybe,	{ readonly, { maybe, { poly, { polymaybe,	{ polymaybe, { polymaybe, { polymaybe, NO CHANGE	$(M C)A[x]+R[x]+none[y]$ $(M C)A[x]+R[x]+(M C)A[y]$ $(M C)A[x]+R[x]+R[y]$ $(M C)A[x]+R[x]+(M C)A[y]+R[y]$

Now consider case 3, s is $x.f = y$, and corresponding constraints $q_y <: \text{maybe} \triangleright q_f$. If f is **readonly**, then **maybe** $\triangleright q_f$ is **readonly**, and there is no change in S . By the inductive hypothesis, $x'.f$ being **readonly** implies that there does not exist a read $x'.f$ such that there is a path from $x'.f$ to update in G . Thus, no path is added to G' through $x.f$ thus preserving the paths from y and the theorem. If f is **poly**, then y becomes **maybe**, or lower in S' , thus properly accounting for the $(M|C)A$ -path from y through $x.f$ that appears in G' .

Finally, consider case 4. s is $y' = x'.f$. If f is **readonly**, then there does not exist a path from $x'.f$ in G . If y' is not **readonly**, then there exists a path form y' . Thus, f becomes **poly** in S' , and $x' <: y'$ in S' . In G' , there are new paths from $x'.f$ and x reflecting S' . ◀

Efficient Reflection String Analysis via Graph Coloring

Neville Grech

Dept. of Informatics and Telecommunications, University of Athens, Greece
and Dept. of Computer Science, University of Malta, Malta
me@nevillegrech.com

George Kastrinis

Dept. of Informatics and Telecommunications, University of Athens, Greece
gkastrinis@di.uoa.gr

Yannis Smaragdakis

Dept. of Informatics and Telecommunications, University of Athens, Greece
yannis@smaragd.org

Abstract

Static analyses for reflection and other dynamic language features have recently increased in number and advanced in sophistication. Most such analyses rely on a whole-program model of the flow of strings, through the stack and heap. We show that this global modeling of strings remains a major bottleneck of static analyses and propose a compact encoding, in order to battle unnecessary complexity. In our encoding, strings are maximally merged if they can never serve to differentiate class members in reflection operations. We formulate the problem as an instance of graph coloring and propose a fast polynomial-time algorithm that exploits the unique features of the setting (esp. large cliques, leading to hundreds of colors for realistic programs). The encoding is applied to two different frameworks for string-guided Java reflection analysis from past literature and leads to significant optimization (e.g., a $\sim 2x$ reduction in the number of string-flow inferences), for a whole-program points-to analysis that uses strings.

2012 ACM Subject Classification Software and its engineering \rightarrow Compilers, Theory of computation \rightarrow Program analysis, Software and its engineering \rightarrow General programming languages

Keywords and phrases reflection, static analysis, graph coloring

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2018.26

Acknowledgements We gratefully acknowledge funding by the European Research Council, grant 307334 (SPADE), a Facebook Research and Academic Relations award, and an Oracle Labs collaborative research grant. In addition, the research work disclosed is partially funded by the REACH HIGH Scholars Program – Post-Doctoral Grants. The grant is part-financed by the European Union, Operational Program II, Cohesion Policy 2014-2020 (Investing in human capital to create more opportunities and promote the wellbeing of society - European Social Fund). Grants.

1 Introduction

Reflection is a language feature that enables the dynamic discovery of an object's type structure (e.g., its fields and supported methods) and full access to the object's state and functionality through dynamically-discovered members. Reflection is not merely one of the dynamic features of a statically-typed language but typically the backbone connecting all dynamic features. For instance, in Java, the most common facility for dynamic code



© Neville Grech, George Kastrinis, and Yannis Smaragdakis;
licensed under Creative Commons License CC-BY

32nd European Conference on Object-Oriented Programming (ECOOP 2018).

Editor: Todd Millstein; Article No. 26; pp. 26:1–26:25

Leibniz International Proceedings in Informatics



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

generation is the *dynamic proxy* pattern [26] (recently estimated to appear in 21% of open-source Java programs [17]), which requires the use of reflection in order to provide a generic implementation of an interface.

Modeling reflection has been a challenge for the static analysis of languages like Java and C#. Ignoring reflection operations during static analysis is one of the top causes of analysis unsoundness [22]. (Or, equivalently, one of the most common assumptions in any claim of soundness for an analysis is the lack of reflection.) Modeling reflection operations statically has attracted much recent research effort [7, 18, 19, 21, 23, 30, 34]. Virtually all of these models have a similar general structure, first explored by Livshits et al. [21, 23]: they model reflection in the context of a whole-program value-flow analysis (such as a points-to analysis, with a full model of the heap) so that the flow of parameters of reflective actions can be approximated. The initial such parameters are merely string values. For instance, a call to `java.lang.Class.getMethod()` takes as argument the name of the method to be looked up dynamically. By examining the flow of string values (which may partially match class, method, or field names) into reflective calls, the analysis can do a first approximation of the effects of reflective actions. This model can then be refined with extra information, such as the use patterns of objects produced via reflection.

Static analysis for reflection, therefore, crucially depends on modeling the flow of string *constants*. For instance, consider a string constant "put" that flows into a string concatenation operation (possibly with statically-unknown parameters), whose result flows to a `getMethod` call. The constant string yields significant information as to which method(s) may be selected dynamically. Such information is typically the differentiator between an infeasibly imprecise static analysis and one that can reliably guess an overapproximation of reflection results.

Tracking strings through a whole-program analysis can be very expensive, however. Type filtering is ineffective, since the `String` type is not elaborated into more detailed subtypes in most languages. This observation holds for a vast array of whole-program static analyses and is surprising in scope. For instance a context-insensitive analysis of the *avrora* DaCapo-Bach benchmark in the DOOP framework [30] computes a points-to set of 2.9 million strings and just 2 million regular objects. Similar effects are found throughout other static analyses that model reflection: a 0-1-CFA analysis with reflection support on the IBM WALA library [7], over a medium-sized benchmark (`ant1r`, from the DaCapo 2006 suite), yields a total points-to set with 6.7 million strings and just 1.7 million non-strings objects (i.e., all other object types together). This effect is surprising: an analysis that only incidentally models the flow of strings (in order to model reflection operations) ends up being dominated by string values, in comparison to all other objects whose flow is the real analysis target.

In this paper, we propose a technique for collapsing string constants so that they impose minimal overhead in a whole-program value-flow analysis, yet retain their ability to act as member selectors for all reflection operations. Specifically, we model the problem of *merging string constants* as a graph coloring problem. Two strings cannot be merged if they can be used as selectors of distinct class members in the same reflection operation. This is denoted by making the strings be neighbor nodes in a *conflict* (a.k.a. *interference*) graph, which we then attempt to color. Any coloring of the graph yields different values for any two neighbors, i.e., conflicting string constants. Colors are then used as values in the whole-program static analysis, instead of string constants. In this way, a color can designate *any* of a finite set of merged strings.

The graph coloring approach has several nuances, both theoretical and experimental. First, our setting suggests the need for a very fast (certainly polynomial time) coloring algorithm, which can, however, tolerate suboptimal coloring results: The optimal coloring

still yields numbers of colors up to several hundreds, due to the existence of large cliques in the interference graph. (The cliques are due to conflicts between all members of a class, including all members declared in supertypes, all the way up to `java.lang.Object`. A deep class hierarchy can easily result in string constants being able to select several hundreds of members from the same class object.) We present a fast, near-linear-time, coloring algorithm that performs very well in this setting. As a second consideration, string merging can theoretically introduce some imprecision, due to spurious data flows in the static analysis. We find experimentally that no imprecision arises in practice, strongly validating the appeal of the string merging insight.

The string merging approach is orthogonal to other reflection analysis, string interning, etc. techniques commonly employed in the literature. Concretely, string merging *complements* any standard reflection analysis algorithm: the same reflection algorithm still applies, but for fewer abstract string constants. Similarly, the technique is agnostic to how compactly strings are represented at the low level.

We apply the string merging approach to the DOOP and SOLAR program analysis frameworks, which employ different static analyses for reflection [19, 30]. The technique yields a size reduction of $\sim 2x$ for points-to sets with string values, or a reduction of $\sim 1.5x$ in the total sizes of computed value sets. This translates to proportional savings for any further analyses as well as a speedup of $\sim 20\%$ for the base points-to analysis. Importantly, these improvements are orthogonal and assumption-less, shrinking the input of an analysis and transparently benefiting any analysis algorithm, setup, or analyzed program, with no pitfalls or drawbacks in other respects.

In all, our work makes the following contributions:

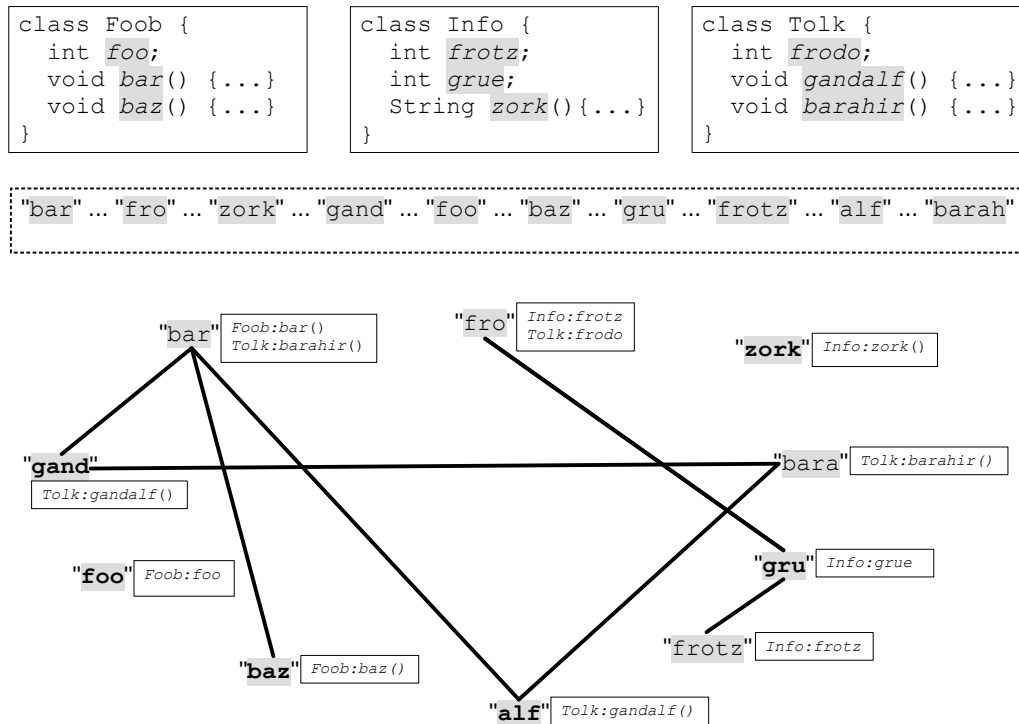
- It identifies a rather surprising problem in whole-program analysis that models reflection: string values feature disproportionately in value-flow results, such as points-to sets.
- It proposes the merging of strings that cannot differentiate members in the same class, and formulates the problem as an instance of graph coloring.
- It shows that a simple but fast graph coloring algorithm is well-suited to the specifics of the problem instance.
- It validates the approach in standard analysis frameworks and quantifies the benefit.

2 Illustration and Intuition

We illustrate (in simplified terms) the insight that our approach exploits, with the example of Figure 1.

A program contains several (10) string constants ("`bar`", "`fro`", etc.), which can be used as selectors for fields and methods of all available classes. An undirected conflict graph is produced, where two constants are neighbors iff they are substrings of the names of different fields or different methods of the same class. (In the full setting, class members also include all supertype members. However, in this example no type has a non-trivial supertype and members of the trivial superclass `java.lang.Object` are ignored for the sake of simplicity.) A single string constant has the potential to be used via reflection to select members of different classes—e.g., "`bar`" can potentially be used to select either `Foob:bar()` or `Tolk:barahir()`. Neither possibility can be precluded *a priori*, since it is the analysis of value flow itself that will determine how these strings get concatenated with others and to which reflection calls they flow.

By coloring the conflict graph (in the standard “graph coloring” sense, of different colors for neighbors), we can merge string constants together without losing *any* of their ability to be used as selectors in reflection operators. In the example, an optimal graph coloring



■ **Figure 1** Illustration of approach: sample classes (top) are accessed via reflection, from a program with 10 (sub)string constants (middle). This induces a conflict graph (bottom): two string constants conflict if they can refer to distinct members of the same type (i.e., method or field) inside the same class. The graph is 2-colorable, so just 2 values can be kept instead of the initial 10. An example coloring is shown: bold vs. non-bold strings.

uses just two colors—e.g., consider the bold and non-bold strings as having different colors. Any whole-program static analysis can then be performed with merely two artificial string constant objects, one for each color, in place of the original string constants. The first object effectively denotes either "bar" or "fro" or "bara" or "frotz", while the second stands for either "zork" or "gand" or "foo" or "baz" or "alf" or "gru". Subsequent analysis of a reflection operation will then proceed as before, e.g., knowing that either "bar" or "fro" or "bara" or "frotz" is used as an argument to a `getField()` call on an `Info` class object is as good as knowing which exact string constant was used: only field "frotz" can be selected.

In this way, the analysis needs to track the flow of a lot fewer values. The end results of a realistic whole-program analysis stop being unduly dominated by string objects.

There are several subtleties in the general approach. The idea of merging strings that cannot refer to members of the same class is simple in dynamic execution terms. In static analysis, however, there are some complicating factors that we explore in the next sections. Briefly:

- Although most published static analyses for reflection employ common base reasoning (Section 3) there are significant differences. Some algorithms track the flow of substrings (known as substring analysis), others only track strings that completely match class names or member names. Some algorithms use backward reasoning based on information other

than string matches (e.g., examining the further uses of the return value of a reflective operation). The approach is mostly orthogonal to these variations but may need slight adaptation to avoid accidental interference. Additionally, not all string constants should be merged. For instance, string constants that match class names should not be merged: they are the primary selectors in reflection logic. Section 5 discusses such topics in more detail.

- Due to independent static analysis imprecision, string merging is not guaranteed to produce identical results as an original analysis that uses full string constants. A string constant can be spuriously computed to flow to a reflection operation over an incompatible class object. Before merging, the spurious flow could fail to find a matching member, whereas after merging it may do so. Such accidental imprecision, although uncommon, can be solved – Section 5 describes solutions as applied to two reflection analysis frameworks. In our experiments (Section 6), we have not found string merging to introduce *any* practical imprecision in static analysis: precision metrics, such as methods called by each site, remain identical.

3 Static Analysis with Reflection

The context of our work is a standard scheme that integrates Java reflection reasoning in a static analysis that tracks the flow of objects inter-procedurally. We discuss it first (Section 3.1) and then present variations (Section 3.2).

3.1 Inter-Procedural Reflection Analysis

Most static analyses for reflection [18,19,21,23,30,34] employ variations of a scheme originally due to Livshits et al. [21,23]. This standard scheme is based on computing the flow of objects in mutual recursion with the computation of the effects of reflection actions. A value-flow analysis (e.g., a standard points-to analysis, for all reference variables in a program) gives information to the reflection analysis and vice versa. This interplay of analyses is necessary because the different elements of reflective actions can be distributed throughout the program. Consider a typical pattern of reflection use:

```
1 String className = ... ;
2 Class c = Class.forName(className);
3 Object o = c.newInstance();
4 String methodName = ... ;
5 Method m = c.getMethod(methodName, ...);
6 m.invoke(o, ...);
```

All of the above statements can occur in distant program locations, across different methods, invoked through virtual calls from multiple sites, etc. Thus, a whole-program analysis with an understanding of heap objects is required to track reflection. This suggests the idea that reflection analysis can leverage points-to analysis—it is a client for points-to information. At the same time, points-to analysis needs the results of reflection analysis—e.g., to determine which method gets invoked in the last line of the above example, or what objects each of the example’s local variables point to. Thus, under the Livshits et al. approach, reflection analysis and points-to analysis become mutually recursive.

This mutual recursion is typically captured in simple logical rules in the Datalog language. Datalog is ideal in that a) its computation model is based on the recursive specification of logical relations; b) it has already been used in a large amount of static analysis research work (e.g., [4, 11, 14, 16, 20, 25, 27, 29, 32, 33]).

V is a set of variables
 H is a set of heap object abstractions
 M is a set of methods
 S is a set of method signatures (including name)
 I is a set of instructions (e.g., invocation sites)
 T is a set of class types
 \mathbb{N} is the set of natural numbers

$\text{CALL}(i: I, sig: S)$	# instruction i is a call $sig(\dots)$.
$\text{ASSIGNRETURNVALUE}(i: I, v: V)$	# instruction i is a <i>return</i> v .
$\text{ACTUALARG}(i: I, n: \mathbb{N}, v: V)$	# the n -th parameter of call instruction i is local var v .
$\text{HEAPTYPE}(h: H, t: T)$	# object h has type t .
$\text{LOOKUP}(sig: S, t: T, m: M)$	# in type t there is a method m with signature sig .
$\text{CONSTANTFORCLASS}(h: H, t: T)$	# string object h matches name of class/type t .
$\text{CONSTANTFORMETHOD}(h: H, sig: S)$	# string object h matches name of method sig .
$\text{REIFIEDCLASS}(t: T, h: H)$	# special object h represents the class object of type t .
$\text{REIFIEDOBJECT}(i: I, t: T, h: H)$	# special object h represents objects of type t allocated with a <code>newInstance</code> call at invocation site i .
$\text{REIFIEDMETHOD}(sig: S, h: H)$	# special object h represents the reflection object for method signature sig .

■ **Figure 2** Input domains and relations representing the input program.

Computation in Datalog consists of monotonic logical inferences that apply to produce more facts until fixpoint. A Datalog rule “ $C(z,x) \leftarrow A(x,y), B(y,z)$.” means that if $A(x,y)$ and $B(y,z)$ are both true, then $C(z,x)$ can be inferred.

We consider the core of the analysis algorithm on the features of the above example: creating a reflective object representing a class (a *class object*) given a name string (library method `Class.forName`), creating a new object given a class object (library method `Class.newInstance`), retrieving a reflective method object given a class object and a signature (library method `Class.getMethod`), and reflectively calling a virtual method on an object (library method `Method.invoke`). This treatment ignores several other APIs (e.g., we show method lookups but not field lookups), which are handled similarly.

The analysis takes as input the relations (i.e., tables filled with information from the program text) shown in Figure 2. Using these inputs, the Livshits et al. reflection analysis can be expressed as a four-rule addition to any points-to analysis. The rest of the points-to analysis (not shown here—see e.g., [11, 15, 32]) supplies more rules for computing a relation $\text{VARPOINTSTO}(v: V, h: H)$ and a relation $\text{CALLGRAPHEDGE}(i: I, m: M)$. Intuitively, the traditional points-to part of the joint analysis is responsible for computing how heap objects flow inter-procedurally through the program, while the added rules contribute only the reflection handling. We explain the rules below.

$\text{VARPOINTSTO}(r, h) \leftarrow$
$\text{CALL}(i, \text{"Class.forName"}), \text{ACTUALARG}(i, 0, p), \text{ASSIGNRETURNVALUE}(i, r),$
$\text{VARPOINTSTO}(p, c), \text{CONSTANTFORCLASS}(c, t), \text{REIFIEDCLASS}(t, h).$

The first rule models a `forName` call, which returns a class object given a string representing the class name. The rule says that if the first argument (0-th parameter, since `forName` is a static method) of a `forName` call points to an object that is a string constant, then the types, t , that match that constant are retrieved. Assuming the result of the `forName` call at instruction i is assigned to a local variable r , and the reflection object for a t is h , r is inferred to point to h .

$$\begin{array}{l} \text{VARPOINTSTO}(r, h) \leftarrow \\ \text{CALL}(i, \text{"Class.newInstance"}), \text{ACTUALARG}(i, 0, v), \text{VARPOINTSTO}(v, h_c), \\ \text{REIFIEDCLASS}(t, h_c), \text{ASSIGNRETVVALUE}(i, r), \text{REIFIEDOBJECT}(i, t, h). \end{array}$$

The above rule reads: if the receiver object, h_c , of a `newInstance` call is a class object for class t , and the `newInstance` call is assigned to variable r , then make r point to the special (i.e., invented) allocation site h that designates objects of type t allocated at the `newInstance` call site.

$$\begin{array}{l} \text{VARPOINTSTO}(r, h_m) \leftarrow \\ \text{CALL}(i, \text{"Class.getMethod"}), \text{ACTUALARG}(i, 0, b), \text{ACTUALARG}(i, 1, p), \\ \text{ASSIGNRETVVALUE}(i, r), \text{VARPOINTSTO}(b, h_c), \text{REIFIEDCLASS}(t, h_c), \\ \text{VARPOINTSTO}(p, c), \text{CONSTANTFORMETHOD}(c, s), \\ \text{LOOKUP}(t, s, _), \text{REIFIEDMETHOD}(s, h_m). \end{array}$$

The above rule gives semantics to `getMethod` calls. It states that if such a call is made with receiver b (for “base”) and first argument p (the string encoding the desired method’s signature), and if the analysis has already determined the objects that b and p may point to, then, assuming p points to a string constant encoding a signature, s , that exists inside the type that b points to (“_” stands for “any” value), the variable r holding the result of the `getMethod` call points to the reflective object, h_m , for this method signature.

$$\begin{array}{l} \text{CALLGRAPHEDGE}(i, m) \leftarrow \\ \text{CALL}(i, \text{"Method.invoke"}), \text{ACTUALARG}(i, 0, b), \text{ACTUALARG}(i, 1, p), \\ \text{VARPOINTSTO}(b, h_m), \text{REIFIEDMETHOD}(s, h_m), \\ \text{VARPOINTSTO}(p, h), \text{HEAPTYPE}(h, t), \text{LOOKUP}(t, s, m). \end{array}$$

Finally, all reflection information can contribute to inferring more call-graph edges. The last rule encodes that an edge can be inferred from the invocation site, i , of a reflective `invoke` call to a method m , if the receiver, b , of the `invoke` (0th parameter) points to a reflective object encoding a method signature, and the argument, p , of the `invoke` (1st parameter) points to an object, h , of a class in which the lookup of the signature produces method m .

3.2 Variations

Several enhancements and variations of this general reflection analysis scheme have been employed in past work. We summarize them below, since we will need to refer to them in later sections.

- Complex mechanisms for substring flow and matching can be used, instead of matching full class and method names. Consider the first of the previous rules, handling `Class.forName` calls. The rule uses predicate `CONSTANTFORCLASS`, which could well encode a substring match instead of a full match of the class name. The complication will then be to propagate strings through concatenation operations, so that the `VARPOINTSTO` relation (the main value-flow relation of the analysis) computes the flow of substrings throughout the program. (That is, when a string constant is in a points-to set, this signifies that the analysis computes that the run-time value is the result of concatenating some prefix and suffix to the string constant.) This tends to put more pressure on the size of the points-to set as string information is allowed to flow through more avenues.
- The base rules show a forward analysis, where string values need to be completely determined for the result of a reflective operation to be modeled. Every one of the four rules is predicated on a past `VARPOINTSTO` result that establishes that a parameter points to a certain string that matches other rule conditions. An alternative is to analyze how the results of reflective operations are used in further code—i.e., to perform a backward analysis. There are many sources of such backward analysis information [18, 19, 30]. For

a simple case [21], if the result of a `c.newInstance` call is cast to a type `T`, then `T` gives a strong hint on the value of reflective class object `c`. This is particularly useful when `c` has been produced by using external sources (e.g., strings from a file or the network) so that its value cannot be determined by normal forward analysis. The same technique can also be used to get higher precision, by cross-validating the inferences of forward and backward reflection analysis before allowing them to affect the rest of the analysis.

- The base scheme shown (see 3rd rule of previous section) uses strings as method selectors only when the analysis has determined the containing class. This may not be necessary, however: a string matching a method name may be descriptive enough to determine both the method and the containing class. Such reasoning is typically performed under further qualifications of precision. For instance, the rule may only fire if the method name matches very few methods in the whole program and/or if the method name is a constant that is close to the reflective operation (e.g., in the same method), to avoid imprecision.

4 String Merging via Coloring

Our approach consists of merging string constants that occur in the program text. As illustrated in Section 2, strings can be merged if they cannot serve to distinguish members of the same class. This agrees with the main forward logic of static reflection analysis, as seen in the 3rd rule of Section 3.1: a string denoting a member name is only used for known class objects. (Exceptions are discussed in Section 5.)

The string-merging approach operates before inter-procedural analysis is performed. Effectively, the analysis input is pre-processed so that the domain H of heap object abstractions gets shrunk: abstract objects representing string constants are merged, while all others remain unchanged. The inter-procedural reflection and value-flow (i.e., points-to) analysis then proceeds unchanged, over the optimized domain. The pre-processing also entails correctly updating all input predicates (e.g., `CONSTANTFORMETHOD`) so that a merged string value can serve to select all members that its constituents can represent.

The more interesting aspects of the technique concern how the string-merging decision is made. As we saw, the problem can be viewed as an instance of standard graph coloring, on an undirected interference/conflict graph with nodes representing all string constants in the program text. Two strings are neighbors iff they match distinct members (of the same kind, i.e., both fields or both methods) in the same class.

Optimal graph coloring is an NP-hard problem. There are, however, strong reasons why our setting is a good fit for algorithms that yield more colors (i.e., do suboptimal merging) but are very fast.

- The minimum number of colors required is large. The input graph contains large cliques of nodes, because of classes with several hundreds of members. Such classes arise due to inheritance hierarchies, since inherited members of a class need to be taken into account for reflective lookups. All strings matching members of such a class form a clique: any two of them are connected in the interference graph, so all of them need to be kept distinct. Even just considering classes in the Java system libraries, cliques of size in the hundreds arise.

Based on this observation, the best that an optimal algorithm can hope to achieve is a reduction of string constant values from the several thousands (as typical in a large Java program) to the few hundreds. Therefore, a sub-optimal coloring can still capture a lot of the benefit—even twice as many colors as the optimal number represent a substantial reduction in the number of string constants that the static analysis needs to track throughout the program code.

- The benefit from string merging is not proportional to the reduction in the number of string constants. The benefit is, instead, reflected well by the reduction of the size of the VARPOINTS_{TO} relation. For instance, if the number of string constants tracked by the analysis drops by a factor of 100 (i.e., on average 100 original string constants are mapped to each color and merged) the ensuing reduction in the complexity of the string-tracking analysis will be much more modest—typically well below 10. Benefits arise only when merged strings would have been inferred to be members of the same points-to set. However, most points-to sets in an analysis are small, containing at most a handful of members. In the worst case, if a variable were to point to a single string constant in the original analysis, no benefit would arise: the string constant would merely be replaced by a merged representative, but the points-to set would still be of size 1.
- Per the above, the benefit of optimal graph coloring is tiny in our setting. Conversely, the speed of the coloring algorithm is crucial. String merging (i.e., graph coloring) is a pre-processing step, whose running time burdens the static analysis itself.

Accordingly, we employ a near-linear-time greedy algorithm for coloring the interference graph of string constants. The coloring (or “numbering”) algorithm specification is simple:

1. Compile an undirected graph $G = \{V, E\}$ where V represents the original string constants and E represents the conflicts between string constants.
2. Apply any total ordering relation \leq defined over V to each edge E and direct each edge according to this. This produces a directed acyclic graph $G' = \{V, E'\}$.
3. For each $v \in V$, its color is established by computing the maximum distance from any root (i.e., node in V with zero in-degree) in G' .

In practice the ordering \leq of strings can be arbitrary (e.g., lexicographic, or numeric by internal index, or random id) and an efficient implementation to establish the maximum distance to a root is to examine strings in a topological order over G' . Therefore each string s in V is examined only after all its predecessors t . We give string s color (i.e., number) i , where i is one higher than the maximum color of any predecessor.

The numbering step trivially gives different numbers to conflicting string constants: for every two conflicting nodes, one will be below the other in the total ordering, so they cannot have the same maximum distance from a root node. Using a standard implementation of topological sorting with a min-heap data structure, the resulting algorithm runs in time $O(e \cdot \log(n))$, where $e = |E|$ (the number of graph edges) and $n = |V|$ (the number of string constants). (Each edge needs to be traversed upon examination of its source node, in order to update the color bound of the edge’s target node.)

In the worst case, this numbering algorithm would yield suboptimal colorings—e.g., if the greatest element of one clique, per the \leq ordering, is the least element of the next, the two cliques cannot reuse colors. However, such adversarial input is unlikely if one picks ordering \leq randomly. Even a lexicographic ordering \leq of strings yields consistently good results due to the nature of our setting: the strings are used for reflection analysis, therefore they need to match original, unobfuscated names of class members. (Reflection on obfuscated member names is not possible, since all string manipulation—e.g., concatenation or substring matching—gets invalidated.) Human-written member names are distributed fairly well lexicographically, so that different classes have members at dispersed points in the global ordering.

Figure 3 shows the above logic in Datalog with stratified aggregation (all ranges of min/max aggregators are computed before the aggregation needs to take place). We reuse relations from Section 3.1 and again only refer to methods—extending to also include field

26:10 Efficient Reflection String Analysis via Graph Coloring

DECLARINGTYPE($m: M, t: T$)	# member m is declared or inherited by type t .
LESSTHAN($s1: H, s2: H$)	# string constant $s1$ is $\sqsubset s2$.
COLORATLEAST($s: H, n: \mathbb{N}$)	# string constant s needs a color of at least n .
COLOREQ($s: H, n: \mathbb{N}$)	# string constant s will be assigned color n .
REPRESENTATIVEFORCOLOR($s: H, n: \mathbb{N}$)	# string constant s will represent color n .

LESSTHAN($string1, string2$) \leftarrow
 CONSTANTFORMETHOD($string1, m1$), DECLARINGTYPE($m1, class$),
 CONSTANTFORMETHOD($string2, m2$), DECLARINGTYPE($m2, class$),
 $m1 \neq m2, string1 < string2$.

COLORATLEAST($string, 0$) \leftarrow CONSTANTFORMETHOD($string, _$).

COLORATLEAST($string1, n + 1$) \leftarrow
 LESSTHAN($string2, string1$), COLORATLEAST($string2, n$).

COLOREQ($string, \mathbf{max}(n)$) \leftarrow COLORATLEAST($string, n$).

REPRESENTATIVEFORCOLOR($\mathbf{min}(string), n$) \leftarrow COLOREQ($string, n$).

■ **Figure 3** An extra input relation, computed relations, and Datalog rules for string coloring algorithm.

members is trivial. The final rule performs an arbitrary aggregation/choice of a representative string constant, among the ones in the input, to stand for all string constants with the same color in the reduced input domain, H . This is a simplified version of the logic used in our actual implementation.

Note that the straightforward realization of these rules in a Datalog engine will likely have worst-case quadratic complexity, instead of the $O(e \cdot \log(n))$ bound for the algorithm implemented with a heap data structure. In practice, this effect can be mitigated with standard Datalog optimization techniques. Although the LESSTHAN relation is worst-case quadratic, it is also local, since a string will only conflict with a small number of others, i.e., up to a constant. We can define a more economical intermediate relation, IMMEDIATELYLESSTHAN, based on LESSTHAN, and compute COLORATLEAST more efficiently using it.

With either a direct implementation or minimal optimization effort of Datalog rules, the running time of the algorithm for sets of string constants in realistic Java applications is virtually instantaneous, i.e., entirely negligible compared to subsequent analysis time.

5 Practical Applications of Technique

The essence of our technique, described in the previous section, is a good illustration of the principles, but it ignores several realistic semantic complications. Additional development is needed to integrate this technique to existing real world analyses without sacrificing soundness and precision.

5.1 Combining Forward with Backward Analyses to Counter Imprecision

String merging, when combined with (unavoidable) static analysis imprecision, is not guaranteed to produce identical results as an original analysis that uses full string constants. String, Method or Class constants could be spuriously computed to flow to a reflection operation producing statically inferred behavior that was not meant to happen at runtime. Before merging string constants, the spurious flow could fail to find matching spurious members, whereas after merging it may do so. Here, backward analyses [18, 19, 21, 30] come into play to correct virtually all precision issues.

```

1 String a = "zork"; // i.e. {gru, alf, baz, foo, gand, zork};
2
3 Class cls = unknown() ? Foob.getClass() : Info.getClass();
4
5 Method m = cls.getMethod(a); // m = zork or baz ?
6 String s = (String) m.invoke(); // m = zork!

```

In the example above, in the original program, variable `a` is assigned to string `"zork"` at line 1. Assuming a class structure as presented in Figure 1, our technique substitutes `"zork"` with another object that represents any of the following strings: `"gru"`, `"alf"`, `"baz"`, `"foo"`, `"gand"`, `"zork"`. At line 3, a conditional assignment with `unknown` predicate causes the static analysis to consider that `m` could either be class `Foob` or class `Info`. At line 5, in the original program we get method `zork` from the classes pointed by variable `cls`. Unfortunately, in the transformed program, the representative of `"zork"` matches both `Info.zork` and `Foob.baz`. Although some imprecision is introduced here, the analysis has means to reverse this. Since the method in `m` is invoked in the next line and the return value is cast, the analysis infers that `m` would not contain `Foob.baz` but just `Info.zork`, which is the only of the two methods that returns a `String`. (The astute reader will note that this is not a 100% sound treatment, however real world reflection analysis tools need to manage and balance precision, soundness and scalability.) In this way, a backward analysis serves the role of cross-validating forward analysis results to negate imprecision. Similarly, since `Info.zork` is only defined in class `Info`, the backward analysis also informs the forward analysis of class constants to infer that variable `cls` only contains class `Info`.

In practice, backward analyses like the ones demonstrated in this example are necessary to maintain a precise analysis whether or not our string coloring technique is applied, and indeed both state-of-the-art static reflection analysis frameworks on which our technique was applied (DOOP [4] and SOLAR [19]) enable these enhancements by default.

5.2 High Confidence Inferences

Although in practice most reflection inferences involve forward and backward analyses, this is not always the case. In DOOP, string constants that originate locally and flow to a reflective operation sink locally are treated as *high-confidence* inferences, and thus do not require confirmation from backward analysis. For instance, we can take the following example:

```

1 String a = in.readLine(); // not statically known
2
3 Class cls = Class.forName(a);
4
5 Method m = cls.getMethod("zork"); // cls must be Info for this to work
6

```

In the original program, we are not able to statically determine the string passed to variable `a` on line 1. However, if the analysis can determine with high confidence that a

method in this unknown class matches "zork" then this inference is used to determine that `cls` points to the class `Info`. Merging "zork" with other strings interferes with this mechanism. By nature, high confidence inferences need to be carried out under strict, syntactically apparent conditions, thus limiting their applicability. These same conditions can also be picked up in the string merging pre-analysis. For instance, in DOOP, only strings that originate in the same method where a matching reflective operation is performed are used for high confidence inferences. A solution we implemented for this scenario is to perform a more *selective* string merging—if a string can flow to a local reflective operation, that string is not merged.

5.3 Selective Unsoundness

Selective unsoundness in the design of static reflection analysis can also cause challenges to our technique. For instance, typical reflection analyses exclude strings that are not meaningful enough to determine class names. A common substring (for instance "Impl") can match several classes. Using such strings to resolve class names naturally leads to imprecision in the analysis. A sensible heuristic in this case is to not perform static reflection analysis on strings that match more than some arbitrary number of classes. When strings are represented by their color, each color encodes multiple strings, matching methods or fields. (Note that a string can sometimes match both a method and a class.) Therefore the string coloring technique is at odds with the selective unsoundness heuristic: a merged string may be filtered out in other analysis reasoning. In this case, the solution we adopted was to not merge string constants if they can match classes. Alternatively, one can include strings that match classes to conflict graphs and have these all in the same clique—each color at most would represent one string that matches a class.

Design decisions and heuristics similar to these are present throughout real-world reflection analyses, which implies that any analysis optimization that may introduce imprecision could also introduce unsoundness.

6 Evaluation

We implemented our string coloring algorithm and applied our general technique to the most recent development version of the DOOP [4] static analysis framework. We have also applied our technique to the SOLAR pointer and reflection analysis framework [19]. Both DOOP and SOLAR are full-featured and handle most complex semantic aspects of the Java language, such as reflection, implicit initialization, exceptions, and more.

On both frameworks, we only needed to perform minimal modifications to their logic to apply this technique. Most of the modifications that we made are optimizations and additional indexing in the reflection logic. These simple optimizations were applied after we discovered that the additional load on the backward analyses necessitated better indexes. These modifications were applied to both the baseline configuration and the string coloring configuration, so both configurations benefited from these performance improvements. On both frameworks, we compare the performance of the analysis with and without string coloring enabled.

This evaluation intends to answer the following research questions:

RQ1 Does the presented technique enhance the efficiency of static analysis?

RQ2 Does the technique compromise the quality of the static analysis? In terms of:

RQ2.1 Soundness.

RQ2.2 Precision.

RQ3 Does our fast string coloring algorithm perform as-predicted, in terms of coloring effectiveness and its translation to string-merging effectiveness?

To answer these research questions, we employ the following metrics:

Var points-to. The size of the VARPOINTS_{TO} relation, on both application and library code. This metric strongly correlates with relevant static analysis time. This is by far the largest relation that is produced as output by the analysis, and describes what stack variables point to. The cost of any further use of analysis results is likely to be highly correlated to the size of VARPOINTS_{TO}.

Heap points-to. The cumulative size of all heap relations, i.e., instance field points-to, static field points-to and array index points-to. These form the second largest relation produced by the analysis, and describes what heap objects point-to.

Relevant static analysis time. The time required to run our static pointer analysis. This includes the time to run the graph coloring algorithm and all associated overheads when this is enabled.

Call graph size. We compare the sizes of call graphs before and after our optimization is applied. A smaller call graph indicates unsoundness, while a larger indicates imprecision and can answer RQ2.

Var points-to string. The size of the var points-to relation subset containing only strings as heap objects.

Size of largest clique. The size of the largest clique in the string conflict graph.

Number of colors. The total number of colors applied to the string conflict graph.

The metrics are established through the use of existing tools, applied to benchmarks from the DaCapo 9.12-Bach Java benchmark suite [2] in the case of the DOOP framework. In the case of the SOLAR framework, we use the same subset of the DaCapo 2006 benchmarks used in the original SOLAR evaluation [19]. Both static analysis frameworks use the PA-Datalog engine, a publicly available, stripped-down version of the commercial LogicBlox Datalog engine. Both frameworks are run with full-featured static handling of reflection. All our run times are established on an idle machine with an Intel Xeon E5-2687W v4 3.00GHz with up to 512 GB of RAM. All experiments had a cutoff time of 6 hours. (This is merely a practical time-budgeting limit. We have occasionally let several instances of heavy analyses run for longer but have not observed analyses that terminate in under 10 hours if they do not terminate in 6.) Timings reported are from a single run, but repeat runs show very low variation (up to about 5%, typically much lower).

Notably, all analyses operate on an already economical representation of string constants. Strings are interned and represented in all relations via a 22-bit identifier. (This means that the maximum string pool size is $2^{22} = 4194304$, which is still three orders of magnitude larger than the number of string constants arising in our benchmarks, as we shall see in Figure 10.) Furthermore, all experiments are conducted with all other standard string merging approaches enabled: all strings that cannot be used for reflection (i.e., that do not match class, method, or field names) are merged into a single, nondescript abstract string.

The DOOP framework is flexible with respect to context sensitivity, so we configure it with several flavors:

insens: No context sensitivity.

1call: A 1-call-site sensitive analysis (heap insensitive). This is also known as 1-CFA [28].

2type+H: A 2-type-sensitive analysis [31] with a 1-type-sensitive heap.

2obj+H: A 2-object sensitive analysis with 1-object sensitive heap.

The SOLAR framework is configured to use a selective 2-type-sensitive+heap. This adds call-site sensitivity for static methods to a 2-type-sensitive analysis. This is the kind of context sensitivity that SOLAR is tuned for [19].

6.1 RQ1: Performance and Efficiency Gains

As shown in Figure 4, the technique achieves an average analysis speedup of about 20% on the DOOP framework. The running time includes all overheads (including pre-processing of the input) and shows benefits throughout all analysis configurations. This captures well the overall deployment mode of the string-merging optimization: the benefit is orthogonal to any other optimizations or analysis options, consistently applicable, and without adding potential downsides.

Running time reduction of the main analysis is only one aspect of the benefit, however. Memory is often a bottleneck for analyzing applications. Figure 5 demonstrates the reduction in memory footprint of the whole analysis database. Our approach shows a larger benefit for this metric, especially with larger analyses, such as the 2-object-sensitive+heap analysis (as there are fewer constant overheads).

Since points-to analysis is mainly used as a general substrate by higher-level analysis clients, the benefit to these is the overall size reduction of the data they import: the points-to sets for local variables (var points-to, Figure 6) and for heap object references (heap points-to, Figure 7). The sizes of these relations typically drop by factors of 1.5x or higher, across all benchmarks and different context sensitivities.

Similar results can be seen for the SOLAR reflection analysis framework, in Figure 8. Notice that, overall, the technique yields slightly less benefit for SOLAR. This is mostly attributed to the fact that SOLAR does not perform substring analysis—only strings fully matching member names are tracked by the analysis. Core reflection analysis coverage improvements such as substring analysis increase the size of the points-to set substantially since strings are allowed to flow in and out through string factory operations such as `StringBuilder.append` and `StringBuilder.toString` respectively.

6.2 RQ2: Precision and Soundness

Throughout our evaluation, the string coloring technique compromises neither precision nor soundness, since either of these conditions hold in both implementations:

- Forward and backward analyses in reflection must agree with each other—this drastically improves precision in reflection analysis, whether or not our technique is applied.
- High-confidence inferences (not requiring both forward and backward analyses) are limited by some condition that allows a preprocessing step to select which strings should be merged and which should not.

Experimentally, we have verified that both precision and soundness are preserved when our technique is enabled. One (of the many) metrics we employ to quantify precision and soundness is the number of call-graph edges (projected context-insensitively). A smaller call-graph is due to unsoundness, while a larger one is due to imprecision. Call-graph edge numbers are shown in Figure 9, and, as we can see, remain virtually identical.

6.3 RQ3: Effectiveness of Coloring Algorithm and String Merging

The graph coloring algorithm of Section 4 is very inexpensive. Figure 10 reports running times for the DaCapo Bach benchmarks, at less than a second to run on average. These

		insens	1call	2type+H	2obj+H
avroa	original	427	919	443	599
	coloring	394	829	405	512
	speedup	8%	11%	9%	17%
batik	original	331	626	1076	1657
	coloring	289	531	879	1376
	speedup	15%	18%	22%	20%
eclipse	original	321	599	492	926
	coloring	306	503	445	741
	speedup	5%	19%	11%	25%
h2	original	386	753	2621	14535
	coloring	317	470	2128	13694
	speedup	22%	60%	23%	6%
jython	original	17305	-	-	-
	coloring	15659	-	-	-
	speedup	11%	-%	-%	-%
luindex	original	166	221	148	213
	coloring	100	201	138	186
	speedup	66%	10%	7%	15%
lusearch	original	152	188	150	210
	coloring	135	167	140	179
	speedup	13%	13%	7%	17%
pmd	original	240	344	271	469
	coloring	175	305	250	433
	speedup	37%	13%	8%	8%
sunflow	original	328	498	293	402
	coloring	264	466	267	334
	speedup	24%	7%	10%	20%
xalan	original	385	883	817	1877
	coloring	351	627	666	1482
	speedup	10%	41%	23%	27%
average speedup		21%	21%	13%	17%

■ **Figure 4** Points-to analysis time in seconds, including overheads of graph coloring. Empty values indicate the analysis did not terminate within six hours.

numbers likely include several extra overheads (since they are inside a Datalog engine, where reasoning is performed as database table joins) but are still entirely negligible compared to the subsequent static analysis.

We also need to evaluate experimentally how effective the algorithm is, in terms of the number of colors it produces. Figure 10 shows this number of colors, also giving the size of the largest clique in the string-conflict graph (which is a lower bound even for optimal coloring) and the total number of string constants, i.e., nodes in the graph. The algorithm achieves significant reduction factors (mean 6.5x, i.e., 6.5 strings on average are merged into one) leaving little benefit for an algorithm that achieves tighter coloring. The largest clique for most benchmarks is the same (size 176), which is an artifact of the way the DaCapo benchmarks are packaged, with common libraries and a common harness among many benchmarks.

Contrasting Figure 10 with the earlier Figures 6-7 illustrates the numbers involved in our setting: relatively few string constants (in the thousands) result in tens of millions of extra points-to facts for the analysis, since they propagate to several points-to sets each.

		insens	1call	2type+H	2obj+H
avrora	original	2419	4639	2401	2923
	coloring	2076	3905	1940	2243
	ratio	1.17x	1.19x	1.24x	1.30x
batik	original	2798	4851	6622	8361
	coloring	2336	3714	5195	6805
	ratio	1.20x	1.31x	1.27x	1.23x
eclipse	original	3006	4930	3540	6090
	coloring	2646	4128	3124	4645
	ratio	1.14x	1.19x	1.13x	1.31x
h2	original	3100	7309	14722	46377
	coloring	2260	3711	10207	34277
	ratio	1.37x	1.97x	1.44x	1.35x
jython	original	42659	-	-	-
	coloring	37286	-	-	-
	ratio	1.14x	-	-	-
luindex	original	941	1502	979	1362
	coloring	800	1227	861	1080
	ratio	1.18x	1.22x	1.14x	1.26x
lusearch	original	941	1505	974	1354
	coloring	808	1254	862	1105
	ratio	1.16x	1.20x	1.13x	1.23x
pmd	original	1772	2877	1928	2903
	coloring	1500	2239	1701	2408
	ratio	1.18x	1.28x	1.13x	1.21x
sunflow	original	1813	2949	1919	2145
	coloring	1568	2494	1690	1879
	ratio	1.16x	1.18x	1.14x	1.14x
xalan	original	3751	7920	6730	12245
	coloring	2788	5004	4894	8996
	ratio	1.35x	1.58x	1.38x	1.36x
average ratio		1.20x	1.35x	1.22x	1.27x

■ **Figure 5** Memory footprint (in KB). Empty values indicate the analysis did not terminate within six hours.

		insens	1call	2type+H	2obj+H
avrora	original	23256	106429	24257	31118
	coloring	17587	88507	15355	18950
	ratio	1.32x	1.20x	1.58x	1.64x
batik	original	23035	83980	85499	106435
	coloring	15480	60202	59352	80094
	ratio	1.49x	1.39x	1.44x	1.33x
eclipse	original	18178	68590	29365	66516
	coloring	13258	52717	21421	42516
	ratio	1.37x	1.30x	1.37x	1.56x
h2	original	31745	129704	211418	576222
	coloring	18549	56162	129666	397483
	ratio	1.71x	2.31x	1.63x	1.45x
jython	original	514245	-	-	-
	coloring	481310	-	-	-
	ratio	1.07x	-	-	-
luindex	original	7322	21934	7524	13800
	coloring	4802	16047	4844	8486
	ratio	1.52x	1.37x	1.55x	1.63x
lusearch	original	7490	21733	7558	13673
	coloring	4938	16109	5018	8947
	ratio	1.52x	1.35x	1.51x	1.53x
pmd	original	11638	42510	14489	30456
	coloring	7118	28513	9963	21297
	ratio	1.64x	1.49x	1.45x	1.43x
sunflow	original	15209	52802	16475	19202
	coloring	10708	41000	11000	13623
	ratio	1.42x	1.29x	1.50x	1.41x
xalan	original	32664	132560	94050	180099
	coloring	18169	74031	59405	125325
	ratio	1.80x	1.79x	1.58x	1.44x
average ratio		1.49x	1.50x	1.51x	1.49x

■ **Figure 6** Var points-to size (in thousands). Empty values indicate the analysis did not terminate within six hours.

As discussed in Section 4, merging string constants does not translate into proportional shrinking of string points-to sets, because many original points-to sets would not contain multiple merged strings. (The median points-to set size is 1 for many analysis settings, which allows no shrinking in most cases!) To quantify the actual reduction in string-flow inferences, we show in Figure 11 the change (for the DOOP framework) in the size of points-to sets containing strings, i.e., the number of total tuples in the VARPOINTS_TO relation where the target object is a string. (This includes points-to inferences where the string object is not a constant but a completely unknown run-time value. However, the majority of the tuples concern variables that points to constant strings. This does not necessarily mean that the variable is inferred to have a constant string value, just that the constant string is a *substring* of the run-time value.)

As Figure 11 shows, string merging significantly shrinks the number of point-to inferences for strings, by roughly a factor of 2, thus capturing the majority of the potential benefit. Again, the reduction is mostly consistent throughout all the benchmarks analyzed under different context sensitivities. Interestingly, the overall reductions in points-to set sizes comes

		insens	1call	2type+H	2obj+H
avroa	original	3756	2041	864	657
	coloring	2455	1373	562	474
	ratio	1.53x	1.49x	1.54x	1.39x
batik	original	3025	1921	2956	2459
	coloring	1641	1119	2029	1901
	ratio	1.84x	1.72x	1.46x	1.29x
eclipse	original	2894	1789	1329	1408
	coloring	1892	1246	974	996
	ratio	1.53x	1.44x	1.36x	1.41x
h2	original	5252	2722	2965	5425
	coloring	2320	915	1881	3867
	ratio	2.26x	2.97x	1.58x	1.40x
jython	original	45002	-	-	-
	coloring	38388	-	-	-
	ratio	1.17x	-	-	-
luindex	original	924	569	309	343
	coloring	528	382	201	252
	ratio	1.75x	1.49x	1.54x	1.36x
lusearch	original	952	569	309	341
	coloring	550	388	207	261
	ratio	1.73x	1.47x	1.49x	1.31x
pmd	original	1562	1034	876	930
	coloring	825	595	695	778
	ratio	1.89x	1.74x	1.26x	1.20x
sunflow	original	2210	1073	607	433
	coloring	1339	725	408	350
	ratio	1.65x	1.48x	1.49x	1.24x
xalan	original	6002	4096	5857	4168
	coloring	2980	2006	3652	3173
	ratio	2.01x	2.04x	1.60x	1.31x
average ratio		1.74x	1.76x	1.48x	1.32x

■ **Figure 7** Heap points-to (in thousands). Empty values indicate the analysis did not terminate within six hours.

about due to different reasons in context-insensitive versus highly context-sensitive settings. In a context-insensitive setting, strings flow less precisely, so there are more opportunities for merged strings to appear in the same variables. In a highly context-sensitive setting, strings also make up some of the context components, and so we also see fewer references for other object types in the points-to set. We also see that strings are flowing with more precision so there is slightly less gain due to merging different strings in the same variables. Overall, these factors seem to balance themselves out and we see a consistent reduction of the points-to set for all levels of context sensitivity.

7 Related Work

There are two directions of related work: specialized graph coloring algorithms and static analyses for reflection. The former are less related to our approach, for a variety of reasons. First, our graphs have no recognized special properties. Second, most specialized graph coloring algorithms are still trying for (near-)optimal coloring, since they apply to a setting

		relevant analysis time (s)	speedup	var points-to (000')	reduction
antlr	original	409		25406	
	coloring	348	18%	22890	10%
chart	original	2498		187903	
	coloring	2195	14%	160398	15%
eclipse	original	683		65905	
	coloring	599	14%	55204	16%
fop	original	2330		150955	
	coloring	2181	7%	133686	11%
pmd	original	1064		70871	
	coloring	916	16%	50570	29%

■ **Figure 8** Performance improvements of our technique on the SOLAR analysis framework, demonstrated on the subset of the DaCapo 2006 benchmarks used in previous SOLAR work.

where tight coloring yields benefits. We have the luxury of a setting where some additional number of colors makes hardly any difference in the overall benefit. Therefore, coloring simplicity and efficiency become paramount.

Some of the best-known graph coloring results with applications in programming languages can be found in the register allocation literature. Gupta et al. [12] give a fast coloring algorithm based on clique separators, i.e., cliques whose removal would disconnect the graph. In a relatively recent and prominent representative of specialized graph coloring approaches, Hack and Goos [13] give an optimal algorithm for register allocation of SSA-form programs, by showing that such programs have *chordal* interference graphs.

Fully analyzing reflection has been attracting attention for a long time. Multiple approaches have been proposed in the past in an effort to tackle the efficiency, soundness, and precision concerns of such an analysis.

As discussed in Section 3.1, Livshits et al. introduced the idea that reflection analysis and pointer analysis have to work together in order to be effective [21, 23]. They also identify points in a given program where user input affects the resolution of reflective targets and subsequently give the user the option to provide appropriate specifications for the aforementioned points. Additionally, they provide an automated, more conservative and sometimes less precise approach in which type casts applied to the results of reflective allocations are used in order to infer the possible values of said allocated objects.

Other work [18, 24, 30] builds on the latter concept and introduces more sophisticated backward or use-based analyses in the context of Javascript and Java respectively. When objects are retrieved from unknown code (including through reflection), the analysis tries to infer the object’s properties based on the way that it is used in the code text (generalizing to more language constructs other than type casts, e.g., string literals used in a `Class.getField` invocation).

Backward and forward analysis techniques are combined in great variation. For instance, Smaragdakis et al. [30] generalize the backward-information pattern by allowing for inference to arbitrarily cross method boundaries. This backward propagation technique might have adverse effects on precision under certain conditions. To that end, the authors also introduced a forward propagation approach in which type casts on unknown reflection object are used to invent a new object of the correct type at that point that will flow normally in subsequent code. This is a converse compromise since it will not affect the properties of the unknown reflection object.

		insens	1call	2type+H	2obj+H
avrora	original	115445	109226	97504	96618
	coloring	115295	109076	97519	96633
	difference	< 1%	< 1%	< 0.1%	< 0.1%
batik	original	135479	128730	121031	120086
	coloring	135479	128730	121031	120086
	difference	nil	nil	nil	nil
eclipse	original	94375	88553	76758	76212
	coloring	94375	88553	76758	76212
	difference	nil	nil	nil	nil
h2	original	126378	115731	109814	108127
	coloring	126378	115731	109814	108127
	difference	nil	nil	nil	nil
jython	original	6278831	-	-	-
	coloring	6276704	-	-	-
	difference	< 0.1%	nil	nil	nil
luindex	original	64580	60435	55802	55808
	coloring	64580	60435	55802	55808
	difference	nil	nil	nil	nil
lusearch	original	64748	60391	55551	55572
	coloring	64748	60391	55551	55572
	difference	nil	nil	nil	nil
pmd	original	74154	69904	63704	63100
	coloring	74154	69904	63704	63100
	difference	nil	nil	nil	nil
sunflow	original	100394	94964	84609	84062
	coloring	100274	94844	84610	84063
	difference	< 1%	< 1%	< 0.1%	< 0.1%
xalan	original	119042	109191	99748	98263
	coloring	119042	109191	99748	98263
	difference	nil	nil	nil	nil

■ **Figure 9** Number of call-graph edges. Empty values indicate the analysis did not terminate within six hours.

Li et al. [19] developed SOLAR in which they apply three design novelties. Firstly, they use a lazy heap modeling on reflective allocation sites. Secondly, they introduce a collective inference for related reflective calls. Finally, they have in place an automatic identification of problematic reflective calls that potentially could threaten their analysis in terms of soundness, precision and scalability.

Techniques have also been proposed in order to tackle the scalability issues of a full fledged string analysis that is usually part of a reflection analysis. The most common practice is to merge string literals found in the program text into a single object (e.g., `SMUSH_STRINGS` in WALA [7]). The exception to this are string constants that are a possible match with class, method or field names and so could potentially appear in a reflective call. Those literals need to be analyzed with the normal precision of the analysis at hand.

Aydin et al. [1] and Bultan [5] introduced sophisticated string analyses in the context of web applications. The former developed a constraint solver that, given a string constraint, constructs an automaton that accepts all solutions that satisfy the constraint. The latter approach extracts client- and server-side input validation and sanitization functions and models them as deterministic finite automata (DFA) using symbolic fixpoint computations

	Coloring Time (s)	Colors	Largest Clique	String Constants	Compression Ratio
avrora	0.4	228	176	1768	7.8x
batik	0.6	249	176	2140	8.6x
eclipse	0.8	404	196	1836	4.5x
h2	1.4	348	176	2151	6.2x
jython	0.5	279	183	2456	8.8x
luindex	0.1	191	176	711	3.7x
lusearch	0.1	191	176	704	3.7x
pmd	0.4	232	176	1852	8.0x
sunflow	1.9	230	176	1636	7.1x
xalan	0.5	424	260	2724	6.4x
Mean	0.7				6.5x

■ **Figure 10** Various performance metrics of coloring algorithm.

with the aim of identifying errors in input validation and sanitization code. A different advanced technique for string analysis has been presented by Christensen et al. [6]. They analyze complex string expressions and abstract them via a context-free grammar that is then widened to a regular language. Reflection is one of their examples but they only apply it to small benchmarks. In the context of analyzing Android applications, DROIDSAFE [8] employs the JSA String Analyzer [6]. JSA is a flow-sensitive and context-insensitive static analysis that includes a model of common operations on Java’s `String` type. For a given string reference, the analysis computes a multi-level automaton representing all possible string values. DROIDSAFE uses JSA as a first pass (only on application code) to resolve values for string references that are arguments to the Android API. It, subsequently, converts each resolved automaton to a regular expression that represents the possible values of a string value. Generally, more precise string analyses are better suited for sensitive semantic domains and more localized application, rather than whole-program reflection analysis and arbitrary substring flow over the heap and call stack.

Traditionally, an alternative approach of handling reflection in static analysis has been the integration of user input or dynamic information along the facts inferred in a static way. A state-of-the-art example in that direction is the Tamiflex tool [3] which observes the reflective calls in an actual execution of the program and rewrites the original code to a version without reflection calls. Another hybrid dynamic-static technique is presented by Grech et al. [9] in HeapDL. The tool gathers dynamic information in the form of heap snapshots taken during program execution. Subsequently, such dynamic information is supplied to a static analysis to enhance its capabilities, substantially counteracting unsoundness with minimum intrusion to the analysis logic. Another application of this approach is to improve the scalability [10] of the analysis by replacing static reasoning with dynamic information. The use of these tools [9, 10] significantly increases the number of reflective string constants in the analysis environment, which makes the techniques presented in this paper even more effective. Although, hybrid static-dynamic techniques are a practical approach, it is unrealistic to expect that reflection will yield the same results in different program executions, given that such a runtime variability is the fuel of any reflective feature.

		insens	1call	2type+H	2obj+H
avrora	original	10390	36174	19470	17706
	coloring	4730	18300	10567	9099
	ratio	2.20x	1.98x	1.84x	1.95x
batik	original	13544	45280	55379	38459
	coloring	5989	21502	29231	17332
	ratio	2.26x	2.11x	1.89x	2.22x
eclipse	original	12324	45265	26222	47104
	coloring	7404	29392	18278	28001
	ratio	1.66x	1.54x	1.43x	1.68x
h2	original	19114	102602	148192	286138
	coloring	5918	29060	66440	121101
	ratio	3.23x	3.53x	2.23x	2.36x
jython	original	37436	-	-	-
	coloring	10764	-	-	-
	ratio	3.48x	-	-	-
luindex	original	5111	15461	5869	6231
	coloring	2592	9574	3190	3238
	ratio	1.97x	1.61x	1.84x	1.92x
lusearch	original	5301	15475	5863	6080
	coloring	2749	9851	3322	3465
	ratio	1.93x	1.57x	1.76x	1.75x
pmd	original	8550	29199	12067	12456
	coloring	4030	15202	7540	6314
	ratio	2.12x	1.92x	1.60x	1.97x
sunflow	original	8992	28021	13592	9283
	coloring	4495	16241	8118	5681
	ratio	2.00x	1.73x	1.67x	1.63x
xalan	original	25998	105773	84695	113149
	coloring	11503	47244	50050	68774
	ratio	2.26x	2.24x	1.69x	1.65x
average ratio		2.31x	2.02x	1.77x	1.90x

■ **Figure 11** String var points-to (in thousands). Empty values indicate the analysis did not terminate within six hours.

8 Conclusion and Future Work

Reflection analysis has become a mainstream feature of modern whole-program analysis tools. Since applications and libraries in the Java ecosystem use reflection for generality and configurability, it is necessary to use reflection analysis to get a good level of program analysis coverage. On the other hand, reflection analysis is also responsible for the main performance bottleneck in whole-program analysis tools. It needs to statically track string constants that can refer to class members, which tend to dominate analyses' points-to sets. There are not many statically-detectable hints within the semantics of the language to limit string flow.

The approach presented in this paper improves the performance of a static analysis by maximally encoding reflection string constants using graph coloring. Our technique compiles an interference graph of strings, and colors this graph using a fast, almost linear-time algorithm as a simple preprocessing step to encode string constants prior to static analysis. We find that string merging using graph coloring is an uncompromising technique

for addressing some of the inefficiency of static analyses, for all kinds of context-sensitive and context-insensitive analyses and across multiple reflection analysis approaches.

With some adaptations, our technique can lend itself to other similar applications. For instance the technique could apply to the analysis of dynamically typed languages (where many more operations are encoded with reflection-like functionality) or to more specific domains such as the tracking of string constants matching intents in Android applications.

References

- 1 Abdalbaki Aydin, Lucas Bang, and Tevfik Bultan. Automata-based model counting for string constraints. In Daniel Kroening and Corina S. Păsăreanu, editors, *Computer Aided Verification: 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, pages 255–272, Cham, 2015. Springer International Publishing. doi:10.1007/978-3-319-21690-4_15.
- 2 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 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006, Portland, Oregon, USA*, OOPSLA '06, pages 169–190, New York, NY, USA, 2006. ACM. doi:10.1145/1167473.1167488.
- 3 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, ICSE 2011, Waikiki, Honolulu, HI, USA, May 21-28, 2011*, ICSE '11, pages 241–250, New York, NY, USA, 2011. ACM. doi:10.1145/1985793.1985827.
- 4 Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *Proc. of the 24th Annual ACM SIGPLAN Conf. on Object Oriented Programming, Systems, Languages, and Applications, OOPSLA '09*, New York, NY, USA, 2009. ACM.
- 5 Tevfik Bultan. String analysis for vulnerability detection and repair. In *Proceedings of the 22Nd International Symposium on Model Checking Software - Volume 9232*, SPIN 2015, pages 3–9, New York, NY, USA, 2015. Springer-Verlag New York, Inc. doi:10.1007/978-3-319-23404-5_1.
- 6 Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Precise analysis of string expressions. In *Static Analysis, 10th International Symposium, SAS 2003, San Diego, CA, USA, June 11-13, 2003, Proceedings*, SAS '03, pages 1–18. Springer, 2003. doi:10.1007/3-540-44898-5_1.
- 7 Stephen J. Fink et al. WALA UserGuide: PointerAnalysis. http://wala.sourceforge.net/wiki/index.php/UserGuide:PointerAnalysis#Contexts_for_Reflection, 2013.
- 8 Michael I. Gordon, Deokhwan Kim, Jeff H. Perkins, Limei Gilham, Nguyen Nguyen, and Martin C. Rinard. Information flow analysis of android applications in droidsafe. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*. The Internet Society, 2015. URL: <http://www.internetsociety.org/doc/information-flow-analysis-android-applications-droidsafe>.

- 9 Neville Grech, George Fourtounis, Adrian Francalanza, and Yannis Smaragdakis. Heaps don't lie: Countering unsoundness with heap snapshots. *Proc. ACM Program. Lang.*, 1:1–27, 2017. doi:10.1145/3133892.
- 10 Neville Grech, George Fourtounis, Adrian Francalanza, and Yannis Smaragdakis. Shooting from the heap: Ultra-scalable static analysis with heap snapshots. In *International Symposium on Software Testing and Analysis (ISSTA)*, ISSTA '18, New York, NY, USA, 2018. ACM. doi:10.1145/3213846.3213860.
- 11 Salvatore Guarnieri and Benjamin Livshits. GateKeeper: mostly static enforcement of security and reliability policies for Javascript code. In *Proc. of the 18th USENIX Security Symposium, SSYM' 09*, pages 151–168, Berkeley, CA, USA, 2009. USENIX Association. URL: <http://dl.acm.org/citation.cfm?id=1855768.1855778>.
- 12 R. Gupta, M. L. Soffa, and T. Steele. Register allocation via clique separators. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation, PLDI '89*, pages 264–274, New York, NY, USA, 1989. ACM. doi:10.1145/73141.74842.
- 13 Sebastian Hack and Gerhard Goos. Optimal register allocation for ssa-form programs in polynomial time. *Inf. Process. Lett.*, 98(4):150–155, may 2006. doi:10.1016/j.ipl.2006.01.008.
- 14 George Kastrinis and Yannis Smaragdakis. Efficient and effective handling of exceptions in Java points-to analysis. In *Compiler Construction - 22nd International Conference, CC 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, CC '13, pages 41–60. Springer, 2013. doi:10.1007/978-3-642-37051-9_3.
- 15 George Kastrinis and Yannis Smaragdakis. Hybrid context-sensitivity for points-to analysis. In *Proc. of the 2013 ACM SIGPLAN Conf. on Programming Language Design and Implementation, PLDI '13*, New York, NY, USA, 2013. ACM.
- 16 Monica S. Lam, John Whaley, V. Benjamin Livshits, Michael C. Martin, Dzintars Avots, Michael Carbin, and Christopher Unkel. Context-sensitive program analysis as database queries. In *Proceedings of the Twenty-fourth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 13-15, 2005, Baltimore, Maryland, USA, PODS '05*, pages 1–12, New York, NY, USA, 2005. ACM. URL: <http://dl.acm.org/citation.cfm?id=1065167>, doi:10.1145/1065167.1065169.
- 17 Davy Landman, Alexander Serebrenik, and Jurgen J. Vinju. Challenges for static analysis of Java reflection – literature review and empirical study. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, 2017.
- 18 Yue Li, Tian Tan, Yulei Sui, and Jingling Xue. Self-inferencing reflection resolution for Java. In *Proc. of the 28th European Conf. on Object-Oriented Programming, ECOOP '14*, pages 27–53. Springer, 2014. doi:10.1007/978-3-662-44202-9.
- 19 Yue Li, Tian Tan, and Jingling Xue. Effective soundness-guided reflection analysis. In Sandrine Blazy and Thomas Jensen, editors, *Static Analysis - 22nd International Symposium, SAS 2015, Saint-Malo, France, September 9-11, 2015, Proceedings*, SAS '15, pages 162–180. Springer, 2015. doi:10.1007/978-3-662-48288-9_10.
- 20 Percy Liang and Mayur Naik. Scaling abstraction refinement via pruning. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, PLDI '11, pages 590–601, New York, NY, USA, 2011. ACM. doi:10.1145/1993498.1993567.
- 21 Benjamin Livshits. *Improving Software Security with Precise Static and Runtime Analysis*. PhD thesis, Stanford University, December 2006.

- 22 Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. In defense of soundness: A manifesto. *Commun. ACM*, 58(2):44–46, jan 2015. doi:10.1145/2644805.
- 23 Benjamin Livshits, John Whaley, and Monica S. Lam. Reflection analysis for Java. In *Proc. of the 3rd Asian Symp. on Programming Languages and Systems*, pages 139–160. Springer, 2005. doi:10.1007/11575467_11.
- 24 Magnus Madsen, Benjamin Livshits, and Michael Fanning. Practical static analysis of JavaScript applications in the presence of frameworks and libraries. 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, FSE '13*, pages 499–509. ACM, 2013. URL: <http://dl.acm.org/citation.cfm?id=2491411>, doi:10.1145/2491411.2491417.
- 25 Mayur Naik, Alex Aiken, and John Whaley. Effective static race detection for java. In *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, Ottawa, Ontario, Canada, June 11-14, 2006, PLDI '06*, pages 308–319, New York, NY, USA, 2006. ACM. doi:10.1145/1133981.1134018.
- 26 Oracle. Proxy (Java Platform SE 8), 2016. URL: <http://docs.oracle.com/javase/8/docs/api/java/lang/reflect/Proxy.html>.
- 27 Thomas W. Reps. Demand interprocedural program analysis using logic databases. In R. Ramakrishnan, editor, *Applications of Logic Databases*, pages 163–196. Kluwer Academic Publishers, 1994.
- 28 Olin Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie Mellon University, may 1991.
- 29 Yannis Smaragdakis and George Balatsouras. Pointer analysis. *Foundations and Trends® in Programming Languages*, 2(1):1–69, 2015. doi:10.1561/25000000014.
- 30 Yannis Smaragdakis, George Balatsouras, George Kastrinis, and Martin Bravenboer. More sound static handling of Java reflection. In *Proc. of the Asian Symp. on Programming Languages and Systems, APLAS '15*. Springer, 2015.
- 31 Yannis Smaragdakis, Martin Bravenboer, and Ondřej Lhoták. Pick your contexts well: Understanding object-sensitivity. In *Proc. of the 38th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, POPL '11*, pages 17–30, New York, NY, USA, 2011. ACM.
- 32 John Whaley, Dzintars Avots, Michael Carbin, and Monica S. Lam. Using Datalog with binary decision diagrams for program analysis. In *Proc. of the 3rd Asian Symp. on Programming Languages and Systems*, pages 97–118. Springer, 2005. doi:10.1007/11575467_8.
- 33 John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation 2004, Washington, DC, USA, June 9-11, 2004, PLDI '04*, pages 131–144, New York, NY, USA, 2004. ACM. doi:10.1145/996841.996859.
- 34 Yifei Zhang, Tian Tan, Yue Li, and Jingling Xue. Ripple: Reflection analysis for android apps in incomplete information environments. In Gail-Joon Ahn, Alexander Pretschner, and Gabriel Ghinita, editors, *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy, CODASPY 2017, Scottsdale, AZ, USA, March 22-24, 2017*, pages 281–288. ACM, 2017. URL: <http://dl.acm.org/citation.cfm?id=3029806>, doi:10.1145/3029806.3029814.

