33rd European Conference on Object-Oriented Programming

ECOOP 2019, July 15–19, 2019, London, United Kingdom

Edited by

Alastair F. Donaldson

LIPICS

Editors

Alastair F. Donaldson
Department of Computing,
Imperial College London, UK
alastair.donaldson@imperial.ac.uk

ACM Classification 2012
Software and its engineering

ISBN 978-3-95977-111-5

Published online and open access by
Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at https://www.dagstuhl.de/dagpub/978-3-95977-111-5.

Publication date
July, 2019

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

Digital Object Identifier: 10.4230/LIPIcs.ECOOP.2019.0

LIPIcs – Leibniz International Proceedings in Informatics

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

ISSN 1868-8969

https://www.dagstuhl.de/lipics

Contents

Regular Papers

33rd European Conference on Object-Oriented Programming (ECOOP 2019).
Editor: Alastair F. Donaldson
LIPICS Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Tool Insights Papers

Experience Reports

Pearls

Brave New Ideas

Message from the Chairs

We are delighted to welcome you to London for the 33rd European Conference on Object-Oriented Programming (ECOOP 2019), to be held during July 15–19. ECOOP is the European forum for bringing together researchers, practitioners, and students to share their ideas and experiences on all topics related to programming languages, software development, object-oriented technologies, systems and applications.

This year, ECOOP is once again co-located with the CurryOn conference, which is focussed on the intersection of emerging languages and industrial challenges associated with programming languages. As well as technical papers and keynotes, ECOOP 2019 features several workshops, a doctoral symposium, a poster session, and a summer school.

Awards and keynotes

ECOOP usually features keynotes from the winners of the Senior and Junior Dahl-Nygaard Prize winners. We are deeply saddened that the AITO Dahl-Nygaard Senior Prize winner, Laurie Hendren (McGill University) died in May 2019. Laurie was a leading light in the Programming Languages field, and her passing is a terrible loss to our community.

Winner of the 2019 AITO Dahl-Nygaard Junior Prize, Ilya Sergey (Yale-NUS College and National University of Singapore) will present a keynote, and we are privileged to have two further keynotes, from Azadeh Farzan (University of Toronto), and Simon Peyton Jones (Microsoft Research). Our congratulations go to Yossi Gil and Ori Roth (Technion), whose paper "Fling—A Fluent API Generator" was selected to receive an AITO Distinguished Paper Award.

Paper selection process

Authors had two main routes open to them when submitting to ECOOP 2019. There was a "Journal First" route, whereby authors could submit their papers to be considered for a special issue of the *Science of Computer Programming* journal, presenting the associated paper at ECOOP if accepted. There was also a standard route, whereby authors could submit their papers directly to the conference to be considered for presentation and inclusion in these Dagstuhl LIPIcs conference proceedings. In addition, and new for ECOOP 2019, authors could submit papers to the conference in six distinct categories:

- **Research Papers.** This was the most traditional paper category, for research papers demonstrating advances in the Programming Languages (PL) field.
- **Tool Insights Papers.** This category aimed to solicit articles focussing on the practical details of the design and implementation of PL tools—details that are often omitted from regular papers despite being fascinating and worthy of communication.
- **Reproduction Studies.** Independently reconstructing prior experiments, to validate or refute important results of earlier work, can be extremely valuable; this category welcomed papers reporting on such studies.
- **Experience Reports.** This category was for papers focussing on noteworthy applications of existing PL techniques, tools and ideas in interesting domains, potentially in the context of other communities.
- **Pearls.** Originating in the Journal of Functional Programming, and common to conferences such as ICFP and POPL, ECOOP 2019 welcomed so-called "pearl" articles that explain a known idea in a new and elegant way, to the benefit of the PL community.

33rd European Conference on Object-Oriented Programming (ECOOP 2019).
Editor: Alastair F. Donaldson
LIPICS Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

- **Brave New Ideas:** This final new category solicited forward-looking articles on ideas in the PL field that may take some time to fully substantiate, but for which early communication to the community is likely to be of benefit.

In total, ECOOP 2019 received 82 submissions, of which the 28 papers appearing in these proceedings were accepted (34.1% acceptance rate). The 82 total submissions comprised 60 Research Papers (19 accepted), 3 Tool Insights Papers (2 accepted), 5 Experience Reports (1 accepted), 4 Pearls (3 accepted), 5 Brave New Ideas (3 accepted), 5 Journal First Papers (none accepted, so no ECOOP 2019 special issue of *Science of Computer Programming*), and no Reproduction Studies.

The new paper categories and dual submission routes were largely successful in increasing the number of papers submitted to ECOOP compared with the 2018 edition of the conference, and we hope that next year's Program Chair will take them forward.

Each submission was evaluated by at least three members of the Program Committee (PC), External Review Committee (ERC) and selected additional reviewers. Papers for which a PC member was a co-author were reviewed exclusively by non-PC members. Authors were given a chance to respond to all reviews of their paper, except in a few cases where it was deemed necessary to solicit additional reviews for borderline papers after the author response period had closed.

The review process was double-blind until the point of review submission. On submitting a review for a paper, the identities of authors of the paper were revealed to the reviewer. The identities of reviewers remained hidden from authors, except that 8 papers were accepted subject to a shepherding process. In these cases, one reviewer served as shepherd for the paper, revealing to the authors that they had reviewed the paper (but not necessarily specifying for which review they were responsible), and acting as an intermediary between the authors and the other reviewers with the aim of helping the authors improve their paper based on key suggestions from the reviewers. We were delighted that all relevant papers ultimately passed the shepherding process and are included in these proceedings.

For environmental reasons, to ease pressure on PC members with family responsibilities, and to make it easier for researchers across the world to commit to serving on the PC, ECOOP 2019 did not feature an in-person PC meeting. Instead, rigorous discussion was conducted online using the HotCRP tool, and a number of conference calls were held for specific papers where it was proving hard to reach a decision asynchronously.

Authors of accepted papers were also invited to submit artifacts, which were evaluated by a separate Artifact Evaluation Committee (AEC). As detailed further in the *Message from the Artifact Evaluation Chairs* below, the committee received 16 artifacts and accepted 14 of them.

Acknowledgements

Putting ECOOP 2019 together has been a big team effort that would not have been possible without help from a lot of people. We offer particular thanks to Annabel Satin, without whose assistance and advice ECOOP could not have happened.

We are very grateful to the authors of *all* submitted papers (whether accepted or not) for taking the time to send their work to ECOOP, and to our keynote speakers and authors of accepted papers who will present at the event. We thank the 25 PC members, 15 ERC members and 15 additional reviewers for their generally very thorough reviewing efforts. We thank our Artifact Evaluation Chairs Maria Christakis and Manuel Rigger for coordinating the evaluation process, and the Artifact Evaluation Committee for their efforts. We are grateful to many other people for contributing to various aspects of the program: our Workshop Chairs Julian Dolby and Sebastian Erdweg for putting together a comprehensive schedule of workshops; Julia Belyakova and Goran Piskachev for chairing the Doctoral Symposium; Sarah Mount for her work as Diversity Chair; James Noble and Jan Vitek for co-organizing the Summer School; Jacob Hughes and Alisa Maas for their efforts as Student Volunteer Co-Chairs; Stefan Marr for managing the ECOOP web site; Edd Barrett for his tireless efforts as Publicity Chair; Heather Miller for serving as Sponsorship Chair; our poster chair Lisa Nguyen; and our Video chair Benjamin Chung.

Michael Wagner (Dagstuhl) provided excellent support in the preparation of these proceedings, the HotCRP tool was invaluable in facilitating the review process, and the ECOOP 2019 website was powered by the conf.researchr.org service.

We gratefully acknowledge our sponsor AITO as well as our financial supporters—Google, Huawei, Facebook, JetBrains, Oracle, IBM Research, Mozilla and Uber—for their generous contributions.

Finally, we hope that if you are attending ECOOP 2019 that you have a fantastic time, that you find the presentations thought-provoking and inspiring, and that you meet lots of interesting people. Thank you for supporting the event!

Alastair F. Donaldson
ECOOP 2019 Program Chair
Imperial College London

Laurence Tratt
ECOOP 2019 General Chair
King's College London

ECOOP: Looking Forward: a Message from the AITO President

A warm welcome to all: I hope that you will enjoy London and the excellent scientific program. Thanks to the organizers, headed up by Laurie Tratt, for working hard on arranging the conference—if just half of their efforts pay off, it will be a great success. ECOOP continues to have great student volunteers that help make things run smoothly—and who get to experience the conference. Thanks to Alastair Donaldson for his dedicated work as PC Chair and to the PC members, who collaborated in assembling a fine scientific program. One of the strong features of ECOOP is the workshops held in connection with the main conference that allows intense interaction between participants. Thanks to all workshop organizers. A final thanks goes to Annabel Satin, the AITO coordinator, without her, things would be a lot more difficult.

This year's Dahl-Nygaard Senior award honours Laurie Hendren, for her continuous and significant contributions for the past 30+ years to the field of object-oriented programming languages and compilation. Sadly, Laurie passed away in May due to illness.

This year's Dahl-Nygaard Junior award goes to Ilya Sergey, who has made a number of significant contributions in the development and application of programming language techniques to various problems across the programming spectrum, covering object-oriented , functional, distributed, and concurrent programming, as well as the blockchain and smart contracts.

You are encouraged to submit a nomination for either or both awards for next year.

The world is changing and so is ECOOP. ECOOP 1998 had more than 700 attendees, many workshops, a large tutorial program, and many exhibitors. Since then many things have changed starting with the `.com` bust, which meant a reduction in participation from industry and consequently also a reduction in tutorial attendance and exhibits. The past two decades has also seen a number of more specialised conferences in the OO area focusing on specific topic, e.g., aspects, Java, programming, tools, so it is perhaps natural that some move on from ECOOP to such conferences on subtopics within OO, while ECOOP still covers new, and less established OO ideas of the future.

These trends meant that we have evolved ECOOP and that there is lower attendance, significantly reduced exhibits, and a change in tutorials from fully paid introductory tutorials to an academic program of summer school tutorials. The introduction of Curry On has been successful in maintaining the link between industry and academia.

A good workshop program, besides the strong papers in the main conference, has been one of the hallmarks of ECOOP. A high quality workshop program is important to attract strong academics who are not only trendsetters, but also active participants willing to have lively discussions on their views. And for industry to absorb new trends and, conversely, pass on best practices.

Naturally, AITO continually assess the focus and direction of each ECOOP. The AITO General Assembly meeting, which traditionally is held in connection with the main conference includes a discussion on the upcoming ECOOP conferences. We appreciate all input from ECOOP attendees, so I will conclude by encouraging you to pass on your thoughts to any AITO Member.

33rd European Conference on Object-Oriented Programming (ECOOP 2019).
Editor: Alastair F. Donaldson
LIPICS Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

We do hope that you will enjoy the conference and its associated events. We do ask that everyone maintains a respectful attitude toward everyone else including avoiding behavior that might be viewed as disrespectful or unwanted. At the previous ECOOP, we did have a report of unwanted behaviour in connection with one of the social events; please be respectful at all times – even if you have enjoyed some of the local pints. If you experience any kind of behaviour that causes you discomfort, please contact one of the organisers, or an AITO Exec Member—even if you want your concerns to be kept confidential.

That said, do not forget to enjoy the conference and have fun.

Looking forward, ECOOP 2020 will be in Berlin, we hope to see you there.

Eric Jul
AITO President
University of Oslo

Message from the Artifact Evaluation Chairs

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

This year, the committee evaluated 16 artifacts, which correspond to 57% of all accepted papers. 14 of the artifacts were accepted (a 88% acceptance rate). In total, 50% of the research papers published at ECOOP 2018 have successfully passed the AE process, indicated by an artifact-evaluation badge. This outcome is similar to the outcomes of previous ECOOP editions; in 2018, 38% of the research papers, and in 2017, 59% of the research papers were accompanied by accepted artifacts.

The AE process for 2019 was a continuation of the AE process of previous ECOOP editions. In particular, the process was still based on the artifact evaluation guidelines by Shriram Krishnamurthi, Matthias Hauswirth, Steve Blackburn, and Jan Vitek published on the Artifact Evaluation site.[1] The guidelines for artifacts that contain mechanized proofs developed by the ECOOP 2018 AEC were also reused to help both reviewers and authors in creating and reviewing such artifacts.

Each artifact was evaluated by three AEC members, which corresponded to a reviewer load of two to three artifacts. The reviewing process consisted of two phases. In the "kick-the-tires" phase, reviewers briefly verified the basic integrity of the artifacts to discover any issues that could prevent the evaluation of the artifact (e.g., a corrupted virtual machine image) and to assign a grade for the getting-started guide. In case of any issues, reviewers could, as part of a response phase, indicate issues and ask clarifying questions to the authors. Authors, in turn, could respond to the reviewers' first feedback, and update their artifacts to address any issues that were raised by the reviewers. In the second phase, each reviewer had three weeks to do a comprehensive evaluation of each artifact. Reviewers were asked to assess the consistency of the artifact with respect to the paper, the artifact's completeness, documentation, and reusability for future research and to decide on an overall grade. The review phase was then followed by a discussion phase, in which artifacts were discussed to converge on either the artifacts' acceptance or rejection. Authors that received an acceptance notification were given one week of time to incorporate reviewers' feedback and submit the camera-ready version of their artifacts.

We would like to thank the 19 members of this year's AEC, who donated their valuable time and effort to make the AE process possible. We would also like to thank Michael Wagner for the publication of the artifacts volume, and the Program Chair Alastair Donaldson for helping us coordinate the artifact evaluation with the paper review process.

Maria Christakis
ECOOP 2019 Artifact Evaluation co-chair
Max Planck Institute for Software Systems

Manuel Rigger
ECOOP 2019 Artifact Evaluation co-chair
ETH Zurich

[1] `http://www.artifact-eval.org`

ECOOP 2019 Conference Organization

General Chair
Laurence Tratt *(King's College London, UK)*

Program Chair
Alastair F. Donaldson *(Imperial College London, UK)*

Artifact Evaluation Co-Chairs
Maria Christakis *(MPI-SWS, Germany)*
Manuel Rigger *(ETH Zurich, Switzerland)*

Workshop Co-Chairs
Julian Dolby *(IBM Research, USA)*
Sebastian Erdweg *(JGU Mainz, Germany)*

Web Chair
Stefan Marr *(University of Kent, UK)*

Publicity Chair
Edd Barrett *(King's College London, UK)*

Sponsorship Chair
Heather Miller *(Carnegie Mellon University, UK)*

Diversity Chair
Sarah Mount *(Aston University, UK)*

Indispensable Organisational Memory, AITO Liaison and Finance Chair
Annabel Satin

Posters Chair
Lisa Nguyen Quang Do *(Paderborn University, Germany)*

Summer School Co-Chairs
James Noble *(Victoria University of Wellington, New Zealand)*
Jan Vitek *(Northeastern University, USA)*

Doctoral Symposium Co-Chairs
Julia Belyakova *(Northeastern University, USA)*
Goran Piskachev *(Fraunhofer IEM, Germany)*

Student Volunteer Co-Chairs
Jacob Hughes *(King's College London, UK)*
Alisa Joy Maas *(University of Wisconsin-Madison, USA)*

Video Chair
Benjamin Chung *(Northeastern University, UK)*

Program Committee
Robert Atkey *(University of Strathclyde, UK)*
Eva Darulova *(MPI-SWS, Germany)*
Mariangiola Dezani *(Università di Torino Italy)*
Dino Distefano *(Facebook and Queen Mary University of London, UK)*
Derek Dreyer *(MPI-SWS, Germany)*
Sophia Drossopoulou *(Imperial College London, UK)*
Cezara Drăgoi *(INRIA, ENS, CNRS, France)*
Alexey Gotsman *(IMDEA Software Institute, Spain)*
Christian Hammer *(University of Potsdam, Germany)*
Tim Harris *(Amazon, UK)*
Matthias Hauswirth *(Università della Svizzera italiana, Switzerland)*
Akash Lal *(Microsoft Research, India)*
Doug Lea *(State University of New York Oswego, USA)*
Heather Miller *(Carnegie Mellon University, USA)*
Bruno C. d. S. Oliveira *(The University of Hong Kong, Hong Kong)*
Corina S. Pasareanu *(NASA and Carnegie Mellon University, USA)*
David Pearce *(Victoria University of Wellington, New Zealand)*
Luís Pina *(George Mason University, USA)*
Alex Potanin *(Victoria University of Wellington, New Zealand)*
Azalea Raad *(MPI-SWS, Germany)*
Ajitha Rajan *(University of Edinburgh, UK)*
Ilya Sergey *(Yale-NUS College and National University of Singapore, Singapore)*
Manu Sridharan *(University of California, Riverside, USA)*
Emma Söderberg *(Lund University, Sweden)*
Tijs van der Storm *(CWI and University of Groningen, Netherlands)*

External Review Committee
Suparna Bhattacharya *(Hewlett-Packard Enterprise, India)*
Viviana Bono *(University of Torino, Italy)*
Junjie Chen *(Peking University, China)*
Mike Dodds *(Galois, Inc., USA)*
Susan Eisenbach *(Imperial College London, UK)*
Ganesh Gopalakrishnan *(University of Utah, USA)*
Bart Jacobs *(KU Leuven, Belgium)*
Jeroen Ketema *(ESI (TNO), Netherlands)*
Ana Milanova *(Rensselaer Polytechnic Institute, USA)*
Jessica Paquette *(Apple Inc., USA)*
Gregor Richards *(University of Waterloo, Canada)*
Philipp Ruemmer *(Uppsala University, Sweden)*
Alexander J. Summers *(ETH Zurich, Switzerland)*
Martin Vechev *(ETH Zurich, Switzerland)*
John Wickerson *(Imperial College London, UK)*

Artifact Evaluation Committee

Sara Achour *(MIT, USA)*
Julia Belyakova *(Northeastern University, USA)*
Junjie Chen *(Peking University, China)*
Marco Eilers *(ETH Zurich, Switzerland)*
Juan Fumero *(University of Manchester, UK)*
Tianxiao Gu *(Alibaba Group, USA)*
Gowtham Kaki *(Purdue University, USA)*
Maria Kechagia *(University College London, UK)*
David Leopoldseder *(Johannes Kepler University, Austria)*
Yue Li *(Aarhus University, Denmark)*
Michael Marcozzi *(Imperial College London, UK)*
Darya Melicher *(Carnegie Mellon University, USA)*
Lisa Nguyen Quang Do *(Paderborn University, Germany)*
Khanh Nguyen *(University of California, Los Angeles, USA)*
Burcu Kulahcioglu Ozkan *(MPI-SWS, Germany)*
Christian Schilling *(IST Austria, Austria)*
Vanya Yaneva *(University of Edinburgh, UK)*

Doctoral Symposium Committee

Phi-Diep Bui *(Uppsala University, Sweden)*
Olivier Flückiger *(Northeastern University, USA)*
Remigius Meier *(ETH Zurich, Switzerland)*
Charith Mendis *(MIT CSAIL, USA)*
Lisa Nguyen Quang Do *(Paderborn University, Germany)*
Nathalie Oostvogels *(Vrije Universiteit Brussel, Belgium)*
Hila Peleg *(Technion, Israel)*
Michael Reif *(TU Darmstadt, Germany)*
Andreas Schuler *(University of Applied Sciences Upper Austria, Austria)*
Ilina Stoilkovska *(Vienna University of Technology, Austria)*
Kirshanthan Sundararajah *(Purdue University, USA)*
Yanlin Wang *(University of Hong Kong, China)*

Doctoral Symposium Academic Panel

Ben Hermann *(University of Paderborn, Germany)*
Neel Krishnaswami *(University of Cambridge, UK)*
Guido Salvaneschi *(TU Darmstadt, Germany)*
Eelco Visser *(Delft University of Technology, Netherlands)*

Posters Committee

David Darais *(University of Vermont, USA)*
Stefan Kruüger *(University of Paderborn, Germany)*
Michael Reif *(TU Darmstadt, Germany)*
Michael D. Shah *(Northeastern University, USA)*
Justin Smith *(North Carolina State University, USA)*

External Reviewers

Samer Al-Kiswani
Michael Emmi
Bernd Finkbeiner
Colin Gordon
Xuejing Huang
Tiark Rompf
Anthony Sloane
Alexandros Tasos
Laurence Tratt
Louis Wasserman
Matthew Windsor
Ningning Xie
Hao Xu
Weixin Zhang
Jinxu Zhao

33rd European Conference on Object-Oriented Programming (ECOOP 2019).
Editor: Alastair F. Donaldson
LIPICS Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

List of Authors

Alex Aiken (11)
Stanford University, USA

Saswat Anand (11)
Stanford University, USA

Alen Arslanagić (23)
University of Groningen, The Netherlands

Ellen Arteca (16)
Northeastern University, Boston, MA, USA

Osbert Bastani (11)
University of Pennsylvania, Philadelphia, USA

Walter Binder (20)
Università della Svizzera italiana, Switzerland

Eric Bodden (21)
Heinz Nixdorf Institute, Paderborn University, Paderborn, Germany; Fraunhofer IEM, Paderborn, Germany

Daniele Bonetta (20)
Oracle Labs, United States

Eric Campbell (12)
Cornell University, Ithaca, NY, USA

Bor-Yuh Evan Chang (1)
University of Colorado Boulder, USA

Benjamin Chung (24)
Northeastern University, Boston, MA, USA

Lazaro Clapp (11)
Stanford University, USA

Dave Clarke (2)
Storytel, Stockholm, Sweden

Jan de Muijnck-Hughes (6)
University of Glasgow, UK

Coen De Roover (10)
Software Languages Lab, Vrije Universiteit Brussel, Belgium

Isabella Defilippis (20)
Universidad Privada Boliviana, Bolivia

Pantazis Deligiannis (18)
Microsoft Research, Redmond, USA

Julian Dolby (21)
IBM Research, New York, USA

Matthias Eichholz (12)
Technische Universität Darmstadt, Germany

Kiko Fernandez-Reyes (2)
Uppsala University, Sweden

Nate Foster (12)
Cornell University, Ithaca, NY, USA

George Fourtounis (15)
University of Athens, Department of Informatics and Telecommunications, Greece

Philippa Gardner (9)
Imperial College London, UK

Yossi Gil (13)
Technion I.I.T Computer Science Dept., Haifa, Israel

Elisa Gonzalez Boix (27)
Vrije Universiteit Brussel, Belgium

Krishnan Govindraj (18)
Microsoft Research, Bangalore, India

Robbert Gurdeep Singh (27)
Universiteit Gent, Belgium

Jafar Hamin (19)
imec-DistriNet, Department of Computer Science, KU Leuven, Belgium

Ludovic Henrio (2)
Univ Lyon, EnsL, UCBL, CNRS, Inria, LIP, France

Michael Homer (5)
School of Engineering and Computer Science, Victoria University of Wellington, New Zealand

Bart Jacobs (19)
imec-DistriNet, Department of Computer Science, KU Leuven, Belgium

Lingxiao Jiang (22)
School of Information Systems, Singapore Management University, Singapore

Einar Broch Johnsen (2)
University of Oslo, Norway

Hong Jin Kang (22)
School of Information Systems, Singapore Management University, Singapore

Neelakantan R. Krishnaswami (9, 14)
Department of Computer Science and Technology, University of Cambridge, United Kingdom

33rd European Conference on Object-Oriented Programming (ECOOP 2019).
Editor: Alastair F. Donaldson
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Raja Krishnaswamy (18)
Microsoft Azure, Redmond, USA

Akash Lal (18)
Microsoft Research, Bangalore, India

Julia Lawall (22)
Sorbonne Université/Inria/LIP6, France

Benjamin Livshits (8)
Imperial College London, UK; Brave Software, London, UK

David Lo (22)
School of Information Systems, Singapore Management University, Singapore

Linghui Luo (21)
Heinz Nixdorf Institute, Paderborn University, Paderborn, Germany

Rupak Majumdar (28)
MPI-SWS, Saarbrücken, Germany

Petar Maksimović (9)
Imperial College London, UK; Mathematical Institute SASA, Serbia

Dhruv C. Makwana (14)
Unaffiliated

Stefan Marr (5, 27)
School of Computing, University of Kent, UK

Hidehiko Masuhara (17)
Tokyo Institute of Technology, Japan

Shawn Meier (1)
University of Colorado Boulder, USA

Mira Mezini (12)
Technische Universität Darmstadt, Germany

Aleksandr Misonizhnik (7)
JetBrains Research, Saint Petersburg State University, Russia

Dmitry Mordvinov (7)
JetBrains Research, Saint Petersburg State University, Russia

Sergio Mover (1)
École Polytechnique, Institute Polytechnique de Paris, Palaiseau, France

Suvam Mukherjee (18)
Microsoft Research, Bangalore, India

Gilles Muller (22)
Sorbonne Université/Inria/LIP6, France

James Noble (5)
School of Engineering and Computer Science, Victoria University of Wellington, New Zealand

Sergio Oporto (20)
Universidad Privada Boliviana, Bolivia

Lionel Parreaux (25)
DATA Lab, EPFL, Lausanne, Switzerland

Daniel A. A. Pelsmaeker (26)
Delft University of Technology, Delft, The Netherlands

Marcus Pirron (28)
MPI-SWS, Saarbrücken, Germany

Jorge A. Pérez (23)
University of Groningen, The Netherlands

Nitin John Raj (18)
International Institute of Information Technology, Hyderabad, India

Aseem Rastogi (18)
Microsoft Research, Bangalore, India

Chandramouleswaran Ravichandran (18)
Microsoft Azure, Redmond, USA

Gregor Richards (16)
University of Waterloo, Waterloo, ON, Canada

Richard Roberts (5)
School of Engineering and Computer Science, Victoria University of Wellington, New Zealand

Eduardo Rosales (20)
Università della Svizzera italiana, Switzerland

Andrea Rosà (20)
Università della Svizzera italiana, Switzerland

Ori Roth (13)
Technion I.I.T Computer Science Dept., Haifa, Israel

Guido Salvaneschi (3, 12)
Technische Universität Darmstadt, Germany

Christophe Scholliers (27)
Universiteit Gent, Belgium

Amir Shaikhha (25)
Department of Computer Science, University of Oxford, UK

Rahul Sharma (11)
Microsoft Research, Bangalore, India

Yannis Smaragdakis (15)
University of Athens, Department of Informatics and Telecommunications, Greece

Thodoris Sotiropoulos (8)
Athens University of Economics and Business, Greece

Matthias Springer (17)
Tokyo Institute of Technology, Japan

Quentin Stiévenart (10)
Software Languages Lab, Vrije Universiteit Brussel, Belgium

Haiyang Sun (20)
Università della Svizzera italiana, Switzerland

Ferdian Thung (22)
School of Information Systems, Singapore Management University, Singapore

Andrew Tolmach (4)
Portland State University, Portland, OR, USA

Carmen Torres Lopez (27)
Vrije Universiteit Brussel, Belgium

Alexi Turcotte (16)
Northeastern University, Boston, MA, USA

Hendrik van Antwerpen (26)
Delft University of Technology, Delft, The Netherlands

Noah Van Es (10)
Software Languages Lab, Vrije Universiteit Brussel, Belgium

Wim Vanderbauwhede (6)
University of Glasgow, UK

Vlad Vergu (4)
Delft University of Technology, Delft, The Netherlands

Alex Villazón (20)
Universidad Privada Boliviana, Bolivia

Eelco Visser (4, 26)
Delft University of Technology, Delft, The Netherlands

Jan Vitek (24)
Northeastern University, Boston, MA, USA; Czech Technical University in Prague, Czech Republic

Erik Voogd (23)
University of Groningen, The Netherlands

Conrad Watt (9)
University of Cambridge, UK

Pascal Weisenburger (3)
Technische Universität Darmstadt, Germany

Tobias Wrigstad (2)
Uppsala University, Sweden

Nobuko Yoshida (28)
Imperial College London, UK

Francesco Zappa Nardelli (24)
Inria of Paris, Paris, France

Damien Zufferey (28)
MPI-SWS, Saarbrücken, Germany

Lifestate: Event-Driven Protocols and Callback Control Flow

Shawn Meier
University of Colorado Boulder, USA
shawn.meier@colorado.edu

Sergio Mover
École Polytechnique, Institute Polytechnique de Paris, Palaiseau, France
sergio.mover@lix.polytechnique.fr

Bor-Yuh Evan Chang
University of Colorado Boulder, USA
evan.chang@colorado.edu

Abstract

Developing interactive applications (apps) against event-driven software frameworks such as Android is notoriously difficult. To create apps that behave as expected, developers must follow complex and often implicit *asynchronous programming protocols*. Such protocols intertwine the proper registering of callbacks to receive control from the framework with appropriate application-programming interface (API) calls that in turn affect the set of possible future callbacks. An app violates the protocol when, for example, it calls a particular API method in a state of the framework where such a call is invalid. What makes automated reasoning hard in this domain is largely what makes programming apps against such frameworks hard: the specification of the protocol is unclear, and the control flow is complex, asynchronous, and higher-order. In this paper, we tackle the problem of specifying and modeling event-driven application-programming protocols. In particular, we formalize a core meta-model that captures the dialogue between event-driven frameworks and application callbacks. Based on this meta-model, we define a language called *lifestate* that permits precise and formal descriptions of application-programming protocols and the callback control flow imposed by the event-driven framework. Lifestate unifies modeling what app callbacks can expect of the framework with specifying rules the app must respect when calling into the framework. In this way, we effectively combine lifecycle constraints and typestate rules. To evaluate the effectiveness of lifestate modeling, we provide a dynamic verification algorithm that takes as input a trace of execution of an app and a lifestate protocol specification to either produce a trace witnessing a protocol violation or a proof that no such trace is realizable.

2012 ACM Subject Classification Software and its engineering → Software verification

Keywords and phrases event-driven systems, application-programming protocols, application framework interfaces, callbacks, sound framework modeling, predictive dynamic verification

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2019.1

Related Version An extended version of the paper is available at [33], `https://arxiv.org/abs/1906.04924`.

Supplement Material ECOOP 2019 Artifact Evaluation approved artifact available at `https://dx.doi.org/10.4230/DARTS.5.2.13`

Funding This material is based on research sponsored by DARPA under agreement number FA8750-14-2-0263.

Acknowledgements Many thanks to Edmund S. L. Lam, Chance Roberts, and Chou Yi for help in gathering traces, as well as Alberto Griggio for a convenient tool for running tests. We also thank Aleksandar Chakarov, Maxwell Russek, the Fixr Team, and the University of Colorado Programming Languages and Verification (CUPLV) Group for insightful discussions, as well as the anonymous reviewers for their helpful comments.

33rd European Conference on Object-Oriented Programming (ECOOP 2019).
Editor: Alastair F. Donaldson; Article No. 1; pp. 1:1–1:29
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

```
try { progress.dismiss(); } catch (IllegalArgumentException ignored) {} // race condition?
```

Figure 1 A protocol "fix" [10]. The dismiss call throws an exception if called in an invalid state.

1 Introduction

We consider the essential problem of checking that an application (app) programmed against an event-driven framework respects the required application-programming protocol. In such frameworks, apps implement *callback* interfaces so that the app is notified when an *event* managed by the framework occurs (e.g., a user-interface (UI) button is pressed). The app may then delegate back to the framework through calls to the application programming interface (API), which we term *callin* by analogy to callback. To develop working apps, the programmer must reason about hidden *callback control flow* and often implicit asynchronous programming protocols.

Couple difficult reasoning about the space of possible control flow between callbacks with insufficient framework documentation, and it is unsurprising to find some questionable "fixes" for protocol violations. In Figure 1, we show a snippet found on GitHub. The "race condition?" comment is quoted directly from the app developer. The same asynchronous, implicitly defined, control flow that make it difficult for the app developer to reason about his app is also what makes verifying the absence of such protocol violations hard.

In this paper, we focus on the problems of specifying event-driven protocols (i.e., specifying when the invocation of a callin in the app code causes a protocol violation) and modeling the callback control flow (i.e., modeling the possible executions of callbacks).

Lifecycle Automata are Insufficient for Modeling Callback Control Flow. *Lifecycle* automata are a common representation used to model callback control flow that is both central to Android documentation [1, 39] and prior Android analysis techniques – both static and dynamic ones (e.g., [5, 32, 8]). In Figure 2, we show a lifecycle automaton for the Activity class of the Android framework. The black, solid edges are the edges present in the Android documentation [1] showing common callback control flow. These edges capture, for example, that the app first receives the onStart callback before entering a cycle between the onResume and the onPause callbacks. But this clean and simple class-based model quickly becomes insufficient when we look deeper.

First, there are complex relationships between the callbacks on "related" objects. For

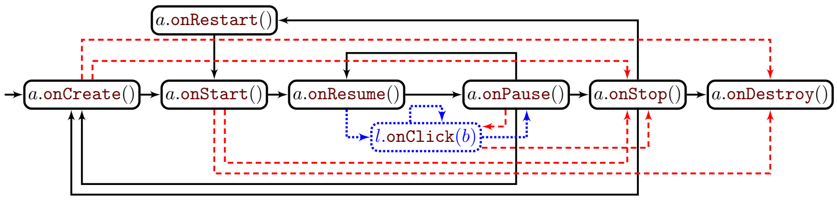


Figure 2 The Activity lifecycle automaton from the Android documentation [1] (shown with solid, black edges →). To capture callback control flow between "related" objects, such component lifecycles are often instantiated and refined with additional callbacks from other objects, such as a onClick callback from the OnClickListener interface (shown with dotted, blue edges ⇢). But there are also less common callback control-flow paths that are often undocumented or easily missed, such as the additional edges induced by an invocation in the app code of the finish callin (shown as dashed, red edges ⇢).

example, an *OnClickListener* object l with an *onClick* callback may be "registered" on a *View* object v that is "attached" to an *Activity* object a. Because of these relationships, the callback control flow we need to capture is somewhat described by modifying the lifecycle automaton for *Activity* a with the additional blue, dotted edges to and from *onClick* (implicitly for *OnClickListener* l) in Figure 2. This modified lifecycle encodes framework-specific knowledge that the *OnClickListener* l's *onClick* callback happens only in the "active" state of *Activity* a between its *onResume* and *onPause* callbacks, which typically requires a combination of static analysis on the app and hard-coded rules to connect callbacks on additional objects such as *OnClickListener*s to component lifecycles such as *Activity*. We refer to such callback control-flow models based on such refined lifecycle automatons as lifecycle++ models.

Second, there are less common framework-state changes that are difficult to capture soundly and precisely. For example, an analysis that relies on a callback control-flow model that does not consider the intertwined effect of a finish call may be unsound. The red, dashed edges represent callback control flow that are not documented (and thus missing from typical callback control flow models). Each one of these edges specifies different possible callback control flow that the framework imposes depending on *if and when* the app invokes the finish callin inside one of the *Activity*'s callbacks. Of course, the lifecycle automaton can be extended to include these red edges. However, this lifecycle automaton is now quite imprecise in the common case because it does not express precisely when certain callback control-flow paths are spurious (i.e., depending on where finish is not called). Figure 2 illustrates why developing callback control flow models is error prone: the effect of calls to finish are subtle and poorly understood.

It is simply too easy to miss possible callback control flow – an observation also made by Wang et al. [52] about lifecycle models. While lifecycle automata are useful for conveying the intuition of callback control flow, they are often insufficiently precise and easily unsound.

In this paper, we re-examine the process of modeling callback control flow. In prior work, modeling callback control flow was almost always a secondary concern in service to, and often built into, a specific program analysis where the analysis abstraction may reasonably mask unsound callback control flow. Instead, we consider modeling callback control flow independent of any analysis abstraction – we identify and formalize the key aspects to effectively model event-driven application-programming protocols at the app-framework interface, such as the effect of callin and callback invocations on the subsequent callback control flow, This first-principles approach enables us to *validate* callback control-flow soundness with real execution traces against the event-driven framework implementation. It is through this validation step that we discovered the red, dashed edges in Figure 2.

Contributions. We make the following contributions:

- We identify essential aspects of event-driven control flow and application-programming protocols to formalize a core abstract machine model λ_{life} (Section 3). This model provides a formal basis for thinking about event-driven frameworks and their application-programming protocols.
- We define a language for simultaneously capturing event-driven application-programming protocols and callback control flow called *lifestates*, which both *model* what callback invocations an app can expect from the framework and *specify* rules the app must respect when calling into the framework (Section 4). Intuitively, lifestates offer the ability to specify traces of the event-driven program in terms of an abstraction of the observable interface between the framework and the app. And thus, this definition leads to a methodology for empirically validating lifestate models against actual interaction traces.

- We define lifestate validation and dynamic lifestate verification. And then, we encode them as model checking problems (Section 5). Given an app-framework interaction trace and a lifestate model, validation checks that the trace is in the abstraction of the observable interface defined by the model. This validation can be done with corpora of traces recorded from any set of apps interacting with the same framework because, crucially, the lifestate model speaks only about the app-framework interface. Then, given a trace, dynamic lifestate verification attempts to prove the absence of a rearrangement of the recorded events that could cause a protocol violation. Rearranging the execution trace of events corresponds to exploring a different sequence of external inputs and hence discovering possible protocol violations not observed in the original trace.
- We implement our model validation and trace verification approach in a tool called Verivita and use it to empirically evaluate the soundness and precision of callback control flow models of Android (Section 6). Our results provide evidence for the hypotheses that lifecycle models, by themselves, are insufficiently precise to verify Android apps as conforming to the specified protocols, that model validation on large corpora of traces exposes surprising unsoundnesses, and that lifestates are indeed useful.

2 Overview: Specifying and Modeling Lifestates

Here, we illustrate the challenges in specifying and modeling event-driven application-programming protocols. In particular, we motivate the need for lifestates that permit specifying the intertwined effect of callin and callback invocations. We show that even if an app is buggy, it can be difficult to witness the violation of the Android application programming protocol. Then, more importantly, we show how an appropriate fix is both subtle to reason about and requires modeling the complex callback control flow that depends on the previous execution of not only the callbacks but also the callins.

Our running example (code shown in Figure 3) is inspired by actual issues in AntennaPod [16], a podcast manager with 100,000+ installs, and the Facebook SDK for Android [27]. The essence of the issue is that a potentially time-consuming background task is started by a user interaction and implemented using the *AsyncTask* framework class. Figure 3 shows buggy code that can potentially violate the application-programming protocol for

```
   class RemoverActivity extends Activity {
    FeedRemover remover;
    void onCreate() {
1    Button button = ...;
2    remover = new FeedRemover(this);
3    button.setOnClickListener(
4     new OnClickListener() {
       void onClick(View view) {
5       remover.execute();    ⚠
       }
      });
    }
   }
```

```
   class FeedRemover extends AsyncTask {
    RemoverActivity activity;
    void doInBackground() {
     ... remove feed ...
    }
    void onPostExecute() {
     // return to previous activity
6    activity.finish();
    }
   }
```

Figure 3 An example app that violates the protocol specified by the interaction of the Android framework components *AsyncTask*, *Button*, and *OnClickListener*. On line 5, `remover.execute()` (marked with ⚠) can throw an `IllegalStateException` if the `remover` task is already running.

AsyncTask. The remover.execute() call (marked with ⚠) throws an IllegalStateException if the *AsyncTask* *t* instance, pointed-to by remover, is already running. So a protocol rule for *AsyncTask* is that *t*.execute() cannot be called twice for the same *AsyncTask* *t*. The IllegalStateException type is commonly used to signal a protocol violation and has been shown to be a significant source of Android crashes [28].

In Figure 3, the RemoverActivity defines an app window that, on creation (via the *onCreate* callback), registers a click listener (via the button.setOnClickListener(...) call on line 3). This registration causes the framework to notify the app of a button click through the *onClick* method. When that happens, the *onClick* callback starts the FeedRemover asynchronous task (via the remover.execute() call on line 5). What to do asynchronously is defined in the *doInBackground* callback, and when the FeedRemover task is done, the framework delegates to the *onPostExecute* callback, which closes the RemoverActivity (via the call to activity.finish()).

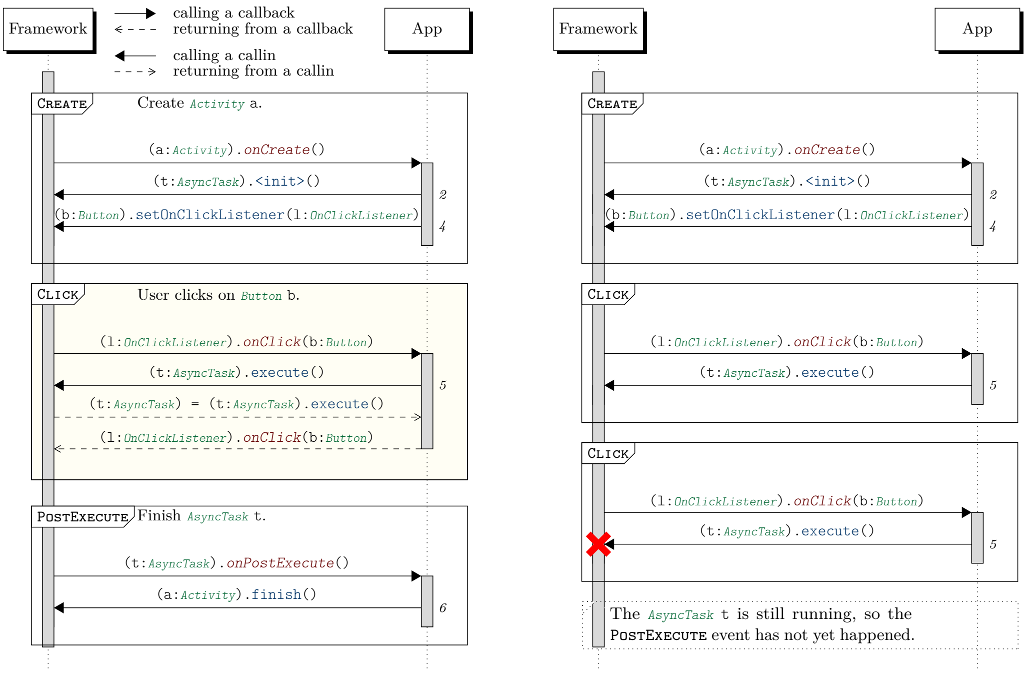
We diagram a common-case execution trace in Figure 4.a. Even though the app is buggy, the trace does not witness the protocol violation. The exception does not manifest because the user only clicks once (Click) before the FeedRemover task completes and generates the post-execute event (PostExecute). And so the (t:*AsyncTask*).execute() callin on the *AsyncTask* instance t is executed only once before the activity is closed (cf. the *onClick* and *onPostExecute* callbacks in Figure 3).

If typically the *Activity* is quickly destroyed after the button click, then seeing a protocol violation in a test is quite unlikely. However, it is possible to click a second time before the *AsyncTask* completes, thereby witnessing a protocol violation. We show this error trace in Figure 4.b: when the app invokes the callin (t:*AsyncTask*).execute() for the second time in the second Click event, the framework is in a state that does not allow this transition. We say that the callin invocation is *disallowed* at this point, and apps must only invoke allowed callins. While the original trace Create;Click;PostExecute does not concretely witness the protocol violation, it has sufficient information to predict the error trace Create;Click;Click. It may, however, be difficult to reproduce this error trace: the button must be pressed twice before the activity.finish() method is called by the PostExecute event destroying the *Activity*. But how can we predict this error trace from the original one?

2.1 Predict Violations from Recorded Interactions

We define the dynamic lifestate verification problem as predicting an error trace that (possibly) witnesses a protocol violation from a trace of interactions or proving that no such error trace exists. Concretely, the input to dynamic lifestate verification is an interaction trace like the one illustrated in Figure 4.a. These traces record the sequence of invocations and returns of callbacks and callins between the framework and the app that result from an interaction sequence. A recorded trace includes the concrete method arguments and return values (e.g., the instance t from the diagrams corresponds to a concrete memory address).

The main challenge, both for the app developer and dynamic lifestate verification, is that the relevant sequence of events that leads to a state where a callin is disallowed is *hidden* inside the framework. The developer must reason about the evolving internal state of the framework by considering the possible callback and callin interactions between the app and the framework to develop apps that both adhere to the protocol and behave intuitively. To find a reasonable fix for the buggy app from Figure 3, let us consider again the error trace shown in Figure 4.b. Here, the developer has to reason that the (t:*AsyncTask*).execute() callin is allowed as soon as t is initialized by the call to (t:*AsyncTask*).<init>() in the Create event and is *disallowed* just after the first call to (t:*AsyncTask*).execute() in the first Click


Framework
App
calling a callback
returning from a callback
calling a callin
returning from a callin
Create
Create Activity a.
(a:Activity).onCreate()
(t:AsyncTask).<init>()
(b:Button).setOnClickListener(l:OnClickListener)
2
4
Click
User clicks on Button b.
(l:OnClickListener).onClick(b:Button)
(t:AsyncTask).execute()
(t:AsyncTask) = (t:AsyncTask).execute()
(l:OnClickListener).onClick(b:Button)
5
PostExecute
Finish AsyncTask t.
(t:AsyncTask).onPostExecute()
(a:Activity).finish()
6
Framework
App
Create
(a:Activity).onCreate()
(t:AsyncTask).<init>()
(b:Button).setOnClickListener(l:OnClickListener)
2
4
Click
(l:OnClickListener).onClick(b:Button)
(t:AsyncTask).execute()
5
Click
(l:OnClickListener).onClick(b:Button)
(t:AsyncTask).execute()
5
The AsyncTask t is still running, so the
PostExecute event has not yet happened.

(4.a) A trace that does not witness a protocol violation since the callin (t:*AsyncTask*).execute() on t is executed only once.

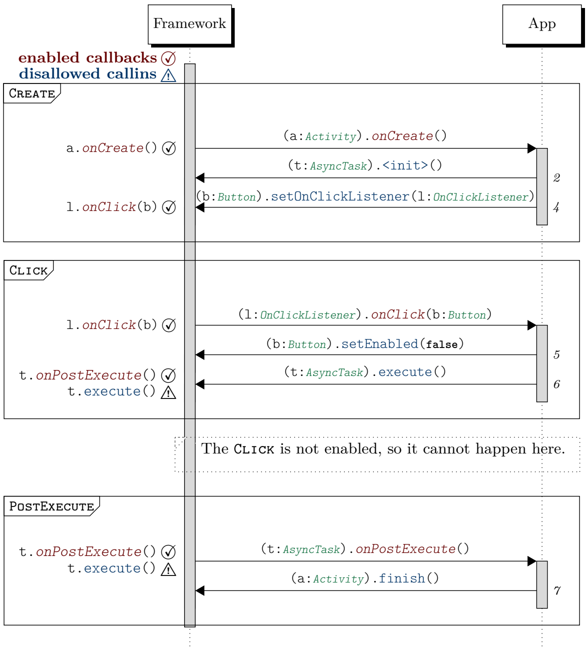
(4.b) The Create;Click;Click interaction sequences witnesses the no-execute-call-on-already-executing-*AsyncTask* protocol violation.

Figure 4 We visualize the interface between an event-driven framework and an app as a dialog between two components. With execution time flowing downwards as a sequence events, control begins on the left with the framework receiving an event. Focusing on the highlighted Click event in Figure 4.a, when a user clicks on the button corresponding to object b of type *Button*, the *onClick* callback is invoked by the framework on the registered listener l. For clarity, we write method invocations with type annotations (e.g., (l:*OnClickListener*).*onClick*(b:*Button*)), and variables b and l stand for some concrete instances (rather than program or symbolic variables). The app then delegates back to the framework by calling an API method to start an asynchronous task t via (t:*AsyncTask*).execute(). To connect with the app source code, we label the callins originating from the app timeline with the corresponding program point numbers in Figure 3. Here, we can see clearly a *callback* as any app method that the framework can call (i.e., with an arrow to the right →), and a *callin* as any framework method that an app can call (i.e., with an arrow to the left ←). We show returns with dashed arrows (but sometimes elide them when they are unimportant).

event. That is, the developer must reason about what sequence of events and callins determine when a callin is allowed or disallowed. Since callins are invoked inside callback methods and callback methods are in turn invoked by the framework to notify the app of an event, the internal framework state determines what events can happen when and hence the callback control flow. In particular, the internal framework state determines when the Create and Click events are *enabled* (i.e., can happen) during the execution. Thus to properly fix this app, the developer must ensure that Create happens before a Click and then only a single Click happens before a PostExecute. How can the app developer constrain the external interaction sequence to conform to this property?

In Figure 5.a, we show a fix based on the above insight that is particularly challenging to verify. The fix adds line 5 that disables *Button* button to indicate when the task has already been started. Thus, this modified version does not violate the no-execute-call-on-already-

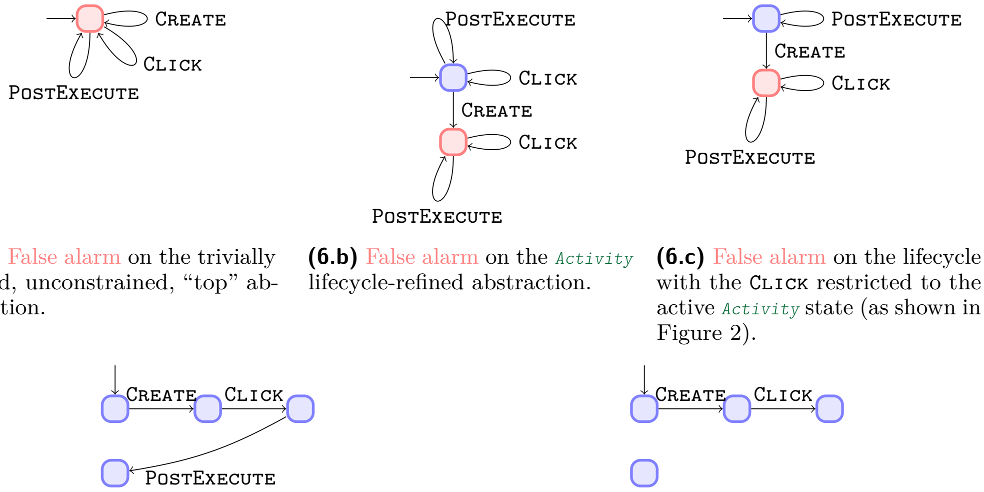
```
  class RemoverActivity extends Activity {
   FeedRemover remover;
   void onCreate() {
1   Button button = ...;
2   remover = new FeedRemover(this);
3   button.setOnClickListener(
4    new OnClickListener() {
      void onClick(View view) {
5 +    button.setEnabled(false);
6      remover.execute();   ⚠
      }
     });
   }
  }

  class FeedRemover extends AsyncTask {
   RemoverActivity activity;
   void doInBackground() {
    ... remove feed ...
   }
   void onPostExecute() {
    // return to previous activity
7   activity.finish();
   }
  }
```



(5.a) A button.setEnabled(false) call prevents the user from clicking, triggering the *onClick* callback.

(5.b) The enabled callbacks and disallowed callins are shown along the CREATE;CLICK;POSTEXECUTE trace from the fixed app in 5.a.

Figure 5 A fixed version of the app from Figure 3 that adheres to the application-programming protocol. The annotations in 5.b show that after the call to (b:*Button*).setEnabled(**false**), the l.*onClick*(b) callback is no longer enabled, and thus the app can assume that the framework will not call l.*onClick*(b) at this point.



(6.a) False alarm on the trivially sound, unconstrained, "top" abstraction.

(6.b) False alarm on the *Activity* lifecycle-refined abstraction.

(6.c) False alarm on the lifecycle with the CLICK restricted to the active *Activity* state (as shown in Figure 2).

(6.d) Verified safe when we consider the effect of *Button*.setEnabled(...).

(6.e) This unsound abstraction is missing the POSTEXECUTE edge.

Figure 6 In previous works, models are generated for an application restricting the possible order of callbacks. In this figure, we show four sound abstractions with different levels of precision, indicating whether they can verify our fixed application 5.a, as well as one unsound abstraction.

executing-*AsyncTask* protocol on line 6. To reason precisely enough about this fix, we must know that the `button.setEnabled(false)` call changes internal framework state that prevents the *onClick* from happening again. Note that this need to reason about complex control flow arises from the interactions between just two framework types *Button* and *AsyncTask* – not to mention that these two are amongst the simplest framework types in Android. There is a clear need here for better automated reasoning tools to support the app developer.

Verivita Approach. Our dynamic verification approach explores all the possible sequences of interactions that can be obtained by replicating, removing, and reordering the events in a trace. By rearranging event traces, the algorithm statically explores different input sequences of events that a user interaction could generate. The algorithm applied to the CREATE;CLICK;POSTEXECUTE trace in Figure 4.a from the buggy app version indeed yields the error trace CREATE;CLICK;CLICK (shown in Figure 4.b). But more critically, our approach also makes it possible to prove that the fixed app version does not have any traces that violates the protocol (by rearranging CREATE;CLICK;POSTEXECUTE).

Central to our approach is capturing the essential, hidden framework state – tracking the set of enabled callbacks and the set of disallowed callins. Figure 5.b illustrates this model state along a trace from the fixed app. After the first CLICK, the application disables the button to prevent a second CLICK via the call to `(b:`*Button*`).setEnabled(false)`, which at that point removes `l.`*onClick*`(b)` from the set of enabled callbacks the framework can trigger.

Verivita addresses the dynamic verification problem by reducing it to a model checking problem. The model is a transition system with

(i) states abstracting the set of enabled callbacks and disallowed callins and
(ii) transitions capturing the possible replication, removing, and reordering of a given interaction trace.

The safety property of interest is that the transition system never visits a disallowed callin. How can we construct such a transition system that over-approximates concrete behavior while being precise enough to make alarm triage feasible? As alluded to in Section 1, lifestate specification is crucial here.

2.2 Specify Event-Driven Protocols and Model Callback Control Flow

In Figure 6, we illustrate the essence of callback control-flow modeling as finite-state automata that over-approximate rearrangements of the CREATE;CLICK;POSTEXECUTE trace shown in Figure 5.b. Automaton 6.a exhibits the trivially sound, unconstrained, "top" abstraction that considers all replications, removals, and reorderings of the interaction trace. This abstraction is the one that assumes all callbacks are always enabled. Since a possible trace in this abstraction includes two CLICK events, a sound verifier must alarm. Meanwhile, Automaton 6.b shows a refined abstraction encoding the Android-specific *Activity* lifecycle. The abstraction is framework-specific but application-independent and captures that the CREATE event cannot happen more than once. The abstraction shown by Automaton 6.b is also insufficient to verify the trace from the fixed app because two CLICK events are still possible.

Automaton 6.c shows a refined, lifecycle++ abstraction that considers the *Activity* lifecycle with additional constraints on an "attached" CLICK event. This abstraction is representative of the current practice in callback control-flow models (e.g., [5, 32, 8]). While Automaton 6.c restricts the CLICK event to come only after the CREATE event, the abstraction is still too over-approximate to verify that the trace from the fixed app is safe – two CLICK events are still possible with this model. But worse is that this model is still, in essence, a lifecycle model that is constrained by Android-specific notions like `View` attachment, `Listener` registration, and the "live" portion of lifecycles. In existing analysis tools, such constrained lifecycle models are typically hard-coded into the analyzer.

We need a better way to capture how the application may affect callback control flow. In this example, we need to capture the effect of the callin `button.setEnabled(false)` at line 5 in Figure 5.a, which is the only difference with the buggy version in Figure 3. The modeling needs to be expressive to remove such infeasible traces and compositional to express state changes independently. Thus, the role of lifestate specification is to describe how the internal model state is updated by observing the history of intertwined callback and callin invocations. For example, we write

$$(\ell_\mathrm{b}{:}\mathtt{Button}).\mathtt{setEnabled}(\mathtt{false}) \nrightarrow (\ell_\mathrm{l}{:}\mathtt{OnClickListener}).\mathtt{onClick}(\ell_\mathrm{b}{:}\mathtt{Button}) \ \text{(for all } \ell_\mathrm{l},\ \ell_\mathrm{b}\text{)}$$

to model when $(\ell_\mathrm{b}{:}\mathtt{Button}).\mathtt{setEnabled}(\mathtt{false})$ is invoked, the click callback is *disabled* on the same button ℓ_b (on all listeners ℓ_l). Also, we similarly specify the safety property of interest

$$(\ell_\mathrm{t}{:}\mathtt{AsyncTask}).\mathtt{execute}() \nrightarrow (\ell_\mathrm{t}{:}\mathtt{AsyncTask}).\mathtt{execute}() \ \text{(for all } \ell_\mathrm{t}\text{)}$$

that when $(\ell_\mathrm{t}{:}\mathtt{AsyncTask}).\mathtt{execute}()$ is called on a task ℓ_t, it *disallows* itself. And analogously, lifestates include specification forms for *enabling callbacks* or *allowing callins*.

Lifestate uniformly models the callback control-flow and specifies event-driven application-programming protocols. The rules that enable and disable callbacks model what callbacks the framework can invoke at a specific point in the execution of the application, while the rules that disallow and allow callins specify what callins the application must invoke to respect the protocol. What makes lifestate unique compared to typestates [50] or lifecycle automata is this unification of the intertwined effects of callins and callbacks on each other.

The complexity of the implicit callback control flow is what makes expressing and writing correct models challenging. An issue whose importance is often under-estimated when developing callback control-flow models is how much the model faithfully reflects the framework semantics. How can we *validate* that a lifestate specification is a correct model of the event-driven framework?

Validating Event-Driven Programming Protocols. As argued in Section 1, a key concern when developing a framework model is that it must over-approximate the possible real behavior of the application. The "top" model as shown in Automaton 6.a trivially satisfies this property, and it may be reasonable to validate an application-independent lifecycle model like Automaton 6.c. However, as we have seen, verifying correct usage of event-driven protocols typically requires callback control-flow models with significantly more precision.

Automaton 6.d shows a correct lifestate-abstraction that contains an edge labeled PostExecute. We express this edge with the rule shown below:

$$(\ell_\mathrm{t}{:}\mathtt{AsyncTask}).\mathtt{execute}() \rightarrow (\ell_\mathrm{t}{:}\mathtt{AsyncTask}).\mathtt{onPostExecute}() \ \text{(for all } \ell_\mathrm{t}\text{)}$$

This rule states that when $(\ell_\mathrm{t}{:}\mathtt{AsyncTask}).\mathtt{execute}()$ is called, its effect is to enable the callback $(\ell_\mathrm{t}{:}\mathtt{AsyncTask}).\mathtt{onPostExecute}()$ on the `AsyncTask` ℓ_t.

If we do not model this rule, we obtain the abstraction in Automaton 6.e. The lifestate model is *unsound* since it misses the PostExecute edge.

The trace Create;Click;PostExecute shown in Figure 5.b is a witness of the unsoundness of the abstraction: Automaton 6.e accepts only proper prefixes of the trace (e.g., Create;Click), and hence the abstraction does not capture all the possible traces of the app.

We can thus use interaction traces to validate lifestate rules: a set of lifestate rules is valid if the abstraction accepts all the interaction traces. The validation applied to the abstraction shown in Automaton 6.e demonstrates that the abstraction accepts `Create`;`Click` as the longest prefix of the trace `Create`;`Click`;`PostExecute`. This information helps to localize the cause for unsoundness since we know that after the sequence `Create`;`Click`, the callback `PostExecute` is (erroneously) disabled.

The encoding of the abstraction from lifestate rules is a central step to perform model validation and dynamic verification. At this point, we still cannot directly encode the abstraction since the lifestate rules contain universally-quantified variables. How can we encode the lifestate abstraction as a transition system amenable to check language inclusion for validation, and to check safety properties for dynamic verification?

From Specification to Validation and Verification. Generalizing slightly, we use the term *message* to refer to any observable interaction between the framework and the app. Messages consist of invocations to and returns from callbacks and callins. The abstract state of the transition system is then a pair consisting of the *permitted-back messages* from framework to app and the *prohibited-in messages* from app to framework. And thus generalizing the example rules shown above, a lifestate specification is a set of rules whose meaning is,

> *If* the message history *matches* $\boldsymbol{r}$, *then* the abstract state is updated according to the specified *effect* on the set of permitted-back and prohibited-in messages.

There are many possible choices and tradeoffs for the matching language $\boldsymbol{r}$. As is common, we consider a regular expression-based (i.e., finite automata-based) matching language.

We exploit the structure of the validation and dynamic verification problem to encode the lifestate abstraction. In both problems, the set of possible objects and parameters is finite and determined by the messages recorded in the trace. We exploit this property to obtain a set of *ground* rules (rules without variables). We can then encode each ground rule in a transition system. Since the rule is ground, the encoding is standard: each regular expression is converted to an automaton and then encoded in the transition system, changing the permitted-prohibited state as soon as the transition system visits a trace accepted by the regular expression, which implicitly yields a model like automata 6.d.

Lifestate offers a general and flexible way to specify the possible future messages in terms of observing the past history of messages. It, however, essentially leaves the definition of messages and what is observable abstract. What observables characterize the interactions between an event-driven framework and an app that interfaces with it? And how do these observables define event-driven application-programming protocols and callback control flow?

2.3 Event-Driven App-Framework Interfaces

Lifestate rules are agnostic to the kinds of messages they match and effects they capture on the internal abstract state. To give meaning to lifestates, we formalize the essential aspects of the app-framework interface in an abstract machine model called λ_{life} in Section 3. This abstract machine model formally characterizes what we consider an *event-driven framework*. The λ_{life} abstract machine crisply defines the messages that the app and the framework code exchange and a formal correspondence between concrete executions of the program and the app-framework interface. We use this formal correspondence to define the semantics of the lifestate framework model, its validation problem, and protocol verification.

We do not intend for λ_{life} to capture all aspects of something as complex as Android; rather, the purpose of λ_{life} is to define a "contract" by which to consider a concrete event-driven framework implementation. And thus, λ_{life} also defines the dynamic-analysis instrumentation we perform to record observable traces from Android applications that we then input to the Verivita tool to either validate a specification or verify protocol violations.

Preview. We have given a top-down overview of our approach, motivating with the dynamic protocol verification problem the need for having both a precise callback control-flow model and an event-driven protocol specification. We also presented how the lifestate language addresses this need capturing the intertwined effect of callins and callbacks. In the next sections, we detail our approach in a bottom-up manner – beginning with formalizing the λ_{life} abstract machine model. We show that, assuming such a model of execution, it is possible to provide a sound abstraction of the framework (i.e., no real behavior of the framework is missed by the abstraction) expressed with a lifestate model. We then formalize how we validate such models and how we use lifestates to verify the absence of protocol violations.

3 Defining Event-Driven Application-Programming Protocols

Following Section 2, we want to capture the essence of the app-framework interface with respect to framework-imposed programming protocols. To do so, we first formalize a small-step operational semantics for event-driven programs with an abstract machine model λ_{life}. The λ_{life} abstract machine draws on standard techniques but explicitly highlights enabled events and disallowed callins to precisely define event-driven protocols. We then instrument this semantics to formalize the interface of the event-driven framework with an app, thereby defining the traces of the observable app-framework interface of a λ_{life} program.

This language is intentionally minimalistic to center on capturing just the interface between event-driven frameworks and their client applications. By design, we leave out many aspects of real-world event-driven framework implementations (e.g., Android, Swing, or Node.js), such as typing, object-orientation, and module systems that are not needed for formalizing the dialogue between frameworks and their apps (cf. Section 2). Our intent is to illustrate, through examples, that event-driven frameworks could be implemented in λ_{life} and that λ_{life} makes explicit the app-framework interface to define *observable traces* consisting of *back-messages* and *in-messages* (Section 3.3).

3.1 Syntax: Enabling, Disabling, Allowing, and Disallowing

The syntax of λ_{life} is shown at the top of Figure 7.a, which is a λ-calculus in a let-normal form. The first two cases of expressions e split the standard call-by-value function application into multiple steps (similar to call-by-push-value [30]). The **bind** λ v expression creates a thunk $\kappa = \lambda[v]$ by binding a function value λ with an argument value v. We abuse notation slightly by using λ as the meta-variable for function values (rather than as a terminal symbol). A thunk may be forced by direct invocation **invoke** κ – or indirectly via event dispatch.

▶ **Example 1** (Applying a Function). Let `t` be bound to an *AsyncTask* and *onPostExecute* to an app-defined callback (e.g., *onPostExecute* from Figure 5.a), then the direct invocation of a callback from the framework can be modeled by the two steps of binding and then invoking:

```
let cb = bind onPostExecute t in invoke cb
```

Now in λ_{life}, a thunk κ may or may not have the *permission* to be forced. Revoking and re-granting the permission to force a thunk via direct invocation is captured by the expressions **disallow** κ and **allow** κ, respectively. A protocol violation can thus be modeled by an application invoking a *disallowed* thunk.

$$
\begin{array}{rrll}
\text{expressions} & e \in \mathbf{Expr} ::= & \texttt{bind}\ v_1\ v_2 \mid \texttt{invoke}\ v \mid \texttt{disallow}\ v \mid \texttt{allow}\ v & \text{thunks and calls} \\
 & & \mid \texttt{enable}\ v \mid \texttt{disable}\ v \mid \texttt{force}\ \kappa & \text{events and forcing} \\
 & & \mid v \mid \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 \mid \cdots & \text{other expressions} \\
\text{functions} & \lambda ::= & x\ \texttt{=>}_g\ e & \\
\text{packages} & g ::= & \texttt{app} \mid \texttt{fwk} & \\
\text{values} & v \in \mathbf{Val} ::= & x \mid \lambda \mid \kappa \mid () \mid \cdots \mid \texttt{thk} &
\end{array}
$$

$$
\text{variables} \quad x \in \mathbf{Var} \qquad \text{thunks} \quad \kappa \in \mathbf{Thunk} ::= \lambda[v] \qquad \text{thunk stores} \quad \mu, \nu ::= \cdot \mid \mu;\kappa
$$

$$
\text{continuations} \quad k ::= \bullet \mid k \triangleright x.e \mid \kappa \mid k \gg \kappa \qquad \text{states} \quad \sigma \in \mathbf{State} ::= \langle e, \mu, \nu, k \rangle \mid \mathsf{bad}
$$

(7.a) The syntax and the semantic domains.

$$\boxed{\sigma \longrightarrow \sigma'}$$

$$
\textsc{Enable}\ \frac{}{\langle \texttt{enable}\ \kappa, \mu, \nu, k \rangle \longrightarrow \langle \kappa, \mu;\kappa, \nu, k \rangle}
\qquad
\textsc{Disable}\ \frac{}{\langle \texttt{disable}\ \kappa, \mu;\kappa, \nu, k \rangle \longrightarrow \langle \kappa, \mu, \nu, k \rangle}
\qquad
\textsc{Event}\ \frac{\kappa \in \mu}{\langle v, \mu, \nu, \bullet \rangle \longrightarrow \langle \texttt{force}\ \kappa, \mu, \nu, \kappa \rangle}
$$

$$
\textsc{Disallow}\ \frac{}{\langle \texttt{disallow}\ \kappa, \mu, \nu, k \rangle \longrightarrow \langle \kappa, \mu, \nu;\kappa, k \rangle}
\qquad
\textsc{Allow}\ \frac{}{\langle \texttt{allow}\ \kappa, \mu, \nu;\kappa, k \rangle \longrightarrow \langle \kappa, \mu, \nu, k \rangle}
$$

$$
\textsc{Invoke}\ \frac{\kappa \notin \nu}{\langle \texttt{invoke}\ \kappa, \mu, \nu, k \rangle \longrightarrow \langle \texttt{force}\ \kappa, \mu, \nu, k \rangle}
\qquad
\textsc{InvokeDisallowed}\ \frac{\kappa \in \nu}{\langle \texttt{invoke}\ \kappa, \mu, \nu, k \rangle \longrightarrow \mathsf{bad}}
\qquad
\textsc{Bind}\ \frac{}{\langle \texttt{bind}\ \lambda\ v, \mu, \nu, k \rangle \longrightarrow \langle \lambda[v], \mu, \nu, k \rangle}
$$

$$
\textsc{Force}\ \frac{(x'\ \texttt{=>}_{g'}\ e')[v'] = \kappa}{\langle \texttt{force}\ \kappa, \mu, \nu, k \rangle \longrightarrow \langle [\kappa/\texttt{thk}][v'/x']e', \mu, \nu, k \gg \kappa \rangle}
\qquad
\textsc{Return}\ \frac{}{\langle v, \mu, \nu, k \gg \kappa \rangle \longrightarrow \langle v, \mu, \nu, k \rangle}
\qquad
\textsc{Finish}\ \frac{}{\langle v, \mu, \nu, \kappa \rangle \longrightarrow \langle v, \mu, \nu, \bullet \rangle}
$$

$$
\textsc{Let}\ \frac{}{\langle \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2, \mu, \nu, k \rangle \longrightarrow \langle e_1, \mu, \nu, k \triangleright x.e_2 \rangle}
\qquad
\textsc{Continue}\ \frac{}{\langle v, \mu, \nu, k \triangleright x.e_2 \rangle \longrightarrow \langle [v/x]e_2, \mu, \nu, k \rangle}
$$

(7.b) Semantics. Explicitly enable, disable, disallow, and allow thunks.

Figure 7 λ_{life}, a core model of event-driven programs capturing *enabledness* of events and *disallowedness* of invocations.

The direct invocation expressions are mirrored with expressions for event dispatch. An **enable** κ expression *enables* a thunk κ for the external event-processing system (i.e., gives the system permission to force the thunk κ), while the **disable** κ expression *disables* the thunk κ.

▶ **Example 2** (Enabling an Event). Let `t` be bound to an *AsyncTask* and `handlePostExecute` to an internal framework-defined function for handling a post-execute event, then enqueuing such an event can be modeled by the two steps of binding then enabling:

```
let h = bind handlePostExecute t in enable h
```

By separating function application and event dispatch into binding to create a thunk $\kappa = \lambda[v]$ and then forcing it, we uniformly make thunks the value form that can be granted permission to be invoked (via **allow** κ) or for event dispatch (via **enable** κ). The **force** κ expression is then an intermediate that represents a thunk that is forcible (i.e., has been permitted for forcing via **allow** κ or **enable** κ).

The remainder of the syntax is the standard part of the language: values v, variable binding $\texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2$, and whatever other operations of interest $\cdots$ (e.g., arithmetic, tuples, control flow, heap manipulation). That is, we have made explicit the expressions to expose the app-framework interface and can imagine whatever standard language features in $\cdots$ in framework implementations. The values v of this expression language are variables x, function values λ, thunks κ, unit $()$, and whatever other base values of interest $\cdots$. Two exceptions are that (1) the currently active thunk is available via the `thk` identifier (see Section 3.2) and (2) functions x `=>`$_g$ e are tagged with a package g (see Section 3.3).

3.2 Semantics: Protocol Violations

At the bottom of Figure 7.a, we consider an abstract machine model enriched with an *enabled-events* store μ, and a *disallowed-calls* store ν. These are finite sets of thunks, which we write as a list $\kappa_1; \cdots ;\kappa_n$. The enabled-events store μ saves thunks that are permitted to be forced by the event loop, while the disallowed-calls store ν lists thunks that are *not* permitted to be forced by invocation. These thunk stores make explicit the event-driven application-programming protocol (that might otherwise be implicit in, for example, flag fields and conditional guards).

A machine state $\sigma : \langle e, \mu, \nu, k\rangle$ consists of an expression e, enabled events μ, disallowed calls ν, and a continuation k. A continuation k can be the top-level continuation $\bullet$ or a continuation for returning to the body of a `let` expression, which are standard. Continuations are also used to record the active thunk via κ and $k \gg \kappa$ corresponding to the run-time stack of activation records. These continuation forms record the active thunk and are for defining messages and the app-framework interface in Section 3.3. Since events occur non-deterministically and return to the main event loop, it is reasonable to assume that a state σ should also include a heap, and the expression language should have heap-manipulating operations through which events communicate. We do not, however, formalize heap operations since they are standard.

We define an operational semantics in terms of the judgment form $\sigma \longrightarrow \sigma'$ for a small-step transition relation. In Figure 7.b, we show the inference rules defining the reduction steps related to enabling-disabling, disallowing-allowing, invoking, creating, and finally forcing thunks. The rules follow closely the informal semantics discussed in Section 3.1. Observe that Enable and Allow both permit a thunk to be forced, and Disable and Disallow remove the permission to be forced for a thunk. The difference between Enable and Disable versus Allow and Disallow is that the former pair modifies the enabled events μ, while the latter touches the disallowed calls ν.

The Event rule says that when the expression is a value v and the continuation is the top-level continuation $\bullet$, then a thunk is non-deterministically chosen from the enabled events μ to force. Observe that an enabled event remains enabled after an Event reduction, hence λ_{life} can model both events that do not self-disable (e.g., the `Click` event from Section 2) and those that are self-disabling (e.g., the `Create` event). The Invoke rule has a similar effect, but it checks that the given thunk is not disallowed in ν before forcing. The InvokeDisallowed rule states that a disallowed thunk terminates the program in the `bad` state. And the Bind rule simply states that thunks are created by binding an actual argument to a function value.

The Force rule implements the "actual application" that reduces to the function body e' with the argument v' substituted for the formal x' and the thunk substituted for the identifier `thk`, that is, $[\kappa/\texttt{thk}][v'/x']e'$. To record the stack of activations, we push the forced thunk κ on the continuation (via $k \gg \kappa$). The Return and Finish rules simply state that the recorded thunk κ frames are popped on return from a Force and Event, respectively. The Return rule returns to the caller via the continuation k, while the Finish rule returns to the top-level event loop $\bullet$. The last line with the Let and Continue rules describe, in a standard way, evaluating let-binding.

A program e violates the event-driven protocol if it ends in the `bad` state from the initial state $\langle e, \cdot, \cdot, \bullet\rangle$.

▶ **Example 3** (Asserting a Protocol Property.)**.** The no-`execute`-call-on-already-executing-*`AsyncTask`* protocol can be captured by a **`disallow`**. We let `execute` be a framework function (i.e., tagged with **`fwk`**) that takes an *`AsyncTask`* `t`.

```
let execute = (t =>fwk disallow thk; ... let h = bind handlePostExecute t in enable h)
```

The `execute` function first disallows itself (via **`disallow thk`**) and does some work (via $\ldots$) before enabling the `handlePostExecute` event handler (writing $e_1; e_2$ as syntactic sugar for sequencing). The **`disallow thk`** asserts that this thunk cannot be forced again – doing so would result in a protocol violation (i.e., the bad state).

In contrast to an event-driven framework implementation, the state of a λ_{life} program does not have a queue. As we see here, a queue is an implementation detail not relevant for capturing event-driven programming protocols. Instead, λ_{life} models the external environment, such as, user interactions, by the non-deterministic selection of an enabled event.

3.3 Messages, Observable Traces, and the App-Framework Interface

To minimally capture how a program is composed of separate framework and app code, we add some simple syntactic restrictions to λ_{life} programs. Function values λ tagged with the **`fwk`** are framework code and the **`app`** tag labels app code. We express a framework implementation $\langle \mathbf{Fun}_{\mathtt{fwk}}, \lambda_{\mathsf{init}} \rangle$ with a finite set of framework functions $\mathbf{Fun}_{\mathtt{fwk}}$ and an initialization function $\lambda_{\mathsf{init}} \in \mathbf{Fun}_{\mathtt{fwk}}$. A program e uses the framework implementation if it first invokes the function λ_{init}, and all the functions labeled as **`fwk`** in e are from $\mathbf{Fun}_{\mathtt{fwk}}$.

In a typical, real-world framework implementation, the framework implicitly defines the application-programming protocol with internal state to check for protocol violations. The Enable, Disable, Allow, and Disallow transitions make explicit the event-driven protocol specification in λ_{life}. Thus, it is straightforward to capture that framework-defined protocols by syntactically prohibiting the app from using **`enable`** κ, **`disable`** κ, **`allow`** κ, and **`disallow`** κ. Again, the enabled-event store μ and the disallowed-call store ν in λ_{life} can be seen as making explicit the implicit internal state of event-driven frameworks that define their application-programming protocols.

The app interacts with the framework only by "exchanging messages." The app-framework dialogue diagrams from Figures 4.a, 4.b, and 5.b depicts the notion of messages as arrows back-and-forth between the framework and the app. The framework invokes callbacks and returns from callins (the arrows from left to right), while the app invokes callins and returns from callbacks (the arrows from right to left). To formalize this dialogue, we label the observable transitions in the judgment form and define an *observable trace* – a trace formed only by these observable messages. Being internal to the framework, the Enable, Disable, Allow, and Disallow transitions are hidden, or unobservable, to the app.

In Figure 8, we define the judgment form $\sigma \xrightarrow[m]{} \sigma'$, which instruments our small-step transition relation $\sigma \longrightarrow \sigma'$ with message m. Recall from Section 2 that we define a callback as an invocation that transitions from framework to app code and a callin as an invocation from app to framework code. In λ_{life}, this definition is captured crisply by the execution context k in which a thunk is forced. In particular, we say that a thunk κ is a callback invocation $\mathsf{cb}\ \kappa$ if the underlying callee function is an app function (package **`app`**), and it is called from a framework function (package **`fwk`**) as in rule ForceCallback. The $\mathrm{thk}(\cdot)$ function inspects the continuation for the running, caller thunk. The $\mathrm{pkg}(\cdot)$ function gets the package of the running thunk.

Analogously, a thunk κ is a callin $\mathsf{ci}\ \kappa$ if the callee function is in the **`fwk`** package, and the caller thunk is in the **`app`** package via rule ForceCallin.

$$\text{back-messages}\quad m^{\mathbf{bk}} \in \Sigma^{\mathbf{bk}} ::= \mathsf{cb}\ \kappa \mid v = \mathsf{ciret}\ \kappa \qquad \text{in-messages}\quad m^{\mathbf{in}} \in \Sigma^{\mathbf{in}} ::= \mathsf{ci}\ \kappa \mid v = \mathsf{cbret}\ \kappa$$

$$\text{messages}\quad m \in \Sigma ::= m^{\mathbf{bk}} \mid m^{\mathbf{in}} \mid \mathsf{dis}\ m^{\mathbf{in}} \mid \epsilon \qquad \text{observable traces}\quad \omega \in \Sigma^{*} ::= \epsilon \mid \omega m$$

$$\boxed{\sigma \xrightarrow[m]{} \sigma'}$$

$$\textsc{ForceCallback}\ \frac{(x' \texttt{=>}_{\texttt{app}}\ e')[v'] = \kappa \qquad \texttt{fwk} = \mathrm{pkg}(k)}{\langle \texttt{force}\ \kappa, \mu, \nu, k\rangle \xrightarrow[\mathsf{cb}\ \kappa]{} \langle [\kappa/\texttt{thk}][v'/x']e', \mu, \nu, k \gg \kappa\rangle}$$

$$\textsc{ForceCallin}\ \frac{(x' \texttt{=>}_{\texttt{fwk}}\ e')[v'] = \kappa \qquad \texttt{app} = \mathrm{pkg}(k)}{\langle \texttt{force}\ \kappa, \mu, \nu, k\rangle \xrightarrow[\mathsf{ci}\ \kappa]{} \langle [\kappa/\texttt{thk}][v'/x']e', \mu, \nu, k \gg \kappa\rangle}$$

$$\textsc{ReturnCallin}\ \frac{(x' \texttt{=>}_{\texttt{fwk}}\ e')[v'] = \kappa \qquad \texttt{app} = \mathrm{pkg}(k)}{\langle v, \mu, \nu, k \gg \kappa\rangle \xrightarrow[v = \mathsf{ciret}\ \kappa]{} \langle v, \mu, \nu, k\rangle}$$

$$\textsc{ReturnCallback}\ \frac{(x' \texttt{=>}_{\texttt{app}}\ e')[v'] = \kappa \qquad \texttt{fwk} = \mathrm{pkg}(k)}{\langle v, \mu, \nu, k \gg \kappa\rangle \xrightarrow[v = \mathsf{cbret}\ \kappa]{} \langle v, \mu, \nu, k\rangle}$$

$$\textsc{InvokeDisallowed}\ \frac{\kappa \in \nu}{\langle \texttt{invoke}\ \kappa, \mu, \nu, k\rangle \xrightarrow[\mathsf{dis\ ci}\ \kappa]{} \mathsf{bad}}$$

$$\mathrm{thk}(\kappa) \overset{\text{def}}{=} \mathrm{thk}(k \gg \kappa) \overset{\text{def}}{=} \kappa \qquad \mathrm{thk}(k \triangleright x.e) \overset{\text{def}}{=} \mathrm{thk}(k) \qquad \mathrm{pkg}(k) \overset{\text{def}}{=} g \text{ if } (x \texttt{=>}_{g}\ e)[v] = \mathrm{thk}(k)$$

Figure 8 The instrumented transition relation $\sigma \xrightarrow[m]{} \sigma'$ defines the app-framework interface and observing the event-driven protocol.

▶ **Example 4** (Observing a Callback). Letting `handlePostExecute` be a framework function (i.e., in package **`fwk`**) and *`onPostExecute`* be an **`app`** function, the observable transition from the framework to the app defines the forcing of `cb` as a callback:

```
let onPostExecute = (t =>app ...) in
let handlePostExecute = (t =>fwk let cb = bind onPostExecute t in invoke cb) in
```

In the above, we focused on the transition back-and-forth between framework and app code via calls. Returning from calls can also be seen as a "message exchange" with a return from a callin as another kind of *back-message* going from framework code to app code (left-to-right in the figures from Section 2). We write a callin-return back-message $v = \mathsf{ciret}\ \kappa$ indicating the returning thunk κ with return value v. Likewise, a return from a callback is another kind *in-message* going from app code to framework code (right-to-left). We instrument returns in a similar way to forcings with the return back-message with ReturnCallin and the return in-message with ReturnCallback.

Finally to make explicit protocol violations, we instrument the InvokeDisallowed rule to record the disallowed-callin invocations. These rules replace the corresponding rules Force, Return, and InvokeDisallowed from Figure 7.b. For replacing the Force and Return rules, we elide two rules, one for each, where there is no switch in packages (i.e., $g' = \mathrm{pkg}(k)$ where g' is the package of the callee message). These "uninteresting" rules and the remaining rules defining the original transition relation $\sigma \longrightarrow \sigma'$ not discussed here are simply copied over with an empty message label ϵ.

Observable Traces and Dynamic-Analysis Instrumentation. As described above, the app-framework interface is defined by the possible messages that can exchanged where messages consist of callback-callin invocations and their returns. A possible app-framework interaction is thus a trace of such observable messages.

▶ **Definition 5** (App-Framework Interactions as Observable Traces). *Let* $\mathrm{paths}(e)$ *be the path semantics of* λ_{life} *expressions* e *that collects the finite sequences of alternating state-transition-state* $\sigma m \sigma'$ *triples according to the instrumented transition relation* $\sigma \xrightarrow[m]{} \sigma'$*. Then, an* observable trace *is a finite sequence of messages* $\omega\colon m_1 \dots m_n$ *obtained from a path by dropping the intermediate states and keeping the non-*ϵ *messages. We write* $[\![e]\!]$ *for the set of the observable traces obtained from the set of paths,* $\mathrm{paths}(e)$*, of an expression* e*.*

states $\hat{\sigma} ::= \langle \hat{\mu}, \hat{\nu}, \omega \rangle \mid \mathsf{bad}\ \omega$ permitted-back $\hat{\mu} ::= \cdot \mid \hat{\mu}; m^{\mathbf{bk}}$ prohibited-in $\hat{\nu} ::= \cdot \mid \hat{\nu}; m^{\mathbf{in}}$

$\boxed{\hat{\sigma} \longrightarrow \hat{\sigma}'}$

PermittedBack
$$\frac{m^{\mathbf{bk}} \in \hat{\mu} \quad \omega' = \omega m^{\mathbf{bk}} \qquad \hat{\mu}' = \mathrm{upd}^{\mathbf{bk}}_{\boldsymbol{S}}(\omega', \hat{\mu}) \quad \hat{\nu}' = \mathrm{upd}^{\mathbf{in}}_{\boldsymbol{S}}(\omega', \hat{\nu})}{\langle \hat{\mu}, \hat{\nu}, \omega \rangle \longrightarrow \langle \hat{\mu}', \hat{\nu}', \omega' \rangle}$$

ProhibitedIn
$$\frac{m^{\mathbf{in}} \in \hat{\nu} \qquad \omega' = \omega(\mathsf{dis}\ m^{\mathbf{in}})}{\langle \hat{\mu}, \hat{\nu}, \omega \rangle \longrightarrow \mathsf{bad}\ \omega'}$$

PermittedIn
$$\frac{m^{\mathbf{in}} \notin \hat{\nu} \quad \omega' = \omega m^{\mathbf{in}} \qquad \hat{\mu}' = \mathrm{upd}^{\mathbf{bk}}_{\boldsymbol{S}}(\omega', \hat{\mu}) \quad \hat{\nu}' = \mathrm{upd}^{\mathbf{in}}_{\boldsymbol{S}}(\omega', \hat{\nu})}{\langle \hat{\mu}, \hat{\nu}, \omega \rangle \longrightarrow \langle \hat{\mu}', \hat{\nu}', \omega' \rangle}$$

Figure 9 This transition system defines an abstraction of the framework-internal state consistent with an observable trace ω with respect to a framework abstraction $\boldsymbol{S}$. The abstract state $\hat{\sigma}$ contains a store of permitted back-messages $\hat{\mu}$ and a store of prohibited in-messages $\hat{\nu}$, corresponding to an abstraction of enabled events and disallowed calls, respectively. The meaning of the framework abstraction $\boldsymbol{S}$ is captured by the store-update functions $\mathrm{upd}^{\mathbf{bk}}_{\boldsymbol{S}}$ and $\mathrm{upd}^{\mathbf{in}}_{\boldsymbol{S}}$, which determine how an abstract store changes on a new message.

An observable trace ω *violates* the event-driven application-programming protocol if ω ends with a disallowed dis message.

These definitions yield a design for a dynamic-analysis instrumentation that observes app-framework interactions. The trace recording in Verivita obtains observable traces ω like the app-framework dialogue diagrams in Section 2 by following the instrumented semantics $\sigma \underset{m}{\longrightarrow} \sigma'$. Verivita maintains a stack similar to the continuation k to emit the messages corresponding to the forcings and returns of callbacks and callins, and it emits disallowed $\mathsf{dis}\ \kappa$ messages by observing the exceptions thrown by the framework.

4 Specifying Protocols and Modeling Callback Control Flow

Using λ_{life} as a concrete semantic foundation, we first formalize an abstraction of event-driven programs composed of separate app and framework code with respect to what is observable at the app-framework interface. This abstract transition system captures the possible enabled-event and disallowed-call stores internal to the framework that are consistent with observable traces, essentially defining a family of lifestate framework abstractions. Then, we instantiate this definition for a specific lifestate language that both specifies event-driven application-programming protocols and models callback control flow.

The main point in these definitions is that lifestate modeling of callback control flow can only depend on what is observable at the app-framework interface. Furthermore, the concrete semantic foundation given by λ_{life} leads to a careful definition of soundness and precision and a basis for model validation and predictive-trace verification (Section 5).

Abstracting Framework-Internal State by Observing Messages. In Figure 9, we define the transition system that abstracts the framework-internal state consistent with an observable trace ω. An abstract state $\langle \hat{\mu}, \hat{\nu}, \omega \rangle$ contains a store of *permitted back-messages* $\hat{\mu}$ and a store of *prohibited in-messages* $\hat{\nu}$. What the transition system captures are the possible traces consistent with iteratively applying a framework abstraction $\boldsymbol{S}$ to the current abstract state: it performs a transition with a back-message $m^{\mathbf{bk}}$ only if $m^{\mathbf{bk}}$ is permitted $m^{\mathbf{bk}} \in \hat{\mu}$, and a transition with an in-message $m^{\mathbf{in}}$ only if $m^{\mathbf{in}}$ is not prohibited $m^{\mathbf{in}} \notin \hat{\nu}$. The trace ω in an abstract state saves the history of messages observed so far. In the most general setting for modeling the event-driven framework, the transition system can update the stores $\hat{\mu}$ and $\hat{\nu}$ as a function of the history of the observed messages ω. These updates are formalized with

$$\text{parametrized messages} \quad \boldsymbol{m} ::= \mathsf{cb}\ \lambda[\boldsymbol{p}] \mid \boldsymbol{p}' = \mathsf{ciret}\ \lambda[\boldsymbol{p}] \mid \mathsf{ci}\ \lambda[\boldsymbol{p}] \mid \boldsymbol{p}' = \mathsf{cbret}\ \lambda[\boldsymbol{p}]$$

$$\begin{array}{ll} \text{lifestate rules} & \boldsymbol{s} ::= \boldsymbol{r} \rightarrow \boldsymbol{m} \mid \boldsymbol{r} \nrightarrow \boldsymbol{m} \\ \text{lifestate abstractions} & \boldsymbol{S} ::= \cdot \mid \boldsymbol{s}\boldsymbol{S} \\ \text{trace matchers: regular expressions of parametrized messages} & \boldsymbol{r} \end{array}$$

$$\text{symbolic variables} \quad \ell \in \mathbf{SVar} \qquad \text{parameters} \quad \boldsymbol{p} \in \mathbf{SVar} \cup \mathbf{Val} \qquad \text{binding maps} \quad \theta ::= \cdot \mid \theta, \ell \mapsto v$$

(10.a) A lifestate abstraction is a set of rules that permits ($\rightarrow$) or prohibits ($\nrightarrow$) parametrized messages $\boldsymbol{m}$.

$$\begin{aligned} \mathrm{upd}^{\mathbf{bk}}_{\boldsymbol{S}}(\omega, \hat{\mu}) &\stackrel{\text{def}}{=} \left\{ m^{\mathbf{bk}} \;\middle|\; \mathrm{consistent}_{\boldsymbol{S}}(\omega) \wedge \left(\neg\,\mathrm{prohibit}_{\boldsymbol{S}}(\omega, m^{\mathbf{bk}}) \wedge (\mathrm{permit}_{\boldsymbol{S}}(\omega, m^{\mathbf{bk}}) \vee m^{\mathbf{bk}} \in \hat{\mu})\right) \right\} \\ \mathrm{upd}^{\mathbf{in}}_{\boldsymbol{S}}(\omega, \hat{\nu}) &\stackrel{\text{def}}{=} \left\{ m^{\mathbf{in}} \;\middle|\; \mathrm{consistent}_{\boldsymbol{S}}(\omega) \rightarrow \left(\neg\,\mathrm{permit}_{\boldsymbol{S}}(\omega, m^{\mathbf{in}}) \wedge (\mathrm{prohibit}_{\boldsymbol{S}}(\omega, m^{\mathbf{in}}) \vee m^{\mathbf{in}} \in \hat{\nu})\right) \right\} \\ \mathrm{permit}_{\boldsymbol{S}}(\omega, m) &\stackrel{\text{def}}{=} \exists \boldsymbol{r} \rightarrow \boldsymbol{m} \in \boldsymbol{S}, \exists \theta, (\omega, \theta \models \boldsymbol{r}) \wedge \theta(\boldsymbol{m}) = m \\ \mathrm{prohibit}_{\boldsymbol{S}}(\omega, m) &\stackrel{\text{def}}{=} \exists \boldsymbol{r} \nrightarrow \boldsymbol{m} \in \boldsymbol{S}, \exists \theta, (\omega, \theta \models \boldsymbol{r}) \wedge \theta(\boldsymbol{m}) = m \\ \mathrm{consistent}_{\boldsymbol{S}}(\omega) &\stackrel{\text{def}}{=} \forall m \in \Sigma, (\mathrm{permit}_{\boldsymbol{S}}(\omega, m) \leftrightarrow \neg\,\mathrm{prohibit}_{\boldsymbol{S}}(\omega, m)) \end{aligned}$$

$$\hat{\mu}^{\text{init}}_{\boldsymbol{S}} \stackrel{\text{def}}{=} \mathrm{upd}^{\mathbf{bk}}_{\boldsymbol{S}}(\epsilon, \Sigma^{\mathbf{bk}}) \qquad\qquad \hat{\nu}^{\text{init}}_{\boldsymbol{S}} \stackrel{\text{def}}{=} \mathrm{upd}^{\mathbf{in}}_{\boldsymbol{S}}(\epsilon, \emptyset)$$

(10.b) Semantics of a lifestate framework abstraction. The store-update functions $\mathrm{upd}^{\mathbf{bk}}_{\boldsymbol{S}}$ and $\mathrm{upd}^{\mathbf{in}}_{\boldsymbol{S}}$ find rules from $\boldsymbol{S}$ that match the given trace ω and update the store according to a consistent binding θ from symbolic variables to values.

Figure 10 Lifestate is a language for simultaneously specifying event-driven protocols and modeling callback control flow in terms of the observable app-framework interface.

the store-update functions $\mathrm{upd}^{\mathbf{bk}}_{\boldsymbol{S}}(\omega, \hat{\mu})$ and $\mathrm{upd}^{\mathbf{in}}_{\boldsymbol{S}}(\omega, \hat{\nu})$ that define an abstraction $\boldsymbol{S}$ of the event-driven framework describing both its application-programming protocol and its callback control flow. A framework abstraction $\boldsymbol{S}$ also defines the initial abstract state $\langle \hat{\mu}^{\text{init}}_{\boldsymbol{S}}, \hat{\nu}^{\text{init}}_{\boldsymbol{S}}, \epsilon \rangle$ that contains the initial (abstract) state of the stores of the permitted back-messages and prohibited in-messages.

The semantics $[\![\boldsymbol{S}]\!]$ of a framework abstraction $\boldsymbol{S}$ is the set of observable traces of the transition system defined in Figure 9 instantiated with $\boldsymbol{S}$. We get sequences of states from the transition relation $\hat{\sigma} \longrightarrow \hat{\sigma}'$, read the observable trace ω from the final state, and form a set of all such observable traces.

▶ **Definition 6** (Soundness of a Framework Abstraction).

(1) *A framework abstraction $\boldsymbol{S}$ is a sound abstraction of a λ_{life} program e if $[\![e]\!] \subseteq [\![\boldsymbol{S}]\!]$;*

(2) *A framework abstraction $\boldsymbol{S}$ is a sound abstraction of a framework implementation $\langle \mathbf{Fun}_{\mathtt{fwk}}, \lambda_{\mathtt{init}} \rangle$ if and only if $\boldsymbol{S}$ is sound for every possible program e that uses the framework implementation $\langle \mathbf{Fun}_{\mathtt{fwk}}, \lambda_{\mathtt{init}} \rangle$.*

The possible observable traces of a framework abstraction $\boldsymbol{S}$ is slightly richer than observable λ_{life} traces in that callback-return messages ($v = \mathsf{cbret}\ \kappa$) may also be prohibited (in addition callin-invocations $\mathsf{ci}\ \kappa$). Prohibiting callback-return messages corresponds to specifying a protocol where the app yields an invalid return value. If we desire to capture such violations at the concrete level, it is straightforward to extend λ_{life} with "return-invalid" transitions by analogy to InvokeDisallowed transitions.

Finally, if $\boldsymbol{S_1}$ and $\boldsymbol{S_2}$ are sound specifications, we say that $\boldsymbol{S_1}$ is *at least as precise* as $\boldsymbol{S_2}$ if $[\![\boldsymbol{S_1}]\!] \subseteq [\![\boldsymbol{S_2}]\!]$.

A Lifestate Abstraction. We arrive at lifestates by instantiating the framework abstraction $\boldsymbol{S}$ in a direct way as shown in Figure 10. To describe rules independent of particular programs or executions, we *parametrize messages* with symbolic variables $\ell \in \mathbf{SVar}$. The definition of the parametrized messages $\boldsymbol{m}$ is parallel to the non-parameterized version but using parameters instead of simply concrete values v. We call a message $\boldsymbol{m}$ *ground* when it does not have symbolic variables (from $\mathbf{SVar}$), and we distinguish the ground and parameterized messages by using normal m and bold $\boldsymbol{m}$ fonts, respectively. For example, the parametrized callback-invocation message $\mathsf{cb}\ \lambda[\ell]$ specifies that a callback function λ is invoked with an arbitrary value from $\mathbf{Val}$. The variable ℓ can be used across several messages in a rule, expressing that multiple messages are invoked with, or return, the same value.

A lifestate abstraction $\boldsymbol{S}$ is a set of rules, and a *rule* consists of trace matcher $\boldsymbol{r}$ that when matched either *permits* ($\rightarrow$ operator) or *prohibits* ($\not\twoheadrightarrow$ operator) a parametrized message $\boldsymbol{m}$. As just one possible choice for the matcher $\boldsymbol{r}$, we consider $\boldsymbol{r}$ to be a regular expression where the symbols of the alphabet are parametrized messages $\boldsymbol{m}$. In matching a trace ω to a regular expression of parametrized messages, we obtain a *binding* θ that maps symbolic variables from the parametrized messages to the concrete values from the trace. Given a binding θ and a message $\boldsymbol{m}$, we write $\theta(\boldsymbol{m})$ to denote the message $\boldsymbol{m'}$ obtained by replacing each symbolic variable ℓ in $\boldsymbol{m}$ with $\theta(\ell)$ if defined.

The semantics of lifestates is given by a choice of store-update functions $\mathrm{upd}^{\mathsf{bk}}_{\boldsymbol{S}}$ and $\mathrm{upd}^{\mathsf{in}}_{\boldsymbol{S}}$ in Figure 10.b and the abstract transition relation $\hat{\sigma} \longrightarrow \hat{\sigma}'$ defined previously in Section 4. The store-update functions work intuitively by matching the given trace ω against the matchers $\boldsymbol{r}$ amongst the rules in $\boldsymbol{S}$ and then updating the store according to the matching rules $\{\boldsymbol{s_1}, \ldots, \boldsymbol{s_n}\} \subseteq \boldsymbol{S}$.

To describe the store-update functions in Figure 10.b, we write $\omega, \theta \models \boldsymbol{r}$ to express that a trace ω and a binding θ satisfy a regular expression $\boldsymbol{r}$. The definition of this semantic relation is standard, except for parametrized messages $\boldsymbol{m}$. Here, we explain this interesting case for when the trace ω and the binding θ satisfy the regular expression $\boldsymbol{m}$ (i.e., $\omega, \theta \models \boldsymbol{m}$):

$$\omega, \theta \models \boldsymbol{m} \quad \text{iff} \quad \omega = m \text{ and } \theta(\boldsymbol{m}) = m \text{ for some ground message } m$$

A necessary condition for $\omega, \theta \models \boldsymbol{m}$ is, for example, that θ must assign a value to all the variables in $\boldsymbol{m}$, to get a ground message, and the message must be equal to the trace ω. Note that, if there is no such ground message for $\boldsymbol{m}$ with the binding θ, then $\omega, \theta \not\models \boldsymbol{m}$. The full semantics of matching parametrized regular expressions is given in the extended version [33].

Now, the function $\mathrm{upd}^{\mathsf{bk}}_{\boldsymbol{S}}(\omega, \hat{\mu})$ captures how the state of the permitted back-messages store $\hat{\mu}$ changes according to the rules $\boldsymbol{S}$. As a somewhat technical point, a back-message can only be permitted if the rules $\boldsymbol{S}$ are *consistent* with respect to the given trace ω (i.e., $\mathrm{consistent}_{\boldsymbol{S}}(\omega)$). The $\mathrm{consistent}_{\boldsymbol{S}}(\omega)$ predicate holds iff there are no rules that permits and prohibits m for the same message m and trace ω. Then, if the predicate $\mathrm{consistent}_{\boldsymbol{S}}(\omega)$ is true, the back-message m^{bk} must not be prohibited given the trace ω (i.e., $\neg\,\mathrm{prohibit}_{\boldsymbol{S}}(\omega, m^{\mathsf{bk}})$). Finally, if back-message m^{bk} is not prohibited, either it is permitted by a specification for this trace ω (i.e., $\mathrm{permit}_{\boldsymbol{S}}(\omega, m^{\mathsf{bk}})$) or it was already permitted in the current store $\hat{\mu}$ (i.e., $m^{\mathsf{bk}} \in \hat{\mu}$). The function $\mathrm{upd}^{\mathsf{in}}_{\boldsymbol{S}}(\omega, \hat{\nu})$ is similar, but it is defined for the prohibited in-messages store $\hat{\nu}$. An in-message m^{in} is prohibited first if the rules are not consistent. Then, if the rules are consistent, the in-message m^{in} must not be permitted by this trace, and either it is prohibited by a rule for this trace or the in-message was already prohibited in the current store $\hat{\nu}$. The auxiliary predicates $\mathrm{permit}_{\boldsymbol{S}}(\omega, m)$ and $\mathrm{prohibit}_{\boldsymbol{S}}(\omega, m)$ formally capture these conditions. The $\mathrm{permit}_{\boldsymbol{S}}(\omega, m)$ predicate is true iff there is a rule $\boldsymbol{r} \rightarrow \boldsymbol{m}$ in the specification $\boldsymbol{S}$ that permits a message $\boldsymbol{m}$ and a binding θ, such that the trace and the binding satisfy the regular expression ($\omega, \theta \models \boldsymbol{r}$), and the ground message permitted by the rule $\theta(\boldsymbol{m})$ is m. The $\mathrm{prohibit}_{\boldsymbol{S}}(\omega, m)$ predicate is analogous but for prohibit rules.

A key point is that the store-update functions $\mathrm{upd}^{\mathsf{bk}}_{\boldsymbol{S}}(\omega, \hat{\mu})$ and $\mathrm{upd}^{\mathsf{in}}_{\boldsymbol{S}}(\omega, \hat{\nu})$ are defined only in terms of what is observable at the app-framework interface ω and stores of permitted back-messages $\hat{\mu}$ and prohibited in-messages $\hat{\nu}$. Lifestate abstractions $\boldsymbol{S}$ do not depend on framework or app expressions e, nor framework-internal state.

5 Dynamic Reasoning with Lifestates

Lifestates are precise and detailed abstractions of event-driven frameworks that simultaneously specify the protocol that the app should observe and the callback control-flow assumptions that an app can assume about the framework. The formal development of lifestates in the above offers a clear approach for *model validation* and *predictive-trace verification*. In this section, we define the model validation and verification problem and provide an intuition of their algorithms using the formal development in the previous sections. For completeness of presentation, we provide further details in the extended version [33].

Validating Lifestate Specifications. As documentation in a real framework implementation like Android is incomplete and ambiguous, it is critical that framework abstractions have a mechanism to validate candidate rules – in a manner independent of, say, a downstream static or dynamic analysis.

We say that a specification $\boldsymbol{S}$ is *valid* for an observable trace ω if $\omega \in [\![\boldsymbol{S}]\!]$. If a specification $\boldsymbol{S}$ is not valid for a trace ω from a program e, then $\boldsymbol{S}$ is not a sound abstraction of e.

We can then describe an algorithm that checks if $\boldsymbol{S}$ is a valid specification for a trace ω with a reduction to a model checking problem. Lifestate rules specify the behavior of an unbounded number of objects through the use of symbolic variables $\ell \in \mathbf{SVar}$ that are implicitly universally quantified in the language and hence describe an unbounded number of messages. However, as an observable trace ω has a finite number of ground messages, the set of messages that we can use to instantiate the quantifiers is also finite. Thus, the validation algorithm first "removes" the universal quantifier with the *grounding* process that transforms the lifestate abstraction $\boldsymbol{S}$ to a *ground abstraction* S containing only ground rules.

The language $[\![S]\!]$ of a ground specification S can be represented with a finite transition system since the set of messages in S is finite, and lifestate rules are defined using regular expressions. We then pose the validation problem as a model checking problem that we solve using off-the-shelf symbolic model checking tools [12]. The transition system that we check is the parallel composition (i.e., the intersection of the languages of transition systems) of the transition system that accepts only the trace ω and the transition system $\hat{\sigma} \longrightarrow \hat{\sigma}'$ parametrized by the grounded lifestate abstraction S. The lifestate abstraction S is valid if and only if the composed transition system reaches the last state of the trace ω.

Dynamic Lifestate Verification. Because of the previous sections building up to lifestate validation, the formulation of the dynamic verification is relatively straightforward and offers a means to evaluate the expressiveness of lifestate specification.

We define the set of sub-traces of a trace $\omega = \omega_1 \ldots \omega_l$ as $Sub_\omega \stackrel{\text{def}}{=} \{\omega_1, \ldots, \omega_l\}$, where $\omega' \in Sub_\omega$ if ω' is a substring of ω that represents the entire execution of a callback directly invoked by an event handler. We consider the set $[\![(\omega_1 + \ldots + \omega_l)^*]\!]$ of all the traces obtained by repeating the elements in Sub_ω zero-or-more times and $\Omega_{\omega,e} \stackrel{\text{def}}{=} [\![e]\!] \cap [\![(\omega_1 + \ldots + \omega_l)^*]\!]$ its intersection with the traces of the λ_{life} program e.

Given an observable trace ω of the program e (i.e., $\omega \in [\![e]\!]$), the *dynamic verification problem* consists of proving the absence of a trace $\omega' \in \Omega_{\omega,e}$ that violates the application-programming protocol. Since we cannot know the set of traces $[\![e]\!]$ for a λ_{life} program e (i.e., the set of traces for the app composed with the framework implementation), we cannot solve the dynamic verification problem directly. Instead, we solve an abstract version of the problem, where we use a lifestate specification $\boldsymbol{S}$ to abstract the framework implementation $\langle \mathbf{Fun}_{\mathtt{fwk}}, \lambda_{\mathbf{init}} \rangle$. Let $\Omega_{\omega,\boldsymbol{S}} \stackrel{\text{def}}{=} [\![\boldsymbol{S}]\!] \cap [\![(\omega_1 + \ldots + \omega_l)^*]\!]$ be the set of repetitions of the trace ω that can be seen in the app-framework interface abstraction defined by $\boldsymbol{S}$.

Given a trace $\omega \in [\![e]\!]$ and a sound specification $\boldsymbol{S}$, the *abstract dynamic verification problem* consists of proving the absence of a trace $\omega' \in \Omega_{\omega,\boldsymbol{S}}$ that violates the application-programming protocol. If we do not find any protocol violation using a specification $\boldsymbol{S}$, then there are no violations in the possible repetitions of the concrete trace ω. Observe that the key verification challenge is getting a precise enough framework abstraction $\boldsymbol{S}$ that sufficiently restricts the possible repetitions of the concrete trace ω.

We reduce the abstract dynamic verification problem to a model checking problem in a similar way to validation: we first generate the *ground model* S from the lifestate model $\boldsymbol{S}$ and the trace ω. Then, we construct the transition system that only generates traces in the set $\Omega_{\omega,\boldsymbol{S}}$ by composing the transition system obtained from the ground specification S and the automaton accepting words in $[\![(\omega_1 + \ldots + \omega_l)^*]\!]$. This transition system satisfies a safety property iff there is no trace $\omega \in \Omega_{\omega,\boldsymbol{S}}$ that violates the protocol.

6 Empirical Evaluation

We implement our approach for Android in the Verivita tool that

(i) instruments an Android app to record observable traces,

(ii) validates a lifestate model for soundness against a corpus of traces, and

(iii) assesses the precision of a lifestate model with dynamic verification.

We use the following research questions to demonstrate that lifestate is an effective language to model event-driven protocols, and validation is a crucial step to avoid unsoundness.

RQ1 *Lifestate Precision.* Is the lifestate language adequate to model the callback control flow of Android? The paper hypothesizes that carefully capturing the app-framework interface is necessary to obtain precise protocol verification results.

RQ2 *Lifestate Generality.* Do lifestate models generalize across apps? We want to see if a lifestate model is still precise when used on a trace from a new, previously unseen app.

RQ3 *Model Validation.* Is validation of callback control-flow models with concrete traces necessary to develop *sound* models? We expect to witness unsoundnesses in existing (and not validated) callback control-flow models and that validation is a crucial tool to get sound models.

Additionally, we considered the feasibility of continuous model validation. The bottom line is that we could validate 96% of the traces within a 6 minute time budget; we discuss these results further in the extended version [33].

***RQ1*: Lifestate Precision.** The bottom line of Table 1 is that lifestate modeling is essential to improve the percentage of verified traces to 83% – compared to 57% for lifecycle++ and 27% for lifecycle modeling.

Methodology. We collect execution traces from Android apps and compare the precision obtained verifying protocol violations with four different callback control-flow models. The first three models are expressed using different subsets of the lifestate language. The *top* model is the least precise (but clearly sound) model where any callback can happen at all times, like in the Automaton 6.a in Section 2. The *lifecycle* model represents the most precise callback control-flow model that we can express only using back-messages, like in Automaton 6.b. The *lifestate* model uses the full lifestate language, and hence also in-messages like in the Automaton 6.d, to change the currently permitted back-messages. It represents the most precise model that we can represent with lifestate. To faithfully compare the precision of the formalisms, we improved the precision of the lifecycle and lifestate models minimizing the false alarms from verification. And at the same time, we continuously run model validation to avoid unsoundnesses, as we discuss below in *RQ3*. As a result of this process, we modeled the behavior of several commonly-used Android classes, including `Activity`, `Fragment`, `AsyncTask`, `CountdownTimer`, `View`, `PopupMenu`, `ListView`, and `Toolbar` and their subclasses. Excluding similar rules for subclasses, this process resulted in a total of 167 lifestate rules.

We further compare with an instance of a *lifecycle++* model, which refines component lifecycles with callbacks from other Android objects. Our model is a re-implementation of the model used in FlowDroid [5] that considers the lifecycle for the UI components (i.e., `Activity` and `Fragment`) and bounds the execution of a pre-defined list of callback methods in the active state of the `Activity` lifecycle, similarly to the example we show in Figure 2. We made a best effort attempt to faithfully replicate the FlowDroid model (and discuss how we did so in the extended version [33]).

To find error-prone protocols, we selected *sensitive callins*, shown in the first column of Table 1, that frequently occur as issues on GitHub and StackOverflow [16, 41, 3, 36, 47, 46]. We then specify the lifestate rules to allow and disallow the sensitive callins.

To create a realistic trace corpus for *RQ1*, we selected five apps by consulting Android user groups to find those that extensively use Android UI objects, are not overly simple (e.g., student-developed or sample-projects apps), and use at least one of the sensitive callins. To obtain realistic interaction traces, we recorded manual interactions from a non-author user who had no prior knowledge of the internals of the app. The user used each app 10 times for 5 minutes (on an x86 Android emulator running Android 6.0) – obtaining a set of 50 interaction traces. With this trace-gathering process, we exercise a wide range of behaviors of Android UI objects that drives the callback control-flow modeling.

To evaluate the necessity and sufficiency of lifestate, we compare the verified rates (the total number of verified traces over the total number of verifiable traces) obtained using each callback control-flow model. We further measure the verification run time to evaluate the trade-off between the expressiveness of the models and the feasibility of verification.

Table 1 Precision of callback control-flow models. The *sensitive callin* column lists protocol properties by the callin that crashes the app when invoked in a bad state. We collect a total of 50 traces from 5 applications with no crashes. The *sensitive* column lists the number of traces where the application invokes a sensitive callin. To provide a baseline for the precision of a model, we count the number of traces without a manually-confirmed real bug in the *verifiable* column. There are four columns labeled *verified* showing the number and percentage of verifiable traces proved correct using different callback control-flow models. The *lifestate* columns capture our contribution. The *lifecycle++* columns capture the current practice for modeling the Android framework. The *bad* column lists the number of missed buggy traces and is discussed further in *RQ2*.

properties	non-crashing traces		callback control-flow models								
			top		lifecycle		lifestate		lifecycle++		
sensitive callin	sensitive (n)	verifiable (n)	verified (n)	(%)	verified (n)	(%)	verified (n)	(%)	verified (n)	(%)	bad (n)
`AlertDialog`											
dismiss	16	6	0	0	0	0	6	100	6	100	0
show	43	34	17	50	17	50	28	82	24	71	0
`AsyncTask`											
execute	4	4	0	0	4	100	4	100	0	0	0
`Fragment`											
getResources	10	10	0	0	0	0	10	100	4	40	0
getString	10	10	0	0	0	0	2	20	0	0	0
setArguments	19	19	1	5	1	5	19	100	13	68	0
total	102	83	18	22	22	27	69	83	47	57	0

Table 2 The table shows the precision results for the 1577 non-crashing traces that contained a sensitive callins from a total of 2202 traces that we collected from 121 distinct open source app repositories. We note that lifestate takes slightly longer than lifecycle; for this reason, lifestate performs slightly worse than lifecycle for execute. The bad column is 0 for models other than lifecycle++ because of continuous validation. Note that out of 64 total buggy traces, lifecycle++ missed 27 bugs (i.e., had a 42% false-negative rate).

properties	non-crashing traces		callback control-flow models								
			top		lifecycle		lifestate		lifecycle++		
sensitive callin	sensitive (n)	verifiable (n)	verified (n)	(%)	verified (n)	(%)	verified (n)	(%)	verified (n)	(%)	bad (n)
`AlertDialog`											
dismiss	94	59	54	92	54	92	54	92	58	98	3
show	145	144	125	87	124	86	125	87	127	88	0
`AsyncTask`											
execute	415	415	0	0	415	100	412	99	262	63	0
`Fragment`											
getResources	156	155	89	57	89	57	128	83	116	75	0
getString	220	193	124	64	124	64	134	69	131	68	24
setArguments	456	456	59	13	108	24	437	96	435	95	0
startActivity	91	91	0	0	0	0	12	13	19	21	0
total	1577	1513	451	30	914	60	1302	86	1148	76	27

Discussion. In Table 1, we show the number of verified traces and the verified rates broken down by sensitive callins and different callback control-flow models – aggregated over all apps. As stated earlier, the precision improvement with lifestate is significant, essential to get to 83% verified. We also notice that the lifecycle model is only slightly more precise than the trivial top model (27% versus 22% verified rate). Even with unsoundnesses discussed later, lifecycle++ is still worse than the lifestate model, with 57% of traces proven.

Lifestate is also expressive enough to prove most verifiable traces – making manual triage of the remaining alarms feasible. We manually examined the 14 remaining alarms with the lifestate model, and we identified two sources of imprecision:

(1) an insufficient modeling of the attachment of UI components (e.g., is a `View` in the `View` tree attached to a particular `Activity`?), resulting in 13 alarms;

(2) a single detail on how Android options are set in the app's XML, resulting in 1 alarm.

The former is not fundamental to lifestates but a modeling tradeoff where deeper attachment modeling offers diminishing returns on the verified rate while increasing the complexity of the model and verification times. The latter is an orthogonal detail for handling Android's XML processing (that allows the framework to invoke callbacks via reflection).

***RQ2*: Lifestate Generality.** The bottom line of Table 2 is that the lifestate model developed for *RQ1* as-is generalizes to provide precise results (with a verified rate of 86%) when used to verify traces from 121 previously unseen apps. This result provides evidence that lifestates capture general behaviors of the Android framework. While the lifecycle++ model verifies 76% of traces, it also misses 27 out of 64 buggy traces (i.e., has a 42% false-negative rate).

Methodology. To get a larger corpus, we cloned 121 distinct open source apps repositories from GitHub that use at least one sensitive callin (the count combines forks and clones). Then, we generated execution traces using the Android UI Exerciser Monkey [2] that interacts with the app issuing random UI events (e.g., clicks, touches). We attempted to automatically generate three traces for each app file obtained by building each app.

Discussion. From Table 2, we see that the lifestate verified rate of 86% in this larger experiment is comparable with the verified rate obtained in *RQ1*. Moreover, lifestate still improves the verified rate with respect to lifecycle, which goes from 60% to 86%, showing that the expressivity of lifestate is necessary.

Critically, the lifecycle++ model does not alarm on 42% of the traces representing real defects. That is, we saw unsoundnesses of the lifecycle++ model manifest in the protocol verification client.

The verified rate for the lifecycle model is higher in this larger corpus (60%) compared to the rate in *RQ1* (27%), and the precision improvement from the top abstraction is more substantial (60% to 30% versus 27% to 22%). This difference is perhaps to be expected when using automatically-generated traces that may have reduced coverage of app code and bias towards shallower, "less interesting" callbacks associated with application initialization instead of user interaction. In these traces, it is possible that UI elements were not exercised as frequently, which would result in more traces provable solely with the lifecycle specification. Since coverage is a known issue for the Android UI Exerciser Monkey [4]), it was critical to have some evidence on deep, manually-exercised traces as in *RQ1*.

Bug Triage. We further manually triage every remaining alarm from both *RQ1* and *RQ2*. Finding protocol usage bugs was not necessarily expected: for *RQ1*, we selected seemingly well-developed, well-tested apps to challenge verification, and for *RQ2*, we did not expect automatically generated traces to get very deep into the app (and thus deep in any protocol).

Yet from the *RQ1* triage, we found 2 buggy apps out of 5 total. These apps were Puzzles [10] and SwiftNotes [13]. Puzzles had two bugs, one related to `AlertDialog.show` and one for `AlertDialog.dismiss`. Swiftnotes has a defect related to `AlertDialog.show`.

In the *RQ2* corpus, we found 7 distinct repositories with a buggy app (out of 121 distinct repositories) from 64 buggy traces (out of 2202). We were able to reproduce bugs in 4 of the repositories and strongly suspect the other 3 to also be buggy. Three of the buggy apps invoke a method on `Fragment` that requires the `Fragment` to be attached. This buggy invocation happens within unsafe callbacks. Audiobug [51] invokes `getResources`. NextGisLogger [35] and Kistenstapeln [14] invoke `getString`. We are able to reproduce the Kistenstapeln bug.

Interestingly, one of the apps that contain a bug is Yamba [20], a tutorial app from a book on learning Android [21]. We note that the Yamba code appears as a part of three repositories where the code was copied (we only count these as one bug). The tutorial app calls `AlertDialog.dismiss` when an `AsyncTask` is finishing and hence potentially after the `Activity` object used in the `AlertDialog` is not visible anymore. We found similar defects in several actively maintained open source apps where callbacks in an `AsyncTask` object were used either to invoke `AlertDialog.show` or `AlertDialog.dismiss`. These apps included OSM Tracker [22] and Noveldroid [44]. Additionally, we found this bug in a binary library connected with the PingPlusPlus android app [38]. By examining the output of our verifier, we were able to create a test to concretely witness defects in 4 of these apps.

***RQ3*: Model Validation.** The plot in Figure 11 highlights the necessity of applying model validation: lifecycle++ based on a widely used callback control-flow model does not validate (i.e., an unsoundness is witnessed) on 58% of 2183 traces (and the validation ran out of memory for 19 out of the total 2202 traces).

Methodology. We first evaluate the need for model validation by applying our approach to lifecycle++ and quantifying its discrepancies with the real Android executions.

Our first experiment validates the lifecycle++ model on all the traces we collected (bounding each validation check to 1 hour and 4 GB of memory). We quantify the necessity of model validation collecting for each trace if the model was valid and the length of the maximum prefix of the trace that the model validates. Since there are already some known limitations in the lifecycle++ model (e.g., components interleaving), we triage the results to understand if the real cause of failure is a new mistake discovered with the validation process.

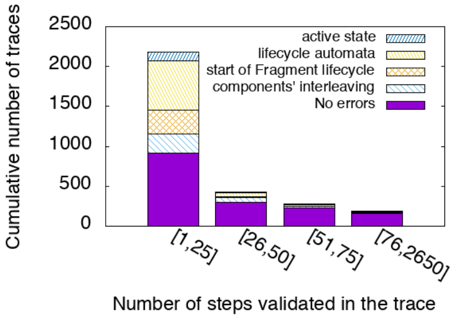


Figure 11 Results of the validation of the lifecycle++ model on all the traces. We plot the cumulative traces grouped by (intervals of) the number of steps validated. The number of traces are further divided into categories, either indicating that validation succeeded, "no errors," or the cause of failure of the validation process.

Our second experiment qualitatively evaluates the necessity of model validation to develop sound lifestate specifications. To create a sound model, we started from the empty model (without rules) and continuously applied validation to find and correct mistakes. In each iteration: we model the callback control flow for a specific Android object; we validate the current model on the entire corpus of traces (limiting each trace to one hour and 4 GB of memory); and when the model is not valid for a trace, we inspect the validation result and repair the specification. We stop when the model is valid for all the traces. We then collected the mistakes we found with automatic validation while developing the lifestate model. We describe such mistakes and discuss how we used validation to discover and fix them.

Discussion: lifecycle++ Validation. From the first bar of the plot in Figure 11, we see that the lifecycle++ model validates only 42% of the total traces, while validation fails in the remaining cases (58%). The bar shows the number of traces that we validated for at least one step, grouping them by validation status and cause of validation failure. From our manual triage, we identified 4 different broad causes for unsoundness:

i) *outside the active lifecycle*: the model prohibits the execution of a callback outside the modeled active state of the `Activity`;

ii) *wrong lifecycle automata*: the model wrongly prohibits the execution of an `Activity` or `Fragment` lifecycle callback;

iii) *wrong start of the `Fragment` lifecycle*: the model prohibits the start of the execution of the `Fragment` lifecycle;

iv) *no components interleaving*: the model prohibits the interleaved execution of callbacks from different `Activity` or `Fragment` objects.

The plot shows that the lifecycle++ model is not valid on 25% of the traces because it does not model the interleaving of components (e.g., the execution of callbacks from different `Activity` and `Fragment` objects cannot interleave) and the start of the `Fragment` lifecycle at an arbitrary point in the enclosing `Activity` object. With FlowDroid, such limitations are known and have been justified as practical choices to have feasible flow analyses [5]. But the remaining traces, 33% of the total, cannot be validated due other reasons including modeling mistakes. In particular, the FlowDroid model imprecisely captures the lifecycle automata (for both `Activity` and `Fragment`) and erroneously confines the execution of some callbacks in the active state of the lifecycle.

The other bars in the plot of Figure 11 show the number of traces we validated for more than 25, 50, and 75 steps, respectively. In the plot, we report the total number of steps in the execution traces that correspond to a callback or a callin that we either used in the lifestate or the lifecycle++ model, while we remove all the other messages. From such bars, we see that we usually detect the unsoundness of the lifecycle++ model "early" in the trace (i.e., in the first 25 steps). This result is not surprising since most of the modeling mistakes we found are related to the interaction with the lifecycle automata and can be witnessed in the first iteration of the lifecycle. We further discovered that the lifecycle++ model mostly validates shorter execution traces, showing that having sound models for real execution traces is more challenging, which we discuss further in the extended version [33].

Discussion: Catching Mistakes During Modeling. We were able to obtain a valid lifestate specification *for over 99.9%* of the traces in our corpus. That is, we were able to understand and model the objects we selected in all but two traces.

Surprisingly, we identified and fixed several mistakes in our modeling of the `Activity` and `Fragment` lifecycle that are due to undocumented Android behaviors. An example of such behavior is the effect of `Activity.finish` and `Activity.startActivity` on the callback control flow for the `onClick` callback. It is unsound to restrict the enabling of `onClick` callbacks to the active state of the `Activity` lifecycle (i.e., between the execution of the `onResume` and `onPause` callbacks). This is the behavior represented with blue edges in Figure 2, what is typically understood from the Android documentation, and captured in the existing callback control-flow models used for static analysis.

We implemented a model where `onClick` could be invoked only when its `Activity` was running and found this assumption to be invalid on several traces. We inferred that the mistake was due to the wrong "bounding" of the `onClick` callbacks in the `Activity` lifecycle since in all the traces:

i) the first callback that was erroneously disabled in the model was the `onClick` callback; and

ii) the `onClick` callback was disabled in the model just after the execution of an `onPause` callback that appeared before in the trace, without an `onResume` callback in between (and hence, outside the active state of the `Activity`.)

It turns out that both `finish` and `startActivity` cause the `Activity` to pause without preventing the pending `onClick` invocations from happening, as represented in the red edges connected to `onClick` in Figure 2. We validated such behaviors by writing and executing a test application and finding its description in several Stack Overflow posts [49, 48]. The fix for this issue is to detect the finishing state of the `Activity` and to not disable the `onClick` callback in this case.

7 Related Work

Several works [5, 8, 42, 45, 24, 40, 37, 43, 24] propose different callback control-flow models. Many previous works, like FlowDroid [5] and Hopper [8], directly implement the lifecycle of Android components. While the main intention of these tools is to implement the lifecycle automata, in practice, they also encode some of the effects of callins invoked in the app code in an ad-hoc manner. For example, FlowDroid determines if and where a callback (e.g., `onClick`) is registered using a pre-defined list of callin methods and an analysis of the app call graph. Hopper implements the lifecycle callback control flow directly in a static analysis algorithm that efficiently explores the interleaving of Android components. In contrast, our work starts from the observation that reasoning about protocol violations requires capturing, in a first-class manner, the effects that invoking a callin has on the future execution of callbacks (and vice-versa).

Callback control-flow graphs [53] are graphs of callbacks generated from an application and a manually written model of the framework. Perez and Le [37] generate callback control-flow graphs with constraints relating program variables to callback invocations analyzing the Android framework. Such models can indirectly capture callin effects via the predicates on the program state. With lifestate, we carefully focus on what is observable at the app-framework interface so that lifestate specifications are agnostic to the internal implementation details of the framework. DroidStar [40] automatically learns a callback typestate automaton for an Android object from a developer-specified set of transition labels using both callbacks and callins symbols. Such automata specifically represent the protocol for a single object and, differently from lifestate, their labels are not parameterized messages. A callback typestate is thus a coarser abstraction than lifestate since it cannot express the relationships between different message occurrences that are required to describe multi-object protocols.

There exist other classes of framework models that represent different and complementary aspects of the framework than the callback control flow captured by lifestate. For example, Fuchs et al. [19] and Bastani et al. [6] represent the "heap properties" implicitly imposed by the framework. EdgeMiner [11] and Scandal [29] model the registration of callbacks. Droidel [9] also captures callback registration by modeling the reflection calls inside the Android framework code. Similarly, Pasket [25] automatically learns implementations of framework classes that behave according to particular design patterns.

While framework models have been extensively used to support static and dynamic analysis, not much attention has been paid to validating that the models soundly capture the semantics of the real framework. Wang et al. [52] recognized the problem of model unsoundness – measuring unsoundnesses in three different Android framework models. Unsoundnesses were found even using a much weaker notion of model validation than we do in this work. A significant advantage of lifestates is that we can validate their correctness with respect to any execution trace, obtained from arbitrary apps, because they speak generically about the app-framework interface.

There exist several programming languages for asynchronous event-driven systems, such as Tasks [18] and P [15]. In principle, such languages are general enough to develop event-driven systems such as Android. The purpose of our formalization λ_{life} is instead to provide a formalization that captures the app-framework interface.

The protocol verification problem for event-driven applications is related to typestate verification [34, 26, 17], but it is more complex since it requires reasoning about the asynchronous interaction of both callbacks and callins. Dynamic protocol verification is similar in spirit to dynamic event-race detection [32, 23, 7, 31], which predicts if there is an event data-race from execution traces. However, a lifestate violation differs from, and is not directly comparable to, an event data-race. A lifestate violation could manifest as a data race on a framework-internal field, but more commonly it results from encountering an undesirable run-time state within the framework.

8 Conclusion

We considered the problem of specifying event-driven application-programming protocols. The key insight behind our approach is a careful distillation of what is observable at the interface between the framework and the app. This distillation leads to the abstract notions of permitted messages from the framework to the app (e.g., enabled callbacks) and prohibited messages into the framework from the app (e.g., disallowed callins). Lifestate specification then offers the ability to describe the event-driven application-programming protocol in terms of this interface – capturing both what the app can expect of the framework and what the app must respect when calling into the framework. We evaluated our approach by implementing a dynamic lifestate verifier called Verivita and showed that the richness of lifestates are indeed necessary to verify real-world Android apps as conforming to actual Android protocols.

References

1 Android Developers. The Activity Lifecycle. https://developer.android.com/guide/components/activities/activity-lifecycle.html, 2018.

2 Android Developers. UI/Application exerciser monkey. https://developer.android.com/studio/test/monkey.html, 2018.

3 Android Topeka. Crash if rotate device right after press floating action button #4 Topeka for Android. https://github.com/googlesamples/android-topeka/issues/4, 2015.

4 Yauhen Leanidavich Arnatovich, Minh Ngoc Ngo, Hee Beng Kuan Tan, and Charlie Soh. Achieving High Code Coverage in Android UI Testing via Automated Widget Exercising. In *Asia-Pacific Software Engineering Conference (APSEC)*, 2016. doi:10.1109/APSEC.2016.036.

5 Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *Programming Language Design and Implementation (PLDI)*, 2014. doi:10.1145/2594291.2594299.

6 Osbert Bastani, Saswat Anand, and Alex Aiken. Specification Inference Using Context-Free Language Reachability. In *Principles of Programming Languages (POPL)*, 2015. doi:10.1145/2676726.2676977.

7 Pavol Bielik, Veselin Raychev, and Martin T. Vechev. Scalable race detection for Android applications. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 2015. doi:10.1145/2814270.2814303.

8 Sam Blackshear, Bor-Yuh Evan Chang, and Manu Sridharan. Selective control-flow abstraction via jumping. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 2015. doi:10.1145/2814270.2814293.

9 Sam Blackshear, Alexandra Gendreau, and Bor-Yuh Evan Chang. Droidel: A general approach to Android framework modeling. In *State of the Art in Program Analysis (SOAP)*, 2015. doi:10.1145/2771284.2771288.

10 Chris Boyle. Simon Tatham's Puzzles. https://github.com/chrisboyle/sgtpuzzles/blob/658f00f19172bdbceb5329bc77376b40fe550fcb/app/src/main/java/name/boyle/chris/sgtpuzzles/GamePlay.java#L183, 2014.

11 Yinzhi Cao, Yanick Fratantonio, Antonio Bianchi, Manuel Egele, Christopher Kruegel, Giovanni Vigna, and Yan Chen. EdgeMiner: Automatically detecting implicit control flow transitions through the Android framework. In *Network and Distributed System Security (NDSS)*, 2015. URL: https://www.ndss-symposium.org/ndss2015/edgeminer-automatically-detecting-implicit-control-flow-transitions-through-android-framework.

12 Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. The nuXmv Symbolic Model Checker. In *Computer-Aided Verification (CAV)*, 2014. doi:10.1007/978-3-319-08867-9_22.

13 Adrian Chifor. Swiftnotes. https://f-droid.org/en/packages/com.moonpi.swiftnotes/, 2015.

14 D120. Kistenstapeln. https://github.com/d120/Kistenstapeln-Android, 2015.

15 Ankush Desai, Vivek Gupta, Ethan K. Jackson, Shaz Qadeer, Sriram K. Rajamani, and Damien Zufferey. P: safe asynchronous event-driven programming. In *Programming Language Design and Implementation (PLDI)*, 2013. doi:10.1145/2491956.2462184.

16 Martin Fietz. FeedRemover: already running - issue #1304 - AntennaPod/AntennaPod. https://github.com/AntennaPod/AntennaPod/issues/1304, 2015.

17 Stephen J. Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. Effective typestate verification in the presence of aliasing. *ACM Trans. Softw. Eng. Methodol.*, 17(2), 2008. doi:10.1145/1348250.1348255.

18 Jeffrey Fischer, Rupak Majumdar, and Todd D. Millstein. Tasks: language support for event-driven programming. In *Partial Evaluation and Program Manipulation (PEPM)*, 2007. doi:10.1145/1244381.1244403.

19 Adam P. Fuchs, Avik Chaudhuri, and Jeffrey S. Foster. SCanDroid: Automated security certification of Android applications. Technical Report CS-TR-4991, University of Maryland, College Park, 2009.

20 Marko Gargenta. Yamba. https://github.com/learning-android/Yamba/blob/429e37365f35ac4e5419884ef88b6fa378c023f8/src/com/marakana/android/yamba/StatusFragment.java, 2014.

21 Marko Gargenta and Masumi Nakamura. *Learning Android*. O'Reilly Media, 2014.

22 Nicolas Guillaumin. OSMTracker for Android. https://github.com/nguillaumin/osmtracker-android/blob/d80dea16e456defe5ab62ed8b5bc35ede363415e/app/src/main/java/me/guillaumin/android/osmtracker/gpx/ExportTrackTask.java, 2015.

23 Chun-Hung Hsiao, Cristiano Pereira, Jie Yu, Gilles Pokam, Satish Narayanasamy, Peter M. Chen, Ziyun Kong, and Jason Flinn. Race detection for event-driven mobile applications. In *Programming Language Design and Implementation (PLDI)*, 2014. doi:10.1145/2594291.2594330.

24 Jinseong Jeon, Kristopher K. Micinski, and Jeffrey S. Foster. SymDroid: Symbolic execution for Dalvik bytecode. Technical report, Department of Computer Science, University of Maryland, College Park, 2012.

25 Jinseong Jeon, Xiaokang Qiu, Jonathan Fetter-Degges, Jeffrey S. Foster, and Armando Solar-Lezama. Synthesizing framework models for symbolic execution. In *International Conference on Software Engineering (ICSE)*, 2016. doi:10.1145/2884781.2884856.

26 Pallavi Joshi and Koushik Sen. Predictive Typestate Checking of Multithreaded Java Programs. In *Automated Software Engineering (ASE)*, 2008. doi:10.1109/ASE.2008.39.

27 Vladislav Kaplun. Update RequestAsyncTask.java by kaplad - Pull Request #315 - facebook/facebook-android-sdk. https://github.com/facebook/facebook-android-sdk/pull/315, 2014.

28 Maria Kechagia and Diomidis Spinellis. Undocumented and unchecked: exceptions that spell trouble. In *Mining Software Repositories, (MSR)*, 2014. doi:10.1145/2597073.2597089.

29 Jinyung Kim, Yongho Yoon, Kwangkeun Yi, and Junbum Shin. SCANDAL: Static analyzer for detecting privacy leaks in Android applications. *IEEE Mobile Security Technologies (MoST).*, 2017.

30 Paul Blain Levy. Call-by-push-value: Decomposing call-by-value and call-by-name. *Higher-Order and Symbolic Computation*, 19(4), 2006. doi:10.1007/s10990-006-0480-6.

31 Pallavi Maiya, Rahul Gupta, Aditya Kanade, and Rupak Majumdar. Partial Order Reduction for Event-Driven Multi-threaded Programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2016. doi:10.1007/978-3-662-49674-9_44.

32 Pallavi Maiya, Aditya Kanade, and Rupak Majumdar. Race detection for Android applications. In *Programming Language Design and Implementation (PLDI)*, 2014. doi:10.1145/2594291.2594311.

33 Shawn Meier, Sergio Mover, and Bor-Yuh Evan Chang. Lifestate: Event-Driven Protocols and Callback Control Flow (Extended Version). *CoRR*, abs/, 2019. arXiv:1906.04924.

34 Nomair A. Naeem and Ondrej Lhoták. Typestate-like analysis of multiple interacting objects. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. ACM, 2008. doi:10.1145/1449764.1449792.

35 NextGis. NextGisLogger. https://github.com/nextgis/nextgislogger, 2017.

36 OneBusAway. IllegalStateException: Fragment BaseMapFragment not attached to Activity #570 OneBusAway. https://github.com/OneBusAway/onebusaway-android/issues/570, 2016.

37 Danilo Dominguez Perez and Wei Le. Predicate callback summaries. In *International Conference on Software Engineering (ICSE)*, 2017. doi:10.1109/ICSE-C.2017.95.

38 PingPlusPlus. Ping Plus Plus. https://github.com/PingPlusPlus/pingpp-android, 2017.

39 Steve Pomeroy. The Complete Android Activity/Fragment Lifecycle v0.9.0. https://github.com/xxv/android-lifecycle, 2014.

40 Arjun Radhakrishna, Nicholas V. Lewchenko, Shawn Meier, Sergio Mover, Krishna Chaitanya Sripada, Damien Zufferey, Bor-Yuh Evan Chang, and Pavol Cerný. DroidStar: callback typestates for Android classes. In *International Conference on Software Engineering (ICSE)*, 2018. doi:10.1145/3180155.3180232.

41 Red Reader. Crash during commenting #467 RedReader. https://github.com/QuantumBadger/RedReader/issues/467, 2017.

42 A. Rountev, D. Yan, S. Yang, H. Wu, Y. Wang, and H. Zhang. GATOR: Program analysis toolkit for Android. http://web.cse.ohio-state.edu/presto/software/, 2017.

43 Atanas Rountev and Dacong Yan. Static Reference Analysis for GUI Objects in Android Software. In *Code Generation and Optimization (CGO)*, 2014. doi:10.1145/2544137.2544159.

44 sh1ro. NovelDroid. https://github.com/sh1r0/NovelDroid/blob/f3245055d7a8bcc69a9bca278fbe890081dac58a/app/src/main/java/com/sh1r0/noveldroid/SettingsFragment.java, 2016.

45 Eric Smith and Alessandro Coglio. Android platform modeling and Android app verification in the ACL2 theorem prover. In *Verified Software: Theories, Tools, and Experiments (VSTTE)*, 2015. doi:10.1007/978-3-319-29613-5_11.

46 StackOverflow Post. Got exception: fragment already active. https://stackoverflow.com/questions/10364478/got-exception-fragment-already-active, 2012.

47 StackOverflow Post. Alertdialog creating exception in android. https://stackoverflow.com/questions/15104677/alertdialog-creating-exception-in-android, 2013.

48 StackOverflow Post. OnClickListener fired after onPause? https://stackoverflow.com/questions/31432014/onclicklistener-fired-after-onpause, 2015.

49 StackOverflow Post. Android: click event after Activity.onPause(). https://stackoverflow.com/questions/38368391/android-click-event-after-activity-onpause, 2016.

50 Robert E. Strom and Shaula Yemini. Typestate: A Programming Language Concept for Enhancing Software Reliability. *IEEE Trans. Software Eng.*, 12(1), 1986.

51 Matthias Urhahn. AudioBug. https://github.com/d4rken/audiobug, 2017.

52 Yan Wang, Hailong Zhang, and Atanas Rountev. On the unsoundness of static analysis for Android GUIs. In *State of the Art in Program Analysis (SOAP)*, 2016. doi:10.1145/2931021.2931026.

53 Shengqian Yang, Dacong Yan, Haowei Wu, Yan Wang, and Atanas Rountev. Static Control-Flow Analysis of User-Driven Callbacks in Android Applications. In *International Conference on Software Engineering (ICSE)*, 2015. doi:10.1109/ICSE.2015.31.

Godot: All the Benefits of Implicit and Explicit Futures

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

Dave Clarke
Storytel, Stockholm, Sweden

Ludovic Henrio
Univ Lyon, EnsL, UCBL, CNRS, Inria, LIP, France
ludovic.henrio@ens-lyon.fr

Einar Broch Johnsen
University of Oslo, Norway
einarj@ifi.uio.no

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

Abstract

Concurrent programs often make use of futures, handles to the results of asynchronous operations. Futures provide means to communicate not yet computed results, and simplify the implementation of operations that synchronise on the result of such asynchronous operations. Futures can be characterised as implicit or explicit, depending on the typing discipline used to type them.

Current future implementations suffer from “future proliferation”, either at the type-level or at run-time. The former adds future type wrappers, which hinders subtype polymorphism and exposes the client to the internal asynchronous communication architecture. The latter increases latency, by traversing nested future structures at run-time. Many languages suffer both kinds.

Previous work offer partial solutions to the future proliferation problems; in this paper we show how these solutions can be integrated in an elegant and coherent way, which is more expressive than either system in isolation. We describe our proposal formally, and state and prove its key properties, in two related calculi, based on the two possible families of future constructs (data-flow futures and control-flow futures). The former relies on static type information to avoid unwanted future creation, and the latter uses an algebraic data type with dynamic checks. We also discuss how to implement our new system efficiently.

2012 ACM Subject Classification Software and its engineering → Concurrency control; Software and its engineering → Concurrent programming languages; Software and its engineering → Concurrent programming structures

Keywords and phrases Futures, Concurrency, Type Systems, Formal Semantics

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2019.2

Supplement Material ECOOP 2019 Artifact Evaluation approved artifact available at `https://dx.doi.org/10.4230/DARTS.5.2.1`

Funding Part of this work was funded by the Swedish Research Council, Project 2014-05-545.

1 Introduction

Concurrent programs often make use of futures [4] and promises [27], which are handles to possibly not-yet-computed values, that act like a one-off channel for communicating a result from (often a single) producers to consumers. Futures and promises simplify concurrent programming in several ways. Perhaps most importantly, they add elements of structured

33rd European Conference on Object-Oriented Programming (ECOOP 2019).
Editor: Alastair F. Donaldson; Article No. 2; pp. 2:1–2:28
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

```
def addition(x: Int, y: Int): Int
  x + y
end
```

```
def addition(x: Fut[Int], y: Fut[Int]): Int
  get(x) + get(y)
end
```

Figure 1 *Left.* Data-flow, implicitly typed future, i.e., any argument may be a future value, not visible to the developer. *Right.* Control-flow, explicitly typed future, i.e., the function only accepts future values; synchronisation constructs reduce the future nesting level, e.g., `get`.

programming to message passing, i.e., a message send immediately returns a future, which mimics method calls with a single entry and single exit. This simplifies the control-flow logic, avoids explicit call-backs, and allows a single result to be returned to multiple interested parties – without the knowledge of the producer – through sharing of the future handle. A future is *fulfilled* when a value is associated with it. Futures are further used as synchronisation entities: computations can check if a future is fulfilled (`poll`), block on its fulfilment (`get`), and register a piece of code to be executed on its fulfilment (future chaining – `then`), etc. Promises are similar to (and often blurred with) futures. The main difference is that fulfilment is done manually through a separate first-class handle created at the same time as the future.

Futures are often characterised as either *implicit* or *explicit*, depending on the typing discipline used to type them. Implicit futures are transparent, i.e., it is not generally possible to distinguish in a program's source whether a variable holds a future value or a concrete value. As a consequence, an operation `x + y` may block if either `x` or `y` are future values. This is called *wait-by-necessity* because blocking operations are hidden from the programmer and only performed when a concrete value is needed. With implicit futures, any function that takes an integer can be used with a future integer, which makes code more flexible and avoids duplication (Fig. 1, *Left*). Explicit futures, in contrast, use future types to distinguish concrete values from future values, e.g., `int` from `Fut[int]`, and rely on an explicit operation, which we will call `get`, to extract the `int` from the `Fut[int]`. The types and the explicit `get` make it clear in the code what operations may cause latency, or block forever. The types also make harder to reuse code that mixes future and concrete values (Fig. 1, *Right*).

Because implicit futures allow future and concrete values to be used interchangeably, they can delay blocking on a future until its value is needed for a computation to move forward. Implementing the same semantics with explicit futures requires considerable effort to deal with any possible combination of future and concrete values at any given juncture.

Programs built from cooperating concurrent processes, like actor programs, commonly compute operations by multiple message sends across several actors, each returning a future. This is implemented by nesting several futures, e.g., $f_1 \leftarrow f_2 \leftarrow f_3$ such that f_1 is fulfilled by f_2 which is fulfilled by f_3. While implicit futures hide these structures by design, explicit futures suffer from a blow-up in the number of `get` operations that must be applied to extract the value, but also in the amount of wrappers that must be added to type the outermost future value. Notably, this makes tail recursive message-passing methods impossible to type as the number of type wrappers must mirror the depth of the recursion.

Futures are important for structuring and synchronising asynchronous activities and have been adopted in mainstream languages like Java [32, 17], C++ [26], Scala [41], and JavaScript [28]. In the actor world, futures reduce complexity considerably by enabling an actor to internally join on the production of several values as part of an operation. Alternative approaches either become visible in an actors interface and require manual "buffering" of intermediate results as part of the actor's state, or rely on a receive-like construct and the ability to pass an actor any message, which loses the benefit of a clearly defined interface. With the prevalent way of typing futures – as used for example in Java and Scala – a programmer must choose between introducing blocking to remove future type wrappers [20], or break away from typical structured programming idioms when translating a program into an equivalent actor-based program.

This paper unifies and extends two recent bodies of work on explicit and implicit futures. Henrio [20] observed that the literature directly ties *data flow-driven* and *control flow-driven* futures to the implicit- and explicit-future dichotomy, respectively (e.g., Fig. 1). This work explored the design space of data-flow/control-flow and implicit/explicit dichotomy to support this argument, and developed a combination of data-flow and explicit futures, with a future type that abstracts nesting, avoids the aforementioned explosion of future wrappers and `get` calls for tail recursive methods or pipelines with multiple asynchronous stages, making the chains of futures completely transparent. Fernandez-Reyes et al. [13] proposed an *explicit delegation operation* precisely for handling long (possibly tail recursive) pipelines. Instead of introducing a new future type that hides nesting, this work identifies common delegation patterns in actor-based programs, and proposes a delegation operation that avoids creating unwarranted nested futures. In this system the programmer can control exactly on which stages in a pipeline should be possible to synchronise, reducing the number of created futures.

We distinguish two kinds of futures. We call *control-flow futures* the future constructs that can be implemented by a parametric future type and where each synchronisation blocks until exactly one asynchronous task finishes, the fact that a single fulfilment instruction resolves the blocked state explains the control-flow name. We call *data-flow futures* the future constructs where the synchronisation consists in blocking until a concrete usable value is available, consequenty a single synchronisation might wait for the termination of several asynchronous tasks. Data-flow futures are usually implemented by implicit futures.

Contributions. This paper shows how to integrate data-flow futures and control-flow futures, and how to seamlessly combine them. We show how data-flow futures can be implemented using control-flow futures and the converse. Our model provides future delegation, data-flow futures, and control-flow futures at the same time, giving the programmer precise control over future access, as well as automatic elision of unnecessary nested futures. More precisely:

- We overview three inherent problems with both explicit and implicit futures that limit their applicability or performance (Section 2).
- We discuss existing mitigation strategies based on typically available future operations or alternatives (Section 3.1) – as well as recent work on data-flow futures [20] and delegation [13] that aim to address overlapping subsets of these problems – and show that *none addresses all of the problems* (Sections 3.2 and 3.3).
- We propose Godot (Section 4), the first system that seamlessly integrates data-flow futures and control-flow futures in a single explicit system. In addition to *addressing all the problems* in Section 2, the system improves on the data-flow explicit futures of [20] by adding support for parametric polymorphism, and improves on the delegation in [13] by allowing it to be applied automatically for data-flow explicit futures.
- We provide two alternative formalisations of Godot (with a common foundation introduced in Section 4.1). FlowFut shows how to extend a data-flow future language with control-flow futures; it is mostly aimed at languages with no current future support (Section 4.2). FutFlow shows how to extend a control-flow future language with data-flow futures; it is aimed at languages with typical explicit future support (Section 4.3).
- We prove progress and type preservation of FlowFut and FutFlow; and
- We introduce a type-driven optimisation strategy for eliding the creation of nested futures (Section 5) and a discussion on the implementation of our system.

In addition to the above, Section 6 discusses related work and Section 7 concludes.

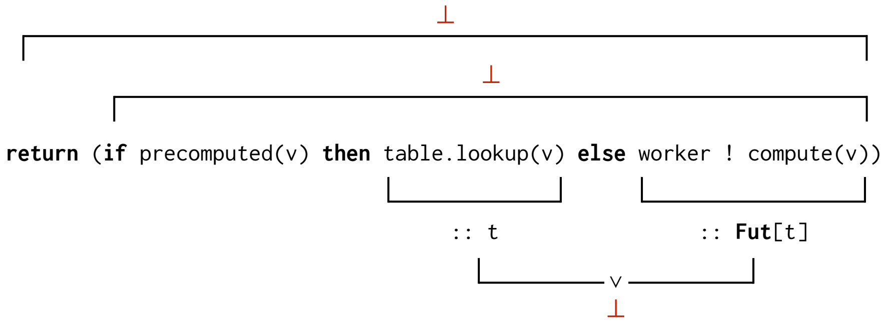

⊥
⊥
return (if precomputed(v) then table.lookup(v) else worker ! compute(v))
:: t
:: Fut[t]
∨
⊥

Figure 2 Type Proliferation making code untypable; ⊥ denotes the absence of a type for a term.

2 Problems Inherent in Explicit and Implicit Futures

Both implicit and explicit futures have limitations. In this section, we overview the problems that exist with exising futures. We use examples presented in pseudocode, where `o ! m` and `o.m` denote an asynchronous and a synchronous call to a method `m` of an object `o`, respectively.

The Type Proliferation Problem. The way explicit futures are generally added to languages, they end up mirroring the communication structure of a program: the result of an asynchronous operation is typed **Fut**`[t]`, the result of an asynchronous operation that returns the result of another asynchronous operation is **Fut**`[`**Fut**`[t]]`, etc. This breaks abstraction and makes code inflexible. For example, consider the following code example that returns values from two different sources. If the answer is precomputed, it is fetched from a table, otherwise the computation is delegated to some worker (see Figure 2 for details).

```
return if precomputed(v) then table.lookup(v) else worker ! compute(v)
```

As denoted by the ⊥ types, this is not well-typed as the branches have different types, without any `join`: `table.lookup(v)` returns a value of type `t`, whereas `worker ! compute(v)` returns a **Fut**`[t]`. Thus, such a common pattern will not work straightforwardly in a program. For similar reasons, tail recursive asynchronous methods are not possible to type as the depth of the recursion must be mirrored in the returned future type. Last, also an effect of the same root cause, explicit futures complicate code reuse – forcing code duplication for operations that should be possible to apply to values of both future and concrete type.

This problem has been previously identified in [16, 20], where the authors showed that there was no direct encoding from implicit futures to explicit futures because an unbounded number of control-flow synchronisations and an unbounded parametric type may be needed to encode a single data-flow future. This is typically the case if one tries to write an asynchronous tail recursive function. For this reason there is no simple encoding of data-flow futures with control-flow futures; Section 4.3 will show how, with a boxing operator and a few changes in the type system, we are able to encode data-flow futures using control-flow futures and to overcome the type resolution problem.

We call this problem, which applies to explicit futures, the *Type Proliferation Problem.*

The Future Proliferation Problem. Implicit futures avoid the Type Proliferation Problem by abstracting whether a variable has been computed or not. However, the way implicit futures are generally added to languages, a similar problem appears at run-time. While tail recursion is possible, running tail calls in constant space is not possible because each recursive call gives rise to an additional future indirection.

The creation of nested futures $f_1 \leftarrow f_2 \leftarrow f_3$ (etc.) introduces additional latency because the fulfilment of a nest of futures of depth n adds n additional operations, which in worst-case must be scheduled separately. Moreover, because a future can be fulfilled with an unfulfilled future, in some implementations, an actor may be falsely deemed schedulable, only to take a step to block on the unfulfilled nested future. For example, f_1 will be "falsely fulfilled" by the unfulfilled future f_2; if the activity blocking on f_1 is scheduled to run before f_2 and f_3 are fulfilled, the operation will block again on f_2 or f_3 (possibly both).

This problem, which applies to both implicit and explicit futures, was pointed out in [13]. We call it the *Future Proliferation Problem.*

The Fulfilment Observation Problem. The abstraction of implicit futures further loses precision. Consider the following code snippet that could be part of a simple load balancer, that farms out jobs to idle workers, and a call to the load balancer to perform some work.

```
def perform(job : Job) { return idle_worker() ! do_work(job) }
var f = load_balancer ! perform(my_job)
```

A call to `perform()` results in a nested future: the outermost future captures whether the load balancer has found an idle worker and successfully delegated the job; the innermost future captures the result of `do_work()`. With explicit futures we can observe the state of the task:

```
get(f) −−block until do_work has been called
get(get(f)) −−block until do_work has finished
```

However, with implicit futures, it is not possible to make this distinction as any access will block until the innermost value is returned. Thus, we cannot observe the current stage of such an operation using futures. Concurrent and scheduling library developers need to access the intermediate steps of computations, and this issue hinders the code that they can write.

Similarly, if an unfulfilled future is stored somewhere, say in a hash table implemented by an actor, retrieving it is tricky without accidentally blocking on the production of the future – an unknown operation – rather than the result of `hash_table.lookup()`. Since a hash table may store both concrete and future values due to the nature of implicit futures, knowing when to *not* call **get** on the result of a hash table lookup is not discernible by local reasoning.

This has been highlighted in [21, 20] as the major source of difference between existing implicit and explicit futures. Because of this different behaviour, there is no simple encoding of control-flow futures with data-flow futures. In Section 4.2 we will show such an encoding that relies on a slight adaptation of the type system, and a boxing operator.

This problem applies to implicit futures, we call it the *Fulfilment Observation Problem.*

Following this problem overview, the next section presents existing partial solutions.

3 Current Solutions to Future Problems

This section surveys how existing techniques can be used to partially overcome the problems outlined in Section 2. In particular, in Sections 3.2–3.3, we give an informal overview of prior work that this paper amalgamates to address all of the problems in a coherent way.

3.1 Standard Mitigation Strategies and Problem Avoidance

Manual Unpacking of Futures. Avoiding the Type Proliferation Problem is possible by manually unpacking and returning the concrete value of each future using the aforementioned `get` operation. In the case of the guarded return example, we could write the following:

```
return if precomputed(v) then table.lookup(v) else get(worker ! compute(v))
```

This causes the `else` branch to block its execution until the `compute()` method has finished and is notified of the fulfilment of the current future. This has several problems:

- **Bottleneck.** The enclosing actor is blocked from processing other requests while waiting for `worker ! compute(v)` to finish. This causes subsequent messages to block, even if they could be served from precomputed data. Thus, the blocking `get` introduces a bottleneck.
- **False Fulfilment.** Delaying the return until the concrete value is produced avoids false fulfilment but instead adds an additional step to the operation which adds and unnecessary latency. The task of unpacking the innermost future and fulfilling the outermost must now be scheduled before the client of the outermost future is unblocked. Notably, this changes fulfilment from pull – clients blocking until the value is available, to push – propagating fulfilment of a nested future inwards out. (We revisit this in Section 5.)

Some actor languages that use futures provide a cooperative scheduling construct "`await`" that allows the current method to be suspended pending the fulfilment of a future *without blocking the currently executing actor*. This avoids the bottleneck problem above, but at the same time introduces race conditions due to the possible interleaving of suspended methods – these race conditions only appear through side effects [8].

Explicit Spawning of a Task. The explicit creation of a task can be used to solve the Type Proliferation Problem. In the case of the example, the `then` branch spawns a task for something that needed not be asynchronous:

```
return if precomputed(v) then async(table.lookup(v)) else worker ! compute(v)
```

This causes the type checker to accept the program at the expense of performance. The creation of a task involves memory allocation, scheduling of the task, and computation of the task body, which is a simple asynchronous operation. This is feasible, but not optimal.

Future Chaining to Avoid Blocking and Nesting. Future chaining can be used to avoid unnecessary blocking in some cases. Future chaining supports the construction of pipelines of futures which are not nested, but still need to be represented at run-time. For example, here is how we could add the result of `worker ! compute(v)` to the table of precomputed values (so it effectively becomes a cache) *without* delaying the returning of the result to a client:

```
var result = worker ! compute(v)
result.then(fun r => this.table.add(v, r))
return result
```

The `then` method attaches a callback function that will be run upon the fulfilment of `result`, with `r` bound to the value used to fulfil `result`. Although the callback registration happens before the `return`, the execution of the registered function does not happen until after the future is fulfilled, meaning it causes no delay.

While chaining can avoid some Type Proliferation, it does not enable tail recursive calls.

Changing the Program Structure: Replace Return with Message Send. An alternative solution is to give up on structured programming ideals and instead of returning values back up the call stack, instruct the producer of a value how to communicate the result to its consumers. Here is an example of how that might look in the Type Proliferation Problem example:

```
if precomputed(v) then client ! receive(table.lookup(v)) --send result to client
else worker ! compute(v, client) --pass client id to worker
```

With this design, a method that previously returned a value must be passed the identity of the consumer of the result as an argument (possibly a list of consumers) to explicitly send the result to the consumer(s) according to some agreed-upon protocol. Instead of id(s), it can take as input some lambda function that know how to communicate the result back to interested parties. A downside of this solution is that the consumers must be known at the time of the call. This is in contrast to a caller sharing a returned future with whoever might be interested in the result after the call is made.

This solution requires the existence of a specific method in the consumer for each operation and causes an operation to be spread over multiple methods. Submitting multiple jobs for execution requires manually handling the possibility of the results coming back in any order, and possibly provide multiple different methods for getting the results.

Returning values differently from synchronous and asynchronous computations increases complexity for functions and data structures that should be usable in both contexts. This is typical in, e.g., Cilk [6] where a function can be "spawned" asynchronously or called synchronously, and in many actor languages (e.g., Joelle [10], ABS [22] and Encore [7]) where an actor's interface is asynchronous externally but synchronous internally.

Changing the Program Structure: Use Promises Instead of Futures. Both the Type Proliferation Problem and the Future Proliferation Problem can be overcome by resorting to manually handled *promises*: instead of passing the identity of the recipient around, we pass around a pointer to a shared space where the result can be stored. Promises are similar to futures, but are less transparent and, because they are manipulated explicitly both on the side of the producer and the consumer, lack many of the guarantees of futures: promises are created and fulfilled manually and are thus not guaranteed to be fulfilled at all, may be fulfilled more than once, possibly by several actors.[1] With this design, workers are passed a promise created by a client. Upon finishing the work, the worker fulfils the promise.

3.2 Data-flow Explicit Futures

Henrio [20] observed that the traditional dichotomy of implicit and explicit futures was focusing mainly on typing and not on how futures are synchronised, and proposed an alternative categorisation: *control-flow* futures and *data-flow* futures, depending on how the synchronisation on futures works. With control-flow synchronisation, each nested future must be explicitly unpacked using **get** to return another future or a concrete value. Data-flow synchronisation is wait-by-necessity as usual for implicit futures: nesting is invisible, and a **get** always returns a concrete value, even from a nested future. Separating typing from synchronisation allows new combinations of future semantics, such as explicit data-flow futures, which address the Type Proliferation Problem of Section 2.

[1] Futures have static fulfilment guarantees, they are implicitly fulfilled, unless the fulfilling computation gets stuck. Promises have no static fulfilment guarantees, even when the program is not stuck.

The traditional way of typing explicit futures, by a parametric type, has always led to control-flow synchronisation on futures while data-flow futures had no future type. Data-flow synchronisation naturally leads to an alternative type system called *DeF*, such that the run-time structure of futures is no longer mirrored by their type. Instead, a `Fut[t]` type represents *zero* or more nested futures – the *zero* means that a concrete value may appear as a future value. This allows future-typed code to be reused with concrete values but also allows tail recursion and methods returning either a concrete value or a future. In the Type Proliferation Problem, the branches would still have different types (`t` and `Fut[t]`), but `t` can be lifted to `Fut[t]`, collapsing the `Fut[Fut[t]]` returned by the entire asynchronous expression into a `Fut[t]`. Let the keyword **`async`** denote the spawning of an asynchronous task.

```
async (if precomputed(v) then table.lookup(v) else worker ! compute(v))
```

Data-flow explicit futures address the Type Proliferation Problem but it does not address the Future Proliferation Problem or the Fulfilment Observation Problem.

A Formal Introduction to DeF. For simplicity and to align with upcoming sections, we adapt Henrio's DeF calculus to a concurrent, lambda-based calculus. We use an **`async`** construct to spawn tasks and a **`get`** construct for data-flow synchronisation on a future. The types are the basic types $\mathcal{K}$, abstraction and futures.

$$\begin{array}{rcll}
\textit{Expressions} & e & ::= & v \mid e\ e \mid \mathtt{return}\ e \mid \mathtt{async}\ e \mid \mathtt{get}\ e \\
\textit{Values} & v & ::= & c \mid x \mid f \mid \lambda x.e \\
\textit{Types} & \tau & ::= & \mathcal{K} \mid \tau \to \tau \mid \mathtt{Fut}\,\tau \\
\textit{Evaluation context} & E & ::= & \bullet \mid E\ e \mid v\ E \mid \mathtt{return}\ E \mid \mathtt{get}\ E
\end{array}$$

The operational semantics use a small-step reduction semantics with reduction-based, contextual rules for evaluation within tasks. An evaluation context E contains a hole $\bullet$ that denotes where the next reduction step happens. Configurations consist of tasks $(task_f\ e)$, unfulfilled futures (fut_f) and fulfilled futures $(fut_f\ v)$. When a task finishes, i.e., reduces to a value v, the corresponding future is fulfilled with v.

We show the most interesting reduction rules in Figure 3: Red-Async spawns a new computation and puts a fresh future in place of the spawned expression. Red-Get-Val applies **`get`** to a concrete value which reduces to the value itself. Red-Get-Fut applies **`get`** on a future chain of length ≥ 1, reducing it future by future. A run-time test, $isfut?(v)$, is required to check whether v is a future value or a concrete value.

Figure 3 shows the most interesting type rules. We first have two sub-typing rules: a concrete value can be typed as a future, and nested future types are unnecessary. By T-Async, any well-typed expression of type τ can be spawned off in an asynchronous task that returns a $\mathtt{Fut}\,\tau$. By T-Get, **`get`** can be applied to unpack a $\mathtt{Fut}\,\tau$, yielding a value of type τ.

Summary. Data-flow futures allow the programmer to focus on expressing future-like algorithms without explicitly manipulating every synchronisation point. A single future and multiple nested futures are indistinguishable with respect to types and synchronisation. Because the type system allows the implicit lifting of a concrete value to a (fulfilled) future value, code that uses futures can be reused with concrete values.

Reduction rules: $e \rightarrow e'$

(Red-Async)
$$\frac{\textit{fresh } f}{(\textit{task}_g\ E[\texttt{async}\ e]) \rightarrow (\textit{fut}_f)\ (\textit{task}_f\ e)\ (\textit{task}_g\ E[f])}$$

(Red-Get-Val)
$$\frac{\neg \textit{isfut?}(v)}{(\textit{task}_f\ E[\texttt{get}\ v]) \rightarrow (\textit{task}_f\ E[v])}$$

(Red-Get-Fut)
$$\frac{\textit{isfut?}(g)}{(\textit{task}_f\ E[\texttt{get}\ g])\ (\textit{fut}_g\ v) \rightarrow (\textit{task}_f\ E[\texttt{get}\ v])\ (\textit{fut}_g\ v)}$$

Subtyping:
$$\tau <: \texttt{Fut}\ \tau$$
$$\texttt{Fut}\ (\texttt{Fut}\ \tau) <: \texttt{Fut}\ \tau$$

Typing rules: $\Gamma \vdash_\rho e : \tau$

(T-Async)
$$\frac{\Gamma \vdash_\tau e : \tau}{\Gamma \vdash_\rho \texttt{async}\ e : \texttt{Fut}\ \tau}$$

(T-Get)
$$\frac{\Gamma \vdash_\rho e : \texttt{Fut}\ \tau}{\Gamma \vdash_\rho \texttt{get}\ e : \tau}$$

Figure 3 Reduction and typing rules for data-flow explicit futures.

3.3 Delegating Future Fulfilment

To avoid the Type Proliferation Problem and Future Proliferation Problem of Section 2, Fernandez-Reyes et al. [13] proposed a delegation construct that delegates the fulfilment of the current-in-call future to another task in the context of control-flow explicit futures. This **`forward`** construct supports tail-recursive asynchronous methods and allows them to run in *constant space*, because only a single future is needed.[2] The Fulfilment Observation Problem is avoided because of the control-flow synchronisation. Library code can distinguish the futures it manipulates and the concrete values that client programs are interested in.

In contrast to DeF, delegation requires an explicit keyword. This can be seen in the Type Proliferation Problem example by inserting **`return`** in the **`then`**-branch and **`forward`** in the **`else`**-branch. In the **`then`**-branch, the concrete value is returned; in the **`else`**-branch, **`forward`** delegates to a worker to *fulfil the current future.* In both cases, the return type is `Fut[t]`. This shows how a method's return type no longer needs to (but may) mirror the internal communication structure of a method in order to avoid the Fulfilment Observation Problem:

```
async (if precomputed(v) then return table.lookup(v)
       else forward worker ! compute(v))
```

Delegation and explicit future types address the Future Proliferation Problem and Fulfilment Observation Problem, but only in part the Type Proliferation Problem – reuse is still limited by future types, causing code duplication or blocking to remove future types.

A Formal Introduction to Forward. We present the semantics of delegation similarly through a concurrent, lambda-based calculus, adapted from Fernandez-Reyes' work. The syntax reuses the concepts from the previous section and adds the **`forward`** construct which transfers the obligation to fulfil a future to another task and future chaining ($\texttt{then}(e, e)$), which registers a piece of code to be executed on its fulfilment. While the latter is not strictly necessary, its run-time semantics are necessary to express the semantics of **`forward`**, so explicit support for future chaining adds very little complexity. The types are the same as in the previous calculus except that there is no subtyping rule. The typing judgement has an extra parameter, ρ, which prevents the use of **`forward`** under certain circumstances (explained later).

[2] This cannot be observed in Fig. 3 because we have omitted the compilation optimisations [13]. This optimisations follow the same logic as Section 5.

$$e \quad ::= \quad \ldots \mid \texttt{then}(e,e) \mid \texttt{forward}\, e \qquad E \quad ::= \quad \ldots \mid \texttt{then}(E,e) \mid \texttt{then}(v,E) \mid \texttt{forward}\, E$$

We show the most interesting reduction rules in Figure 4: Red-Get captures blocking synchronisation through **get** on a future f. Red-Chain-New attaches a callback e on a future f to be executed (rule Red-Chain-Run) once f is fulfilled. Chaining on a future immediately returns another future which will be fulfilled with the result of the callback. Red-Forward captures delegation. Like **return** it immediately finishes the current task, replacing it with a "chain task" that will fulfil the same future as the removed task. This **chain** will be executed when the delegated task is finished, i.e., when the future h is fulfilled.

Reduction rules: $e \to e'$

$$\frac{}{(task_f\ E[\texttt{get}\, h])\ (fut_h\ v) \to (task_f\ E[v])\ (fut_h\ v)}\ (\text{Red-Get}) \qquad \frac{}{(chain_g\ f\ e)\ (fut_f\ v) \to (task_g\ (e\ v))\ (fut_f\ v)}\ (\text{Red-Chain-Run})$$

$$\frac{fresh\ g}{(task_f\ E[\texttt{then}(h,e)]) \to (fut_g)\ (chain_g\ h\ \lambda x.e)\ (task_f\ E[g])}\ (\text{Red-Chain-New}) \qquad \frac{}{(task_f\ E[\texttt{forward}\, h]) \to (chain_f\ h\ \lambda x.x)}\ (\text{Red-Forward})$$

Typing rules: $\Gamma \vdash_\rho e : \tau$

$$\frac{\Gamma \vdash_\rho e : \texttt{Fut}\,\tau \quad \Gamma, x : \tau \vdash_\bullet e' : \tau'}{\Gamma \vdash_\rho \texttt{then}(e,e') : \texttt{Fut}\,\tau'}\ (\text{T-Chain}) \qquad \frac{\Gamma \vdash_\rho e : \texttt{Fut}\,\rho \quad \rho \neq \bullet}{\Gamma \vdash_\rho \texttt{forward}\, e : \tau}\ (\text{T-Forward})$$

Figure 4 Reduction and typing rules of forward calculus.

The most interesting type rules deal with future chaining and forward. By T-Forward, fulfilment of the current future can be delegated to any expression returning a future. The requirement $\rho \neq \bullet$ prevents the use of **forward** inside lambda expressions. Otherwise, a lambda could be sent to another task and run in a context different from its defining context, which could inadvertently modify the return type of a task, leading to unsoundness. By T-Forward, any type can be used as the result type. Since **forward** halts the execution of the current task, there is no traditional return value from **forward**, which makes this practice sound. T-Chain types the chaining on the result of any expression returning a future.

Summary. Delegation allows the programmer to push the fulfilment of the *current-in-call future* to another task, thereby avoiding future nesting both in types and at run-time. Here, the result of **get** can be another future and a concrete value cannot be used when a future is expected. While Future Proliferation is avoided, the programmer needs to explicitly insert delegation points and there are restrictions on code reuse with and without future values.

4 Godot: Integrating Data- and Control-Flow Futures and Delegation

The core contribution of this paper is Godot [5], a system that seamlessly integrates data-flow explicit futures and control-flow explicit futures, and extends them to increase expressiveness while reducing the number of future values needed at run-time. The resulting system uses **forward**-style *implicitly* on data-flow futures. For clarity, in the sequel, control-flow futures will retain the **Fut** τ type, and data-flow futures will be denoted by **Flow** τ.

4.1 Design Space and Formal Semantics

Godot is formalised as two distinct versions of a core calculus using a concurrent, task-based, modified version of System F: FlowFut that uses data-flow futures as primitives and uses them to encode control-flow futures (Section 4.2); and FutFlow that uses control-flow futures as primitives and uses them to encode data-flow futures (Section 4.3). The target audience for FlowFut is language designers who wish to add Godot to a language without futures. The target audience for FutFlow is language designers who wish to incorporate Godot in a language that already supports control-flow futures.

The core calculus contains tasks, control-flow futures and data-flow futures, and operations on them. For simplicity, we abstract from mutable state, as this would detract from the main points. We use explicit futures, recall that control-flow futures are typed by $\texttt{Fut}\,\tau$ and data-flow futures by $\texttt{Flow}\,\tau$. Operations on data-flow futures are distinguished by a $\star$, e.g., **get** operates on $\texttt{Fut}\,\tau$ and **get*** operates on $\texttt{Flow}\,\tau$, etc.

The calculus consists of two levels: configurations and expressions. Configurations represent the run-time as a collection of concurrent tasks, futures, and asynchronous chained operations. Expressions correspond to programs and what tasks evaluate to. A task represents a unit of work and its result is placed in either a flow or future abstraction, depending on the type system. A task represents any asynchronous computation, it can for example correspond to a runnable task in Java, or a message treatment in actor and active object languages.

Chaining operations on either data-flow and control-flow futures attaches a closure to the future that will be schedulable when the future is fulfilled. Abstracting from mutable state, we cannot model the consequences of closures with side effects, but we can easily integrate any pre-existing approach, e.g., [9]. With respect to the simple calculi in Section 3, we add a **return** expression which immediately finishes a task with a given return value. This expression has been added to show how we reduce the creation of futures upon returning from a task with respect to data-flow futures. The **return** construct shares limitations with the **forward** construct, which we explain in the coming subsections.

The remainder of Section 4.1 introduces parts of the language that are common to both calculi: run-time configurations, types, and their static and run-time semantics. We delay the presentation of expressions and values, their static and run-time semantics and the type and term encodings of one future type in terms of the other to Sections 4.2 and 4.3.

Syntax. The calculus contain run-time configurations, expressions, and values.

$$\mathit{config} ::= \epsilon \mid (\mathit{flow}_f) \mid (\mathit{flow}_f\, v) \mid (\mathit{fut}_f) \mid (\mathit{fut}_f\, v) \mid (\mathit{task}_f\ e) \mid (\mathit{chain}_f\, f\ e) \mid \mathit{config}\ \mathit{config}$$

Configurations represent running programs. A global configuration *config* represents the global state of the system, e.g., $(\mathit{task}_f\ e)\ (\mathit{flow}_f)$ represents a global configuration with a single task running expression e, whose result will fulfil flow f. Partial configurations *config* show a view of the state of the program, and are multisets of unfulfilled futures ((flow_f) and (fut_f)), fulfilled futures ($(\mathit{flow}_f\ v)$ and $(\mathit{fut}_f\ v)$), tasks $(\mathit{task}_f\ e)$, and chains $(\mathit{chain}_f\ f\ e)$, where the empty configuration is ϵ and multiset union is denoted by whitespace.

Note that *flow* and *fut* configurations do not co-exist. Depending on the calculus, a task fulfils either a *flow* or a *fut*. This distinction is clarified in each respective calculus.

Static Semantics. The types, $\tau ::= \mathcal{K} \mid \tau \to \tau \mid X \mid \forall X.\tau \mid \texttt{Flow}\,\tau \mid \texttt{Fut}\,\tau$, are the common basic types ($\mathcal{K}$), abstraction ($\tau \to \tau$), type variables (X), universal quantification ($\forall X.\tau$), flow types ($\texttt{Flow}\,\tau$) and future types ($\texttt{Fut}\,\tau$). In the typing rules, we assume that

$$\frac{f \in \mathit{dom}(\Gamma)}{\Gamma \vdash (\mathit{flow}_f)\ \mathit{ok}}\ (\textsc{T-UFlow}) \qquad \frac{f : \texttt{Flow}\,\tau \in \Gamma \quad \Gamma \vdash_\tau e : \tau}{\Gamma \vdash (\mathit{task}_f\ e)\ \mathit{ok}}\ (\textsc{T-TaskFlow}) \qquad \frac{f : \texttt{Flow}\,\tau \in \Gamma \quad \Gamma \vdash_\bullet v : \tau}{\Gamma \vdash (\mathit{flow}_f\ v)\ \mathit{ok}}\ (\textsc{T-FFlow})$$

$$\frac{f \in \mathit{dom}(\Gamma)}{\Gamma \vdash (\mathit{fut}_f)\ \mathit{ok}}\ (\textsc{T-UFut}) \qquad \frac{f : \texttt{Fut}\,\tau \in \Gamma \quad \Gamma \vdash_\tau e : \tau}{\Gamma \vdash (\mathit{task}_f\ e)\ \mathit{ok}}\ (\textsc{T-TaskFut}) \qquad \frac{f : \texttt{Fut}\,\tau \in \Gamma \quad \Gamma \vdash_\bullet v : \tau}{\Gamma \vdash (\mathit{fut}_f\ v)\ \mathit{ok}}\ (\textsc{T-FFut})$$

$$\frac{f : \texttt{Flow}\,\tau \in \Gamma \quad g : \texttt{Flow}\,\tau' \in \Gamma \quad \Gamma \vdash_\tau e : \tau' \to \tau}{\Gamma \vdash (\mathit{chain}_f\ g\ e)\ \mathit{ok}}\ (\textsc{T-ChainFlow}) \qquad \frac{f : \texttt{Fut}\,\tau \in \Gamma \quad g : \texttt{Fut}\,\tau' \in \Gamma \quad \Gamma \vdash_\tau e : \tau' \to \tau}{\Gamma \vdash (\mathit{chain}_f\ g\ e)\ \mathit{ok}}\ (\textsc{T-ChainFut})$$

$$\frac{}{\Gamma \vdash \diamond\ \mathit{ok}}\ (\textsc{T-Empty}) \qquad \frac{\begin{array}{ll}\Gamma \vdash \mathit{config}_1\,\mathit{ok} & \mathit{defs}(\mathit{config}_1) \cap \mathit{defs}(\mathit{config}_2) = \varnothing \\ \Gamma \vdash \mathit{config}_2\,\mathit{ok} & \mathit{writers}(\mathit{config}_1) \cap \mathit{writers}(\mathit{config}_2) = \varnothing\end{array}}{\Gamma \vdash \mathit{config}_1\ \mathit{config}_2\ \mathit{ok}}\ (\textsc{T-Config}) \qquad \frac{\begin{array}{c}\Gamma \vdash \mathit{config}\ \mathit{ok} \\ \mathit{dom}(\Gamma) = \mathit{defs}(\mathit{config})\end{array}}{\Gamma \vdash \mathit{config}}\ (\textsc{T-GConfig})$$

Figure 5 Well-formed configurations. The helper functions $\mathit{defs}(\mathit{config})$ and $\mathit{writers}(\mathit{config})$ extract the set of futures (data-flow and control-flow) or writers of futures in a configuration.

the types of the premises are *normalised.* We denote the normalised type τ by $\downarrow\!\tau$, i.e., the type τ with flattened flow types, defined inductively:

$$\downarrow\! K = K \qquad \downarrow\! X = X \qquad \downarrow\! \forall X.\tau = \forall \downarrow\! X. \downarrow\! \tau \qquad \downarrow\!(\tau \to \tau') = \downarrow\!\tau \to \downarrow\!\tau'$$
$$\downarrow\!\texttt{Flow}\,(\texttt{Flow}\,\tau) = \downarrow\!\texttt{Flow}\,\tau \qquad \downarrow\!\texttt{Flow}\,\tau = \texttt{Flow}\,\downarrow\!\tau \text{ if } \tau \neq \texttt{Flow}\,\tau' \qquad \downarrow\!\texttt{Fut}\,\tau = \texttt{Fut}\,\downarrow\!\tau$$

Well-Formed Configurations. Type judgements $\Gamma \vdash \mathit{config}\ \mathit{ok}$ express that configurations are well-formed in an environment Γ that gives the types of futures (Figure 5). Unfulfilled flow and future configurations are well-formed if their variable f exists in the environment (T-UFlow, T-UFut). Tasks are well-formed if their body is well-typed with the type of the future or flow they are fulfilling (T-TaskFlow, T-TaskFut).

The meaning of $\Gamma \vdash_\rho e : \tau$ is that e has type τ under Γ inside a task whose static return type is ρ, where $\rho ::= \tau \mid \bullet$. Once the concrete syntax is introduced for the two calculi, this notation is used to express that a `return` inside e must return a value of type ρ. The special form $\bullet$ of ρ disallows the use of `return`. Thus, by (T-FFlow) and (T-FFut), values of fulfilled flow configurations cannot be lambda expressions containing a `return` expression. Chained configurations are well-formed if their bodies are well-typed. Note that the body must be a lambda function (T-ChainFlow, T-ChainFut).

Configurations are well-formed if all sub configurations have disjoint futures and there are not two tasks writing to the same future (T-Config, T-GConfig). (The definitions of auxiliary functions $\mathit{defs}()$ and $\mathit{writers}()$ are straightforward.) These side conditions ensure that there are no races on fulfilment.

Dynamic Semantics. Configurations consist of a multiset of tasks, data-flow futures and chained configurations with an initial program configuration $(\mathit{flow}_{f_{main}})$ $(\mathit{task}_{f_{main}}\ e)$, where f_{main} is fulfilled by the result of e at the end of execution. Configurations are commutative monoids under configuration concatenation, with ϵ as unit (Figure 6). The configuration evaluation rules (Figure 6) describe how configurations make progress, which is either by some subconfiguration making progress, or by rewriting a configuration to one that will make progress using the equations of multisets.

Equivalence relation

$$\mathit{config}\ \epsilon \equiv \epsilon\ \mathit{config} \qquad \mathit{config}\ \mathit{config}' \equiv \mathit{config}'\ \mathit{config} \qquad \mathit{config}\ (\mathit{config}'\ \mathit{config}'') \equiv (\mathit{config}\ \mathit{config}')\ \mathit{config}''$$

Configuration run-time

$$\overset{(\textsc{R-FulfilFlowValue})}{\frac{\neg \mathit{isflow?}(v)}{(\mathit{task}_f\ v)\ (\mathit{flow}_f) \to (\mathit{flow}_f\ v)}} \qquad \overset{(\textsc{R-FulfilFlow})}{\frac{\mathit{isflow?}(g)}{(\mathit{task}_f\ g) \to (\mathit{chain}_f\ g\ \lambda x.x)}}$$

$$\overset{(\textsc{R-FutFulfilValue})}{\frac{v \neq \blacklozenge\, v'}{(\mathit{task}_f\ v)\ (\mathit{fut}_f) \to (\mathit{fut}_f\ v)}} \qquad \overset{(\textsc{R-FlowCompression})}{(\mathit{task}_f\ \blacklozenge\, g) \to (\mathit{chain}_f\ g\ \lambda x.x)}$$

$$\overset{(\textsc{R-ChainRunFlow})}{(\mathit{chain}_g\ f\ e)\ (\mathit{flow}_f\ v) \to (\mathit{task}_g\ (e\ v))\ (\mathit{flow}_f\ v)} \qquad \overset{(\textsc{R-ChainRunFut})}{(\mathit{chain}_g\ f\ e)\ (\mathit{fut}_f\ v) \to (\mathit{task}_g\ (e\ v))\ (\mathit{fut}_f\ v)}$$

$$\overset{(\textsc{R-Config})}{\frac{\mathit{config} \to \mathit{config}''}{\mathit{config}\ \mathit{config}' \to \mathit{config}''\ \mathit{config}'}} \qquad \overset{(\textsc{R-ConfigEquiv})}{\frac{\mathit{config} \equiv \mathit{config}' \quad \mathit{config}' \to \mathit{config}'' \quad \mathit{config}'' \equiv \mathit{config}'''}{\mathit{config} \to \mathit{config}'''}}$$

Figure 6 Configuration run-time and configuration equivalence rules modulo associativity and commutativity. $\blacklozenge\, v$ represents the encoding of a data-flow future in terms of a control-flow futures.

4.2 FlowFut: Primitive Data-Flow and Encoded Control-Flow Futures

This section presents FlowFut which instantiates the expression syntax of Godot presented in the previous section. FlowFut has primitive support for data-flow futures and support for control-flow futures as an extension, using an encoding in terms of data-flow futures. We first describe a sublanguage that only has data-flow futures before extending it with control-flow futures. FlowFut illustrates how to extend a language with data-flow future like ProActive [3], JavaScript, or DeF [20] to support control-flow futures. Note that DeF is the only language that has explicit data-flow futures but it has currently no implementation.

The FlowFut sublanguage contain expressions and values:

$$e ::= v \mid e\ e \mid e\ [\tau] \mid \mathbf{\mathtt{return}}\ e \mid \mathbf{\mathtt{async*}}\ e \mid \mathbf{\mathtt{get*}}\ e \mid \mathbf{\mathtt{then*}}(e, e) \mid \Box\, e \mid \mathbf{\mathtt{unbox}}\, e$$
$$v ::= c \mid x \mid f \mid \lambda x.e \mid \lambda X.e \mid \Box\, v$$

Expressions are values (v), application ($e\ e$), type application ($e\ [\tau]$), the return of expressions (**`return`** e), spawning an asynchronous task returning a data-flow future (**`async*`** e), blocking on the fulfilment of a data-flow future (**`get*`** e) and future chaining to attach a callback on a future to be executed on the future's fulfilment (**`then*`**(e, e)). To support the encoding of control-flow futures, a lifting operation that we call boxing is introduced ($\Box\, e$) together with a dual unboxing operation (**`unbox`** e). Values are constants, variables, data-flow futures, abstraction, and type abstraction. Additionally, a value may be boxed ($\Box\, v$).

Static Semantics. The type system has the common types except the control-flow future type (**`Fut`** τ). In its stead, we use a type encoded in terms of data-flow futures, $\Box\, \tau$. We show explicit flattening rules for the encodings of control-flow futures in terms of data-flow futures.

Types: $\tau ::= \mathcal{K} \mid \tau \to \tau \mid X \mid \forall X.\tau \mid \mathbf{\mathtt{Flow}}\, \tau \mid \Box\, \tau$

Previous flattening rules and: $\downarrow \Box\, \tau = \Box \downarrow \tau$

$$\text{(TF-Env)}\ \frac{}{\vdash \epsilon} \qquad \text{(TF-EnvExpr)}\ \frac{x \notin dom(\Gamma) \quad \Gamma \vdash \tau}{\vdash \Gamma, x : \tau} \qquad \text{(TF-EnvVar)}\ \frac{X \notin dom(\Gamma) \quad \vdash \Gamma}{\vdash \Gamma, X} \qquad \text{(TF-K)}\ \frac{\vdash \Gamma}{\Gamma \vdash \mathcal{K}}$$

$$\text{(TF-Flow)}\ \frac{\Gamma \vdash \tau \quad \tau \neq \mathtt{Flow}\,\tau'}{\Gamma \vdash \mathtt{Flow}\,\tau} \qquad \text{(TF-Arrow)}\ \frac{\Gamma \vdash \tau \quad \Gamma \vdash \tau'}{\Gamma \vdash \tau \to \tau'} \qquad \text{(TF-X)}\ \frac{X \in \Gamma \quad \vdash \Gamma}{\Gamma \vdash X} \qquad \text{(TF-Forall)}\ \frac{\Gamma, X \vdash \tau}{\Gamma \vdash \forall X.\tau} \qquad \text{(Box)}\ \frac{\Gamma \vdash \tau}{\Gamma \vdash \Box\,\tau}$$

Figure 7 Type formation rules where $\Gamma ::= \epsilon \mid \Gamma, x : \tau \mid \Gamma, X$.

$$\text{(T-Constant)}\ \frac{c \text{ has type } \mathcal{K} \quad \Gamma \vdash \mathcal{K}}{\Gamma \vdash_\rho c : \mathcal{K}} \qquad \text{(T-Variable)}\ \frac{x : \tau \in \Gamma \quad \vdash \Gamma}{\Gamma \vdash_\rho x : \tau} \qquad \text{(T-Flow)}\ \frac{f : \mathtt{Flow}\,\tau \in \Gamma \quad \vdash \Gamma}{\Gamma \vdash_\rho f : \downarrow\mathtt{Flow}\,\tau} \qquad \text{(T-ValFlow)}\ \frac{\Gamma \vdash_\rho e : \tau}{\Gamma \vdash_\rho e : \downarrow\mathtt{Flow}\,\tau}$$

$$\text{(T-Return)}\ \frac{\Gamma \vdash_\tau e : \tau \quad \tau \neq \bullet \quad \Gamma \vdash \tau'}{\Gamma \vdash_\tau \mathtt{return}\ e : \tau'} \qquad \text{(T-Abstraction)}\ \frac{\Gamma, x : \tau \vdash_\bullet e : \tau'}{\Gamma \vdash_\rho \lambda x.e : \tau \to \tau'} \qquad \text{(T-Box)}\ \frac{\Gamma \vdash_\rho e : \tau}{\Gamma \vdash_\rho \Box\, e : \Box\,\tau} \qquad \text{(T-Unbox)}\ \frac{\Gamma \vdash_\rho e : \Box\,\tau}{\Gamma \vdash_\rho \mathtt{unbox}\, e : \tau}$$

$$\text{(T-Application)}\ \frac{\Gamma \vdash_\rho e_1 : \tau \to \tau' \quad \Gamma \vdash_\rho e_2 : \tau}{\Gamma \vdash_\rho e_1\, e_2 : \tau'} \qquad \text{(T-TypeAbstraction)}\ \frac{\Gamma, X \vdash_\bullet e : \tau}{\Gamma \vdash_\rho \Lambda X.e : \downarrow\forall X.\tau} \qquad \text{(T-TypeApplication)}\ \frac{\Gamma, X \vdash_\rho e : \forall X.\tau'}{\Gamma \vdash_\rho e\,[\tau] : \downarrow\tau'[\tau/X]}$$

$$\text{(T-AsyncStar)}\ \frac{\Gamma \vdash_\tau e : \tau}{\Gamma \vdash_\rho \mathtt{async*}\ e : \downarrow\mathtt{Flow}\,\tau} \qquad \text{(T-GetStar)}\ \frac{\Gamma \vdash_\rho e : \mathtt{Flow}\,\tau}{\Gamma \vdash_\rho \mathtt{get*}\ e : \tau} \qquad \text{(T-ThenStar)}\ \frac{\Gamma \vdash_\rho e_1 : \mathtt{Flow}\,\tau' \quad \Gamma \vdash_\tau e_2 : \tau' \to \tau}{\Gamma \vdash_\rho \mathtt{then*}(e_1, e_2) : \downarrow\mathtt{Flow}\,\tau}$$

Figure 8 Typing of expressions where futures are encoded as $\mathtt{Fut}\,\tau \mathrel{\widehat{=}} \Box\,\mathtt{Flow}\,\tau$.

Well-Typed Expressions. The type formation rules are given in Figure 7 and the typing rules are given in Figure 8. In places where a **return** may appear, ρ is some τ, the return type of the task, ρ, otherwise $\bullet$, which makes **return** ill-typed. This (or something equivalent) is necessary – otherwise passing a lambda that contains a **return** to another task might change the return type of the *task*, not of the expression.

The type rules consist of the common System F typing rules: typing of a constant (T-Constant), typing variables (T-Variable), the abstraction typing rule (T-Abstraction) that sets the return type of the task to $\bullet$, preventing **return** in lambdas, and application (T-Application). Type abstraction and application are the common ones with the distinctive flattening of the types (T-TypeAbstraction and T-TypeApplication). The rules regarding $\mathtt{Flow}\,\tau$ types state that an expression of type τ can be lifted to a $\mathtt{Flow}\,\tau$ (T-ValFlow), spawning a task returns a data-flow future type and the spawned task sets its returned type to that of the expression running asynchronously (T-AsyncStar). The constructs **get*** e returns the content of the data-flow future (T-GetStar). Chaining on a data-flow future adds a callback to expression e_1, returning immediately a new data-flow future (T-ThenStar). Control-flow futures are encoded in terms of data-flow futures with the $\Box\, e$ operator with type $\Box\,\tau$, where $\mathtt{Fut}\,\tau \mathrel{\widehat{=}} \Box\,\tau$.

Dynamic Semantics. Configurations are as in the previous section, except using control-flow futures. Thus, the initial program configuration is $(fut_{f_{main}})\ (task_{f_{main}}\ e)$, where f_{main} is fulfilled by the result of e at the end of execution. The dynamic semantics are formulated

$$\textsc{(R-}\beta\textsc{)}\quad (\mathit{task}_f\ E[\lambda x.e\ v]) \rightarrow (\mathit{task}_f\ E[e[v/x]])$$

$$\textsc{(R-TypeApplication)}\quad (\mathit{task}_f\ E[(\Lambda X.e)\ [\tau]]) \rightarrow (\mathit{task}_f\ E[e[\tau/X]])$$

$$\textsc{(R-GetStar)}\quad \frac{\mathit{isflow?}(g)}{(\mathit{task}_f\ E[\mathtt{get*}\ g])\ (\mathit{flow}_g\ v) \rightarrow (\mathit{task}_f\ E[v])\ (\mathit{flow}_g\ v)}$$

$$\textsc{(R-GetVal)}\quad \frac{\neg\mathit{isflow?}(v)}{(\mathit{task}_f\ E[\mathtt{get*}\ v]) \rightarrow (\mathit{task}_f\ E[v])}$$

$$\textsc{(R-AsyncStar)}\quad \frac{\mathit{fresh}\ f}{(\mathit{task}_g\ E[\mathtt{async*}\ e]) \rightarrow (\mathit{flow}_f)\ (\mathit{task}_f\ e)\ (\mathit{task}_g\ E[f])}$$

$$\textsc{(R-Return)}\quad (\mathit{task}_f\ E[\mathtt{return}\ v]) \rightarrow (\mathit{task}_f\ v)$$

$$\textsc{(R-ChainRunFlow)}\quad (\mathit{chain}_g\ f\ e)\ (\mathit{flow}_f\ v) \rightarrow (\mathit{task}_g\ (e\ v))\ (\mathit{flow}_f\ v)$$

$$\textsc{(R-FulfilFlowValue)}\quad \frac{\neg\mathit{isflow?}(v)}{(\mathit{task}_f\ v)\ (\mathit{flow}_f) \rightarrow (\mathit{flow}_f\ v)}$$

$$\textsc{(R-ChainVal)}\quad \frac{\neg\mathit{isflow?}(v) \quad \mathit{fresh}\ g}{(\mathit{task}_f\ E[\mathtt{then*}(v, \lambda x.e)]) \rightarrow (\mathit{flow}_g)\ (\mathit{task}_g\ (\lambda x.e)\ v)\ (\mathit{task}_f\ E[g])}$$

$$\textsc{(R-FulfilFlow)}\quad \frac{\mathit{isflow?}(g)}{(\mathit{task}_f\ g) \rightarrow (\mathit{chain}_f\ g\ \lambda x.x)}$$

$$\textsc{(R-ChainFlow)}\quad \frac{\mathit{isflow?}(h) \quad \mathit{fresh}\ g}{(\mathit{task}_f\ E[\mathtt{then*}(h, \lambda x.e)]) \rightarrow (\mathit{flow}_g)\ (\mathit{chain}_g\ h\ \lambda x.e)\ (\mathit{task}_f\ E[g])}$$

$$\textsc{(R-Unbox)}\quad (\mathit{task}_f\ E[\mathtt{unbox}\,(\Box\, v)]) \rightarrow (\mathit{task}_f\ E[v])$$

Figure 9 Run-time semantics.

as a small-step operational semantics with reduction-based, contextual rules for evaluation within tasks. Evaluation contexts E contain a hole $\bullet$ that denotes the location of the next reduction step [40].

$$\begin{array}{lcl} E & ::= & \bullet \mid E\ e \mid v\ E \mid \mathtt{return}\ E \mid \mathtt{get*}\ E \mid \mathtt{then*}(E, e) \mid \mathtt{then*}(v, E) \\ & \mid & \Box\, E \mid \mathtt{unbox}\, E \mid E\ [\tau] \end{array}$$

The reduction rules (Figure 9) are the common β-reduction and type application from System F. The blocking operation $\mathtt{get*}\ v$ performs a run-time check to test whether the value v is a data-flow future or simply a value lifted to one. If it is a data-flow future, the value is extracted (R-GetStar); in case of a value, it is left in place (R-GetVal). Spawning a task creates a fresh data-flow future and task with a new task identifier, and the operation returns immediately the created future (R-AsyncStar). Returning from a task just throws away the execution context (R-Return), so that the task can fulfil its associated future in the next step. This next step depends on whether the value that fulfils the task is a future or a concrete value. If the task finishes with a data-flow future, the run-time chains the returned future to the identity function. This causes the value from the returned future to propagate to the current-in-call future (R-FulfilFlow). If the return value of a task is not a data-flow future, then this simply fulfils the current-in-call future (R-FulfilFlowValue). A chained configuration waits until the dependent data-flow future is fulfilled, then it executes the callback associated with it (R-ChainRunFlow). Expression-level chaining on data-flow futures checks at run-time whether target of the chain operation on is a data-flow future or a lifted value. In the former case, it lifts the chaining from the expression to the configuration level, returning immediately a new data-flow future (R-ChainFlow). In the latter case, chaining creates a new task to apply the chained function (R-ChainVal). The reason for spawning a new task is to preserve consistent behaviour across chaining on fulfilled and unfulfilled futures. If chaining on a fulfilled future executed immediately, and synchronously, we would increase the latency of the current task, or – if FlowFut is implemented in a language with mutable state – potentially introduce a race condition as it is unclear whether a chained lambda function executes directly (and synchronously) or not. This design saves a programmer from such potential hassles.

The unboxing operator unpacks the boxed value (R-Unbox). It is important for encoding of control-flow futures in terms of data-flow futures, described in the upcoming section. Boxed values will be introduced in conjunction with the encoding.

Extending FlowFut with Control-Flow Futures. In this section we show how to extend the language with control-flow futures encoded in terms of data-flow futures. Operations on data-flow futures transparently traverse any number of (invisible-from-the-typing) nested data-flow futures until they reach a concrete value or a control-flow future. The inclusion of the boxed values allow us to straightforwardly encode $\texttt{Fut}\,\tau$ thus: $\texttt{Fut}\,\tau \mathrel{\hat{=}} \square\,\texttt{Flow}\,\tau$. Using this encoding, we extend FlowFut with equi-named operations on control-flow futures, dropping the $\star$ for clarity. It is straightforward to encode each operation using its corresponding $\star$-version combined together with $\square$ and **unbox**:

$$\texttt{get}\ e \mathrel{\hat{=}} \texttt{get*}\ (\texttt{unbox}\ e) \qquad \texttt{then}(e, e') \mathrel{\hat{=}} \square\,\texttt{then*}(\texttt{unbox}\ e, e') \qquad \texttt{async}\ e \mathrel{\hat{=}} \square\,\texttt{async*}\ e$$

A control-flow future is always a boxed value, where the value can be anything including another future (data-flow or control-flow), or a concrete value. To perform control-flow future operations, one always needs to unpack the box and use its equivalent data-flow future operator. When an operator returns a new control-flow future (chaining and spawning a task), the return value needs to be boxed again.

Similarly, we extend FlowFut with type rules for these operations. These are the same as their $\star$-versions except that they use control-flow future types. Chaining takes a control-flow future and a function acting as callback and returns immediately a new control-flow future (T-Then). Spawning a task returns immediately a control-flow future (T-Async). Blocking access on a control-flow future returns the value inside the future (T-Get).

$$\begin{array}{c}\text{(T-Then)}\\ \dfrac{\Gamma \vdash_\rho e_1 : \texttt{Fut}\,\tau' \qquad \Gamma \vdash_\rho e_2 : \tau' \to \tau}{\Gamma \vdash_\rho \texttt{then}(e_1, e_2) : \texttt{Fut}\,\tau}\end{array} \qquad \begin{array}{c}\text{(T-Async)}\\ \dfrac{\Gamma \vdash_\tau e : \tau}{\Gamma \vdash_\rho \texttt{async}\ e : \texttt{Fut}\,\tau}\end{array} \qquad \begin{array}{c}\text{(T-Get)}\\ \dfrac{\Gamma \vdash_\rho e : \texttt{Fut}\,\tau}{\Gamma \vdash_\rho \texttt{get}\ e : \tau}\end{array}$$

Because data-flow futures do not allow observing completion of individual stages of an operation returning a nested future, we design our system to always "forward-compress" the return value of a flow, meaning we treat **return** of data-flow futures implicitly as a **forward** from [13], which addresses the Future Proliferation Problem. This brings us to the final extension of FlowFut with support for **forward**. Forwarding a control-flow future is just unpacking it and returning it, whereas forwarding a data-flow future is equivalent to **return**:

$$\texttt{forward}\ e \mathrel{\hat{=}} \texttt{return}\ (\texttt{unbox}\ e) \qquad\qquad \texttt{forward*}\ e \mathrel{\hat{=}} \texttt{return}\ e$$

And the type rules are straightforward: (Note that τ' can be any well-formed type as the expression will not have a usual return type, but instead finish the enclosing task.)

$$\begin{array}{c}\text{(T-Forward)}\\ \dfrac{\Gamma \vdash \tau' \qquad \Gamma \vdash_\tau e : \texttt{Fut}\,\tau}{\Gamma \vdash_\tau \texttt{forward}\ e : \tau'}\end{array} \qquad \begin{array}{c}\text{(T-Forward-Star)}\\ \dfrac{\Gamma \vdash \tau' \qquad \Gamma \vdash_\tau e : \texttt{Flow}\,\tau}{\Gamma \vdash_\tau \texttt{forward*}\ e : \tau'}\end{array}$$

▶ **Theorem** (Progress for FlowFut)**.** *Given a global configuration config, if* $\Gamma \vdash$ *config ok, then config is a terminal configuration or there exists a config*$'$ *such that config* $\rightarrow$ *config*$'$*.*

▶ **Theorem** (Preservation for FlowFut)**.** *Given a global configuration config, if* $\Gamma \vdash$ *config ok and config* $\rightarrow$ *config*$'$*, then there exists a* Γ' *such that* $\Gamma' \supseteq \Gamma$ *and* $\Gamma' \vdash$ *config*$'$ *ok.*

Proof. Both proofs are by induction on the derivation of the shape of *config*. ◀

We now move to FutFlow, which has control-flow futures as its primitive form of future. Section 5 revisits FlowFut and FutFlow, and discusses optimisation and implementation issues.

4.3 FutFlow: Primitive Control-Flow and Encoded Data-Flow Futures

A large number of programming languages implement control-flow explicit futures, natively (e.g., ABS [22], Encore [7], Joelle [10]) or through standard or third-party libraries (e.g., Java [32, 17], Akka [41]). In this section we explain, through the calculus FutFlow, how to extend such a semantic model to also encompass control-flow futures and delegation. Thus, in contrast to FlowFut, we now encode data-flow futures in terms of control-flow future types.

The calculus (omitting operations on data-flow futures) contains expressions and values:

$$\begin{array}{lcl} e & ::= & v \mid e\,e \mid e\,[\tau] \mid \texttt{return}\,e \mid \texttt{async}\,e \mid \texttt{get}\,e \mid \texttt{then}(e,e) \mid \texttt{forward}\,e \\ v & ::= & c \mid x \mid f \mid \lambda x.e \mid \lambda X.e \end{array}$$

The key difference to FlowFut is the inclusion of `forward` as a primitive.

Static Semantics. The type system has the following types: the common basic types ($\mathcal{K}$), abstraction ($\tau \rightarrow \tau$), type variables (X), universal quantification ($\forall X.\tau$), and control-flow future types ($\texttt{Fut}\,\tau$).

Types: $\quad \tau ::= \mathcal{K} \mid \tau \rightarrow \tau \mid X \mid \forall X.\tau \mid \texttt{Fut}\,\tau$

Well-Typed Expressions. The type formation rules are given in Figure 10 and expression typing is shown in Figure 11 – similar to Section 4.2. Most rules should be straightforward and have appeared before in similar form. Note lack of $\star$ on operators to highlight the control-flow nature.

$$\overset{\text{(TF-Env)}}{\frac{}{\vdash \epsilon}} \qquad \overset{\text{(TF-EnvExpr)}}{\frac{x \notin dom(\Gamma) \qquad \Gamma \vdash \tau}{\vdash \Gamma, x : \tau}} \qquad \overset{\text{(TF-EnvVar)}}{\frac{X \notin dom(\Gamma) \qquad \vdash \Gamma}{\vdash \Gamma, X}} \qquad \overset{\text{(TF-K)}}{\frac{\vdash \Gamma}{\Gamma \vdash \mathcal{K}}}$$

$$\overset{\text{(TF-Fut)}}{\frac{\Gamma \vdash \tau}{\Gamma \vdash \texttt{Fut}\,\tau}} \qquad \overset{\text{(TF-Arrow)}}{\frac{\Gamma \vdash \tau \qquad \Gamma \vdash \tau'}{\Gamma \vdash \tau \rightarrow \tau'}} \qquad \overset{\text{(TF-X)}}{\frac{X \in \Gamma \qquad \vdash \Gamma}{\Gamma \vdash X}} \qquad \overset{\text{(TF-Forall)}}{\frac{\Gamma, X \vdash \tau}{\Gamma \vdash \forall X.\tau}}$$

Figure 10 Type formation rules where $\Gamma ::= \epsilon \mid \Gamma, x : \tau \mid \Gamma, X$.

$$\frac{c \text{ has type } \mathcal{K} \qquad \Gamma \vdash \mathcal{K}}{\Gamma \vdash_\rho c : \mathcal{K}}\ (\text{T-Constant}) \qquad \frac{x : \tau \in \Gamma \qquad \vdash \Gamma}{\Gamma \vdash_\rho x : \tau}\ (\text{T-Variable}) \qquad \frac{f : \texttt{Fut}\,\tau \in \Gamma \qquad \vdash \Gamma}{\Gamma \vdash_\rho f : \texttt{Fut}\,\tau}\ (\text{T-Fut})$$

$$\frac{\Gamma \vdash_\tau e : \tau \qquad \tau \neq \bullet \qquad \Gamma \vdash \tau'}{\Gamma \vdash_\tau \texttt{return}\ e : \tau'}\ (\text{T-Return}) \qquad \frac{\Gamma,\ x : \tau \vdash_\bullet e : \tau'}{\Gamma \vdash_\rho \lambda x.e : \tau \to \tau'}\ (\text{T-Abstraction}) \qquad \frac{\Gamma \vdash_\rho e_1 : \tau \to \tau' \qquad \Gamma \vdash_\rho e_2 : \tau}{\Gamma \vdash_\rho e_1\ e_2 : \tau'}\ (\text{T-Application})$$

$$\frac{\Gamma \vdash \tau' \qquad \tau \neq \bullet \qquad \Gamma \vdash_\tau e_1 : \texttt{Fut}\,\tau}{\Gamma \vdash_\tau \texttt{forward}\ e_1 : \tau'}\ (\text{T-Forward}) \qquad \frac{\Gamma,\ X \vdash_\bullet e : \tau}{\Gamma \vdash_\rho \Lambda X.e : \forall X.\tau}\ (\text{T-TypeAbstraction}) \qquad \frac{\Gamma, X \vdash_\rho e : \forall X.\tau'}{\Gamma \vdash_\rho e\ [\tau] : \tau'[\tau/X]}\ (\text{T-TypeApplication})$$

$$\frac{\Gamma \vdash_\rho e_1 : \texttt{Fut}\,\tau' \qquad \Gamma \vdash_\tau e_2 : \tau' \to \tau}{\Gamma \vdash_\rho \texttt{then}(e_1, e_2) : \texttt{Fut}\,\tau}\ (\text{T-Then}) \qquad \frac{\Gamma \vdash_\tau e : \tau}{\Gamma \vdash_\rho \texttt{async}\ e : \texttt{Fut}\,\tau}\ (\text{T-Async}) \qquad \frac{\Gamma \vdash_\rho e : \texttt{Fut}\,\tau}{\Gamma \vdash_\rho \texttt{get}\ e : \tau}\ (\text{T-Get})$$

Figure 11 Typing of expressions.

$$(\textit{task}_f\ E[\lambda x.e\ v]) \to (\textit{task}_f\ E[e[v/x]])\ (\text{R-}\beta) \qquad \frac{\textit{fresh } f}{(\textit{task}_g\ E[\texttt{async}\ e]) \to (\textit{fut}_f)\ (\textit{task}_f\ e)\ (\textit{task}_g\ E[f])}\ (\text{R-Async})$$

$$(\textit{chain}_g\ f\ e)\ (\textit{fut}_f\ v) \to (\textit{task}_g\ (e\ v))\ (\textit{fut}_f\ v)\ (\text{R-ChainRunFut}) \qquad (\textit{task}_f\ E[\texttt{get}\ h])\ (\textit{fut}_h\ v) \to (\textit{task}_f\ E[v])\ (\textit{fut}_h\ v)\ (\text{R-Get})$$

$$(\textit{task}_f\ v)\ (\textit{fut}_f) \to (\textit{fut}_f\ v)\ (\text{R-FutFulfilValue}) \qquad (\textit{task}_f\ E[(\Lambda X.e)\ [\tau]]) \to (\textit{task}_f\ E[e[\tau/X]])\ (\text{R-TypeApplication})$$

$$(\textit{task}_f\ E[\texttt{forward}\ h]) \to (\textit{chain}_f\ h\ \lambda x.x)\ (\text{R-Forward}) \qquad (\textit{task}_f\ E[\texttt{return}\ v])\ (\textit{fut}_f) \to (\textit{fut}_f\ v)\ (\text{R-Return})$$

$$\frac{\textit{fresh } g}{(\textit{task}_f\ E[\texttt{then}(h, \lambda x.e)]) \to (\textit{fut}_g)\ (\textit{chain}_g\ h\ \lambda x.e)\ (\textit{task}_f\ E[g])}\ (\text{R-Then})$$

Figure 12 Run-time semantics.

Operational semantics. The operational semantics are similar to Section 4.2. Evaluation contexts E contain a hole $\bullet$ that denotes where the next reduction step happens [40]:

$$E ::= \bullet \mid E\ e \mid v\ E \mid \texttt{then}(E, e) \mid \texttt{forward}\ E \mid E\ [\tau] \mid \texttt{get}\ E \mid \texttt{return}\ E$$

The reduction rules are similar to the FlowFut calculus, but work on control-flow futures (Figure 12). Beta reduction works in the traditional fashion. The `async` construct spawns a new task to execute the given expression, and creates a new control-flow future to store its result (R-Async). A chained configuration runs as soon as the dependent future is fulfilled and passes the content of the fulfilled future to the callback expression, running the pending computation on demand (R-ChainRunFut). Getting a value out of a future blocks the execution until the future is fulfilled (R-Get). Tasks fulfil their implicit future implicitly, when there are no more pending expressions to run, or explicitly via the return expression

(R-FutFulfilValue and R-Return). So far, most **forward** examples have avoided future nesting by reusing the current-in-call future with a following asynchronous operation. This avoids creation of an additional future, meaning nesting is not possible. It is also possible to use **forward** to fulfil one existing future with the result of another without nesting or blocking the current computation: **forward** h fulfils the current-in-call future with the value in h by throwing away the remainder of the body of the current task and chaining the identity function on h. This has the effect of copying the eventual result stored in h into the current future (R-Forward). Chaining an expression on a future results immediately in a new future that will eventually contain the result of evaluating the expression, and a chain configuration storing the expression is connected with the original future (R-Then).

Extending FutFlow with Data-Flow Futures. In this section we show, first, the language extensions necessary for encoding data-flow futures in terms of control-flow futures and, second, the encodings.

We extend the calculus with the following expressions and values:

$$e \quad ::= \quad \dots \mid \mathtt{match}(x : e, x : e, e) \mid \blacklozenge e \qquad\qquad v \quad ::= \quad \dots \mid \blacklozenge v$$

Expressions can now use a pattern matching operation, which is a common programming construct [31, 24]. To encode data-flow futures, we define a boxing operation ($\blacklozenge e$) which uses pattern matching for unboxing. To keep constructs simple, **match** and $\blacklozenge$ are not intended as source-level constructs, so $\blacklozenge$ only works on data-flow futures in the formalism.

The type system has the previous common types with the addition of the $\blacklozenge$ type – used later for encoding data-flow futures in terms of control-flow futures. As in FlowFut, we show explicit flattening rules for the encodings of data-flow futures:

Types: $\tau ::= \dots \mid \blacklozenge\,\mathtt{Fut}\,\tau$

Previous flattening rules and: $\downarrow\blacklozenge\,\mathtt{Fut}\,(\blacklozenge\,\mathtt{Fut}\,\tau) = \downarrow\blacklozenge\,(\mathtt{Fut}\,\tau)$

$\downarrow\blacklozenge\,\mathtt{Fut}\,\tau = \blacklozenge\,(\downarrow\mathtt{Fut}\,\tau) \quad \tau \neq \blacklozenge\,\mathtt{Fut}\,\tau'$

Additional type rules for these constructs are found below. Notice how the introduction of the explicit flattening rules require the update of two typing rules (T-TypeAbstraction and T-TypeApplication). This is necessary to flatten $\blacklozenge\,\mathtt{Fut}\,\tau$ types.

$$\text{(Dia)}\ \frac{\Gamma \vdash \tau}{\Gamma \vdash \blacklozenge\,\tau} \qquad \text{(T-TypeAbstraction)}\ \frac{\Gamma,\ X\ \vdash_{\bullet}\ e : \tau}{\Gamma \vdash_\rho \lambda X.e : \downarrow\forall X.\tau} \qquad \text{(T-TypeApplication)}\ \frac{\Gamma, X \vdash_\rho e : \downarrow\forall X.\tau'}{\Gamma \vdash_\rho e\,[\tau] : \downarrow\tau'[\tau/X]}$$

$$\text{(T-Match)}\ \frac{\begin{array}{c}\Gamma, x : \tau \vdash_\rho e_1 : \tau' \\ \Gamma, x : \mathtt{Fut}\,\tau \vdash_\rho e_2 : \tau' \qquad \Gamma \vdash_\rho e_3 : \downarrow\blacklozenge\,\mathtt{Fut}\,\tau\end{array}}{\Gamma \vdash_\rho \mathtt{match}(x : e_1, x : e_2, e_3) : \tau'} \qquad \text{(T-ValFlow)}\ \frac{\Gamma \vdash_\rho e : \tau}{\Gamma \vdash_\rho e : \downarrow\blacklozenge\,\mathtt{Fut}\,\tau} \qquad \text{(T-FlowFut)}\ \frac{\Gamma \vdash_\rho e : \mathtt{Fut}\,\tau}{\Gamma \vdash_\rho \blacklozenge\,e : \downarrow\blacklozenge\,\mathtt{Fut}\,\tau}$$

The **match** construct has two open terms as first and second arguments, the free variables are captured at the declaration site; the third argument is a data-flow future type argument. The first argument is applied if the data-flow future type is actually a value and the second argument is applied to the value of the data-flow future type if the type was lifted from a control-flow future. Essentially, **match** pattern matches on the form of the data-flow future type. An expression of type τ can be lifted to $\blacklozenge\,\mathtt{Fut}\,\tau$ (T-ValFlow and T-FlowFut).

The dynamic semantics include now the pattern matching operation, which performs beta reduction based on the form of the value v (R-Match-Val and R-Match-Fut).

The introduction of the boxing value – used to encode data-flow futures – requires special care when fulfilling of a task. This is reflected in the updated rule R-FutFulfilValue and on R-FlowCompression. If the value is not a data-flow future, i.e., $v \neq \blacklozenge v'$, then the value fulfils the task's future; if the value is a data-flow future, then it builds a chained configuration to ultimately pull the value out, running the identity function.

$$\text{(R-FutFulfilValue)}\quad \frac{v \neq \blacklozenge v'}{(\mathit{task}_f\ v)\ (\mathit{fut}_f) \rightarrow (\mathit{fut}_f\ v)}$$

$$\text{(R-MatchVal)}\quad (\mathit{task}_f\ E[\texttt{match}(x : e_1, x : e_2, v)]) \rightarrow (\mathit{task}_f\ E[e_1[v/x]])$$

$$\text{(R-FlowCompression)}\quad (\mathit{task}_f\ \blacklozenge g) \rightarrow (\mathit{chain}_f\ g\ \lambda x.x)$$

$$\text{(R-MatchFut)}\quad (\mathit{task}_f\ E[\texttt{match}(x : e_1, x : e_2, \blacklozenge g)]) \rightarrow (\mathit{task}_f\ E[e_2[g/x]])$$

With these new constructs, we can encode data-flow futures in terms of control-flow futures: $\texttt{Flow}\,\tau \mathrel{\widehat{=}} \blacklozenge\,\texttt{Fut}\,\tau$. The term $\blacklozenge\, e$ captures the lifting of a control-flow future value to $\texttt{Flow}\,\tau$ (T-FlowFut). All operators on data-flow futures are encoded in terms of primitive operators:

$$\begin{aligned}\texttt{async*}\ e &\mathrel{\widehat{=}} \blacklozenge\,\texttt{async}\ e\\ \texttt{get*}\ e &\mathrel{\widehat{=}} \texttt{match}(x : x, x : \texttt{get}\ x, e)\end{aligned}$$

$$\begin{aligned}\texttt{then*}(e, \mathit{fn}) &\mathrel{\widehat{=}} \texttt{match}(x : \mathit{fn}\ x, f : \blacklozenge\,\texttt{then}(f, \mathit{fn}), e)\\ \texttt{forward*}\ e &\mathrel{\widehat{=}} \texttt{match}(x : \texttt{return}\ x, f : \texttt{forward}\ f, e)\\ \texttt{undiamond}\, e &\mathrel{\widehat{=}} \texttt{get*}\ e\end{aligned}$$

The typing rules for operations on data-flow futures are expressed as an extension to the typing rules of Figure 11. A data-flow future type can be "unlifted" so that we extract its internal value (T-Undiamond). Any expression can be lifted from some τ or from a control-flow future type to a data-flow future (rules T-Flow and T-FlowFut).

$$\text{(T-Async-Star)}\quad \frac{\Gamma \vdash_\rho e : \tau}{\Gamma \vdash_\rho \texttt{async*}\ e : \downarrow\!\texttt{Flow}\,\tau}$$

$$\text{(T-Get-Star)}\quad \frac{\Gamma \vdash_\rho e : \downarrow\!\texttt{Flow}\,\tau}{\Gamma \vdash_\rho \texttt{get*}\ e : \tau}$$

$$\text{(T-Then-Star)}\quad \frac{\Gamma \vdash_\rho e_1 : \downarrow\!\texttt{Flow}\,\tau' \quad \Gamma \vdash_\rho e_2 : \tau' \rightarrow \tau}{\Gamma \vdash_\rho \texttt{then*}(e_1, e_2) : \downarrow\!\texttt{Flow}\,\tau}$$

$$\text{(T-Forward-Star)}\quad \frac{\Gamma \vdash_\tau e : \downarrow\!\texttt{Flow}\,\tau}{\Gamma \vdash_\tau \texttt{forward*}\ e : \tau'}$$

$$\text{(T-Undiamond)}\quad \frac{\Gamma \vdash_\rho e : \downarrow\!\blacklozenge\,\texttt{Fut}\,\tau}{\Gamma \vdash_\rho \texttt{undiamond}\, e : \tau}$$

This concludes the presentation of FutFlow. In the next section, we discuss optimisations in FlowFut and FutFlow, and implementation issues.

▶ **Theorem** (Progress for FutFlow). *Given a global configuration config, if* $\Gamma \vdash$ *config ok, then config is a terminal configuration or there exists a config′ such that config* $\rightarrow$ *config′.*

▶ **Theorem** (Preservation for FutFlow). *Given a global configuration config, if* $\Gamma \vdash$ *config ok and config* $\rightarrow$ *config′, then there exists a* Γ' *such that* $\Gamma' \supseteq \Gamma$ *and* $\Gamma' \vdash$ *config′ ok.*

Proof. Both proofs are by induction on the derivation of the shape of *config*. ◀

4.4 Godot's Solutions to Future Problems

We now revisit the problems of Section 2 and show how Godot addresses them.

The Type Proliferation Problem. Because of the data-flow future component, the Type Proliferation Problem is avoided. Like with DeF, the following statement is typeable (as is tail-recursive functions) and returns a `Flow[t]` (the equivalent of `Flow` τ in code examples):

```
async (if precomputed(v) then table.lookup(v) else worker ! compute(v))
```

Because data-flow futures allow implicitly lifting a concrete value of type `t` to `Flow[t]`, code using data-flow futures can be trivially reused with concrete values. This addresses the Type Proliferation Problem, allowing one data-flow future type to represent zero or many run-time futures. (See Figure 13 for additional details.)

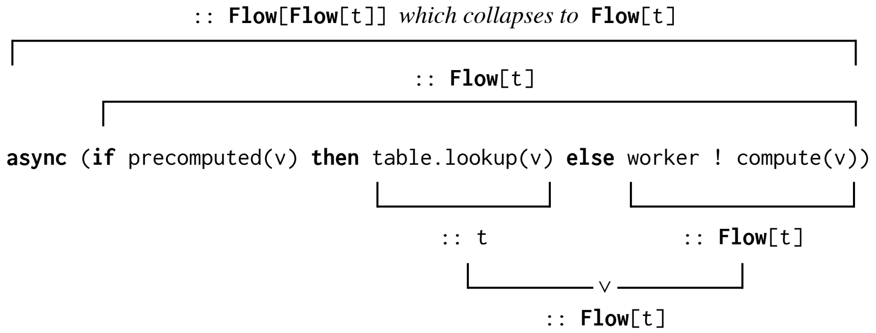

:: Flow[Flow[t]] which collapses to Flow[t]
:: Flow[t]
async (if precomputed(v) then table.lookup(v) else worker ! compute(v))
:: t
:: Flow[t]
∨
:: Flow[t]

Figure 13 Overcoming the Type Proliferation Problem. Compare with Figure 2.

The Future Proliferation Problem. Because of support for delegation, the Future Proliferation Problem is avoided – but in a way that improves on `forward`. Since data-flow futures abstract nesting, we can implicitly turn a `return` into a `forward` based on the return type, and not require the programmer to explicitly choose a forwarding solution. Thus, we can avoid the quirky looking `return` in one branch and `forward` in another, and simply write:

```
if precomputed(v) then return table.lookup(v) else return worker ! compute(v)
```

This allows us to hoist the `return` to write the original statement that would not type with explicit control-flow futures, while still avoiding creation of unnecessary futures:

```
return if precomputed(v) then table.lookup(v) else worker ! compute(v)
```

The Fulfilment Observation Problem. The integration of both kinds of futures in a single system avoids the Fulfilment Observation Problem by allowing a programmer to opt-in on control-flow futures where desirable, without imposing a one-size-fits-all solution. The following function definition uses explicit control-flow futures to allow the observation of both stages – finding an idle worker and dispatching work to it and completing the job:

```
def perform(job : Job[t]) : Fut[t] { return idle_worker() ! do_work(job) }
var f = load_balancer ! perform(my_job) −−f is a control−flow future
get(f) −−block until do_work() has been called
```

In contrast, this function definition uses data-flow futures and therefore will *not* allow the distinction between the two stages, and its **return** will be treated as a **forward**:

```
def perform(job : Job[t]) : Flow[t] { return idle_worker() ! do_work(job) }
var f = load_balancer ! perform(my_job) −−f is a data−flow future
get(f) −−block until do_work() has finished
```

Notably, the integrated system also supports the nesting of *different kinds* of futures. For example, `Flow[Fut[Flow[t]]]` denotes a value computed by a pipeline of zero or more asynchronous operations whose individual completedness cannot be distinguished, followed by a control-flow future corresponding to a single operation *whose completedness can be observed*, followed by another pipeline of zero or more asynchronous operations.

Concluding Remarks. In addition to addressing all three problems of Section 2, Godot overcomes a limitation in the initial DeF proposal for data-flow explicit futures in [20] by adding support for *parametric polymorphism*. In fact, DeF did not study parametric polymorphism and it is not trivial to add, as standard techniques [33] prevent the collapsing of nested future types. For example, in DeF the following function $\texttt{problematic} = (\lambda X.\lambda y : X.\ \texttt{async*}\ y)$ has type $\forall X.X \to \texttt{Flow}\,X$ and, after type application $\texttt{problematic}\ [\texttt{Flow}\,\mathcal{K}] :: \texttt{Flow}\,(\texttt{Flow}\,\mathcal{K})$, which forces a programmer to insert multiple **get** operations to obtain a concrete value from a data-flow future, which breaks the DeF invariant that a single **get** is always enough to access a concrete value. In Godot, the `problematic` function after type application has type $\texttt{Flow}\,\mathcal{K}$, because typing rules normalise flow types and **get*** guarantees access to a concrete value.

Using Godot, a programmer can decide to abstract or expose details about how values are produced through asynchronous operations, by freely choosing between control-flow futures and data-flow futures or any combination thereof. And in the case of data-flow futures, profit from how Godot automatically avoids creating unnecessary (unobservable) futures. As the integration of control-flow futures, delegation, and data-flow futures improves the individual components (e.g., the support for parametric polymorphism with data-flow futures and type-driven automatic insertion of **forward**), *Godot is greater than the sum of its parts.* Moreover, as the next sections will show, it is possible to encode either kind of future in the other, which facilitates their implementation in a programming language. This realisation is an important aspect of our contributions, which extends beyond "taking the union."

This section has put in perspective Godot as solution to the Type Proliferation Problem, Future Proliferation Problem, and Fulfilment Observation Problem through the integration of control-flow futures and data-flow futures in an explicit system with support for implicit delegation. The previous sections 4.1–4.3 explain Godot in detail.

5 Discussion

The preceding two sections showed how to encode data-flow futures in a language that only provides control-flow futures, and the opposite; both approaches rely on small extensions of the type system and encodings of operations for one type of futures into the other. We review below the preceding results from an implementation and optimisation point of view.

5.1 Avoiding Future Nesting through Implicit Delegation

We revisit the example from the introduction to the Fulfilment Observation Problem (Section 2). We imagine that this method runs in the context of an actor that "load-balances" by farming out `jobs` to the worker returned from the `idle_worker()`. As discussed previously, this is a case where control-flow explicit futures insert an additional, possibly unwanted, future indirection due to the additional asynchronous call handing the work off to some worker.

```
def perform(job : Job[t]) : Flow[t] { return idle_worker() ! do_work(job) }
```

as the programmer declared the return type of `perform()` as `Flow[t]`, the implementation is less restricted (e.g., we can either delegate to a worker or return a cached result without the typing problems of Figure 2). The programmer is not interested in any intermediate stages of the computation, and we can compile the method body replacing **`return`** with **`forward`**. *This optimisation is crucial for making asynchronous tail-recursive calls run in constant space.*

We model this example and optimisation in FlowFut (or FutFlow) as if the body of `perform()` executes inside a task, whose final expression is a **`return async*`** with the body of `do_work()` inside it. We express this optimisation in FlowFut as follows (rule ReturnAsync):

$$\overset{\textsc{(ReturnAsync)}}{\frac{\textit{fresh } j}{(\textit{task}_f^i\ E[\texttt{return async*}\ e]) \rightarrow (\textit{task}_f^j\ e)}} \qquad \overset{\textsc{(ReturnThen)}}{\frac{\textit{isflow}?(g)}{(\textit{task}_f^i\ E[\texttt{return then*}(g, e)]) \rightarrow (\textit{chain}_f\ g\ e)}}$$

For clarity, we add identifiers i and j to the tasks to highlight that there is *no reuse of a task* (which models delegating work to another concurrent actor), but a *reuse of a future* (the semantics of **`forward`**). We apply a similar optimisation (rule ReturnThen) when returning the result of future chaining: task i delegates the fulfilment of f to the chain task, and the delegating task finishes (and is removed in the calculus).

How Nesting Causes False Fulfilment. As exemplified in the previous section, implicit delegation avoids the creation of nested futures. We now illustrate why false fulfilment can happen if we do not avoid future nesting. Consider the implementation of **`get*`** and suppose we had a non-optimised version of FlowFut that has nesting of futures. Formally, we would have a reduction rule – $(\textit{flow}_f)\ (\textit{task}_f\ g) \rightarrow (\textit{flow}_f\ g)$ – that fulfils a data-flow future with another data-flow future. As a consequence, **`get*`** must perform a run-time test, and branch on whether a future is fulfilled by a concrete value or another future:

$$\overset{\textsc{(R-GetStarFlow-Unopt)}}{\frac{\textit{isflow}?(g)}{(\textit{task}_f\ E[\texttt{get*}\ g])\ (\textit{flow}_g\ v) \rightarrow (\textit{task}_f\ E[\texttt{get*}\ v])\ (\textit{flow}_g\ v)}} \qquad \overset{\textsc{(R-GetStarVal-Unopt)}}{\frac{\neg\textit{isflow}?(g)}{(\textit{task}_f\ E[\texttt{get*}\ v'])\ \rightarrow (\textit{task}_f\ E[v'])}}$$

The key rule above is R-GetStarFlow-Unopt which shows that a **`get*`** yielding a future reduces to another **`get*`** , meaning we move to another possibly blocked state. This does not happen in FlowFut. Indeed, if a task returns a data-flow future g in a way that delegation could not elide, by R-FulfilFlowValue we use future chaining on g and tell g to fulfil f on its fulfilment instead having f effectively polling g through the implementation of **`get*`** .

5.2 Notes on Implementing Godot

Let us first consider the encodings. Implementing the encoding rules, either as a compilation phase or even as a library is pretty straightforward except from the following points.

1. Both encodings rely on the existence of a boxing construct. Such a construct can be easily encoded with a datatype or an object type. However due to the simplicity of the operations on boxes a native implementation could be more efficient.

2. The encoding of data-flow futures from control-flow ones requires a pattern matching operator (or equivalent) that can distinguish data-flow futures from other values, unless lifting actually creates a fulfilled data-flow future. Standard compiler optimisations are applicable here, such as synthesising different methods from a single specification, e.g., one applies only to values where future types are removed, one applies only to statically verified actual data-flow futures, and one for all other cases, or combinations.

Second, consider the type system extensions. In both cases, the type system of the language can be extended with the existing rules without major difficulty. Additionally, without modifying the type system if the data-flow future language has parametric types it seems possible to encode control-flow future typing rules: all typing rules necessary to extend the data-flow future language can be expressed as typing of parametric types and functions. In the other direction, it is slightly more involved as the type system of the control-flow future language must implement a form of type collapse rule.

Overall, extending a language with data-flow (resp. control-flow) futures to also support control-flow (resp. data-flow) futures raises no particular difficulty, and the encodings avoid adding "native support" for both futures. The compiler and the type checker need a small number of new, simple constructs. A data-flow future language may have to add chaining on data-flow futures, or the control-flow future language should add pattern matching that distinguishes data-flow futures. While all these extensions are minor, they will require some modicum of language modifications. Consequently, the implementation of Godot as an external library of a mainstream language is not straightforward: standard type systems do not perform the implicit lifting required for data-flow futures, and the future chaining required for data-flow futures rarely exist in control-flow futures. However, if a language with data-flow futures already supports chaining, which is a common operation, implementing control-flow futures in a non-intrusive manner – as an external library – seems to raise no difficulty.

6 Related Work

Section 3 discusses the most closely related work on data-flow explicit futures (DeF) [20] and future delegation [13] in detail. Earlier work on adapting a static analysis [16] from explicit to implicit futures revealed the difference between the future accesses, and a translation of control-flow synchronisation in ABS [22] to data-flow synchronisation in ProActive [21] showed that control-flow synchronisation could not be simulated purely by implicit futures.

Futures are means for expressing concurrency while enabling synchronisation at the latest possible time. They were first introduced by Baker and Hewitt in the 70's [4], and later rediscovered by Liskov and Shrira as Promises [27] and by Halstead in the context of MultiLisp [25]. Flanagan and Felleisen did an early formalisation of futures [15] based on MultiLisp's futures with focus on the difference between explicit and implicit future access. In a similar vein, $\lambda(\mathit{fut})$ [29] is a concurrent lambda calculus with futures with cells and handles. Futures in $\lambda(\mathit{fut})$ are explicitly created, similarly to MultiLisp. We now consider futures with respect to the dichotomy between implicit and explicit futures.

Implicit Futures. Implicit futures are indistinguishable from concrete values in source code. Typically, data-flow synchronisation is based on implicit futures. In MultiLisp [25], the future construct creates a thread and returns a future that can be manipulated by operations like assignment, that do not need a real value, but the program would *automatically block* when performing an operation requiring the value, i.e., futures are implicitly accessed but explicitly created. Similar constructs can be found in Alice [35], Oz-Mozart [38], and ProActive [3]. The latter is a Java library for active objects in which futures are implicit like in MultiLisp; futures are implemented with proxies that hide the future and provide a normal object interface, but accessing a proxy leads to a synchronisation with the availability of the object, i.e., the fulfilment of the future.

Explicit Futures. Explicit, typed futures appeared with ABCL/f [37] to represent the result of an asynchronous method invocation, paving the way for active object languages [12]. Explicit futures typically have a parametric future type, and exist in, e.g., Concurrent ML [34], C++ [26] and Java [39], often as libraries. Explicit futures open for different ways of synchronising. Hybrid [30] is an early language with *forwarding*; in this paper, Nierstrasz formulates a version of the *Future Proliferation Problem* of Section 2. Creol [23] features non-blocking polling on futures that enables *cooperative multi-threading* based on future availability. De Boer et al. [11] were probably among the first to offer a rich set of future manipulation primitives for control flow, together with a compositional proof theory. JCobox [36], ABS [22], and Encore have mostly reused these primitives [36, 22, 7]. Encore additionally supports creation and manipulation of parallel tasks as sets of futures [14].

Akka [19, 41] is a scalable actor library implemented on top of Java and Scala, in which futures are used either to allow actor messages to return a value or more automatically in the messages of *typed actors* (akin to active objects). The Akka programmer is advised to use asynchronous reaction on futures, i.e., register code to be executed when the future is fulfilled. Akka's `map` construct is similar to our **`then`** chaining construct. JavaScript promises are data-flow synchronised futures with explicit asynchronous access and no typing. The data-flow nature of the synchronisation distinguishes JavaScript from the other languages with explicit futures and is probably related to the absence of future type. Because it is untyped and promises are explicitly accessed and fulfilled, errors are frequently made when manipulating these promises; Madsen et al. [28] provide a powerful tool to study these errors.

Futures in the Mainstream. Many modern, statically typed programming languages provide control-flow futures through libraries to facilitate the creation and control of asynchronous, concurrent computations. We highlight Completable Futures [32], Listenable Futures [17], Scala Futures [18] and Akka Futures [1], and the `Observable` abstraction from the ReactiveX library [2], where asynchronous computations may return (*emit*) more than one value.

Future libraries of mainstream languages have considerably richer interfaces than the future abstractions in our core calculus, but, as far as we can tell, we provide all the necessary operations to construct most, if not all, of these library interfaces.

Extending existing libraries with support for data-flow futures is an interesting direction of future work. We take some preliminary steps in this direction in the companion artefact of this paper which shows how to integrate data-flow futures with Scala futures.

There is an analogy between futures and the observable abstraction from the ReactiveX library: both are control-flow constructs. Investigating whether the benefits of data-flow futures can be carried over to observables is an interesting future direction of this work.

Finally, most future libraries establish futures as monads, such as Akka Futures or the ReactiveX library. The control-flow futures from this paper are monadic, with **`async`** as its unit and **`get`** as its `join`. Data-flow futures are monadic as well, although they work on a smaller set of types, due to their implicit nature, i.e., they collapse **`Flow`** types.

7 Conclusion

The distinction between implicit and explicit futures is well-known, but recent work highlights that the relation between the typing and synchronisation discipline plays a more crucial aspect.

Following this observation, we identified three problems with existing future implementations: the Type Proliferation Problem restricts the expressiveness of control-flow futures; the Fulfilment Observation Problem limits the synchronisation capacities of data-flow futures; the Future Proliferation Problem makes both data-flow and control-flow futures inefficient. This paper defines Godot, a system supporting *both* data-flow and control-flow futures simultaneously, and in combination; our system is the first to do so, and also to solve the three problems above coherently in a single programming model. Godot shows how to add parametric polymorphism and automatic delegation for data-flow explicit futures, and demonstrates how to encode each type of future in terms of the other. This facilitates implementation of the full Godot system, or subsets, in existing programming languages, with or without support for futures. We believe that our formalisms communicate the core ideas, while not tying ourselves too closely to one particular kind of language or unit of concurrency.

While we developed two possible encodings, starting from a data-flow language seems a bit more promising. Indeed, if data-flow futures are the default, the non-expert programmer is only exposed to futures that do not suffer from Type Proliferation and where Future Proliferation can be avoided automatically. Programs that need control on future synchronisation, e.g., to implement load balancing or scheduling features, can use the encoding of control-flow futures and avoid the Fulfilment Observation Problem.

References

1 Akka Futures. `https://doc.akka.io/docs/akka/current/futures.html`, 2019.

2 Rx Extensions. `http://reactivex.io/`, 2019.

3 Laurent Baduel, Francoise Baude, Denis Caromel, Arnaud Contes, Fabrice Huet, Matthieu Morel, and Romain Quilici. *Programming, Composing, Deploying for the Grid*, pages 205–229. Springer London, London, 2006.

4 Henry G. Baker and Carl E. Hewitt. The Incremental Garbage Collection of Processes. In *Proc. Symposium on Artificial Intelligence Programming Languages*, number 12 in SIGPLAN Notices, page 11, August 1977.

5 Samuel Beckett. Waiting for Godot. *Samuel Beckett: The Complete Dramatic Works*, pages 7–89, 1954.

6 Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An Efficient Multithreaded Runtime System. *J. Parallel Distrib. Comput.*, 37(1):55–69, 1996. `doi:10.1006/jpdc.1996.0107`.

7 Stephan Brandauer, Elias Castegren, Dave Clarke, Kiko Fernandez-Reyes, Einar Broch Johnsen, Ka I Pun, Silvia Lizeth Tapia Tarifa, Tobias Wrigstad, and Albert Mingkun Yang. Parallel Objects for Multicores: A Glimpse at the Parallel Language Encore. In *Advanced Lectures on Formal Methods for Multicore Programming - 15th Intl. School on Formal Methods for the Design of Computer, Communication, and Software Systems (SFM 2015)*, volume 9104 of *Lecture Notes in Computer Science*, pages 1–56. Springer, 2015.

8 Denis Caromel, Ludovic Henrio, and Bernard Serpette. Asynchronous and deterministic objects. In *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 123–134. ACM Press, 2004.

9 Elias Castegren, Dave Clarke, Kiko Fernandez-Reyes, Tobias Wrigstad, and Albert Mingkun Yang. Attached and Detached Closures in Actors. In *Proc. 8th ACM SIGPLAN Intl. Workshop on Programming Based on Actors, Agents, and Decentralized Control*, AGERE 2018, pages 54–61. ACM, 2018. `doi:10.1145/3281366.3281371`.

10 Dave Clarke, Tobias Wrigstad, Johan Östlund, and Einar Broch Johnsen. Minimal Ownership for Active Objects. In G. Ramalingam, editor, *Proc. 6th Asian Symposium on Programming Languages and Systems (APLAS 2008)*, volume 5356 of *Lecture Notes in Computer Science*, pages 139–154. Springer, 2008.

11 Frank S. de Boer, Dave Clarke, and Einar Broch Johnsen. A complete guide to the future. In *Proc. 16th European Symposium on Programming (ESOP'07)*, volume 4421 of *Lecture Notes in Computer Science*, pages 316–330. Springer, 2007.

12 Frank S. de Boer, Vlad Serbanescu, Reiner Hähnle, Ludovic Henrio, Justine Rochas, Crystal Chang Din, Einar Broch Johnsen, Marjan Sirjani, Ehsan Khamespanah, Kiko Fernandez-Reyes, and Albert Mingkun Yang. A Survey of Active Object Languages. *ACM Comput. Surv.*, 50(5):76:1–76:39, 2017. doi:10.1145/3122848.

13 Kiko Fernandez-Reyes, Dave Clarke, Elias Castegren, and Huu-Phuc Vo. Forward to a Promising Future. In Giovanna Di Marzo Serugendo and Michele Loreti, editors, *Proc. 20th IFIP WG 6.1 Intl. Conf. on Coordination Models and Languages (COORDINATION 2018)*, volume 10852 of *Lecture Notes in Computer Science*, pages 162–180. Springer, 2018.

14 Kiko Fernandez-Reyes, Dave Clarke, and Daniel S. McCain. ParT: An asynchronous parallel abstraction for speculative pipeline computations. In Alberto Lluch-Lafuente and José Proenca, editors, *Proc. 18th IFIP WG 6.1 Intl. Conf. on Coordination Models and Languages (COORDINATION 2016)*, volume 9686 of *Lecture Notes in Computer Science*, pages 101–120. Springer, 2016. doi:10.1007/978-3-319-39519-7_7.

15 Cormac Flanagan and Matthias Felleisen. The Semantics of future and an application. *Journal of Functional Programming*, 9(1):1–31, 1999.

16 Elena Giachino, Ludovic Henrio, Cosimo Laneve, and Vincenzo Mastandrea. Actors may synchronize, safely! In *PPDP 2016 18th International Symposium on Principles and Practice of Declarative Programming* , Edinburgh, United Kingdom, September 2016. URL: https://hal.inria.fr/hal-01345315.

17 Google. Listenable Future Explained. https://github.com/google/guava/wiki/ListenableFutureExplained, January 2018.

18 Philipp Haller, Heather Miller, Aleksandar Prokopec, Viktor Klang, Roland Kuhn, and Vojin Jovanovic. Futures and Promises. http://docs.scala-lang.org/overviews/core/futures.html, 2016.

19 Phillip Haller and Martin Odersky. Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2-3):202–220, 2009.

20 Ludovic Henrio. Data-flow Explicit Futures. Research report, I3S, Université Côte d'Azur, April 2018. URL: https://hal.archives-ouvertes.fr/hal-01758734.

21 Ludovic Henrio and Justine Rochas. Multiactive objects and their applications. *Logical Methods in Computer Science*, Volume 13, Issue 4, November 2017. doi:10.23638/LMCS-13(4:12)2017.

22 Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. ABS: A core language for abstract behavioral specification. In Bernhard Aichernig, Frank S. de Boer, and Marcello M. Bonsangue, editors, *Proc. 9th Intl. Symp. on Formal Methods for Components and Objects (FMCO)*, volume 6957 of *Lecture Notes in Computer Science*, pages 142–164. Springer Verlag, 2011.

23 Einar Broch Johnsen, Olaf Owe, and Ingrid Chieh Yu. Creol: A type-safe object-oriented model for distributed concurrent systems. *Theor. Comput. Sci.*, 365(1-2):23–66, 2006. doi:10.1016/j.tcs.2006.07.031.

24 Simon Peyton Jones. *Haskell 98 Language and Libraries: The Revised Report.* Cambridge University Press, 2003.

25 Robert H. Halstead Jr. MultiLisp: A language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, 1985. doi:10.1145/4472.4478.

26 R. Greg Lavender and Douglas C. Schmidt. Active Object: an Object Behavioral Pattern for Concurrent Programming. *Proc. Pattern Languages of Programs*, 1995.

27 Barbara Liskov and Liuba Shrira. Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems. In Richard L. Wexelblat, editor, *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, USA, June 22-24, 1988*, pages 260–267. ACM, 1988. doi:10.1145/53990.54016.

28 Magnus Madsen, Ondrej Lhoták, and Frank Tip. A model for reasoning about JavaScript promises. *PACMPL*, 1(OOPSLA):86:1–86:24, 2017. doi:10.1145/3133910.

29 Joachim Niehren, Jan Schwinghammer, and Gert Smolka. A Concurrent Lambda Calculus with Futures. *Theoretical Computer Science*, 364(3):338–356, November 2006.

30 Oscar Nierstrasz. Active Objects in Hybrid. In Norman K. Meyrowitz, editor, *Proc. Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'87)*, pages 243–253. ACM, 1987.

31 Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala*. Artima Inc, 2008.

32 Oracle. JDK 10 for `java.util.concurrent.Future`. https://docs.oracle.com/javase/10/docs/api/index.html?java/util/concurrent/Future.html, 2018.

33 Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 1st edition, 2002.

34 John H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999.

35 Andreas Rossberg, Didier Le Botlan, Guido Tack, Thorsten Brunklaus, and Gert Smolka. *Alice Through the Looking Glass*, volume 5 of *Trends in Functional Programming*, pages 79–96. Intellect Books, Bristol, UK, ISBN 1-84150144-1, Munich, Germany, February 2006.

36 Jan Schafer and Arnd Poetzsch-Heffter. JCoBox: Generalizing active objects to concurrent components. *ECOOP 2010–Object-Oriented Programming*, pages 275–299, 2010.

37 Kenjiro Taura, Satoshi Matsuoka, and Akinori Yonezawa. ABCL/f: A future-based polymorphic typed concurrent object-oriented language - its design and implementation. In *Proceedings of the DIMACS workshop on Specification of Parallel Algorithms*, pages 275–292. American Mathematical Society, 1994.

38 Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*. MIT Press, March 2004.

39 Adam Welc, Suresh Jagannathan, and Antony Hosking. Safe futures for Java. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications (OOPSLA'05)*, pages 439–453, New York, NY, USA, 2005. ACM Press.

40 Andrew K. Wright and Matthias Felleisen. A Syntactic Approach to Type Soundness. *Inf. Comput.*, 115(1):38–94, 1994. doi:10.1006/inco.1994.1093.

41 Derek Wyatt. *Akka Concurrency*. Artima, 2013.

Multitier Modules

Pascal Weisenburger
Technische Universität Darmstadt, Germany
weisenburger@cs.tu-darmstadt.de

Guido Salvaneschi
Technische Universität Darmstadt, Germany
salvaneschi@cs.tu-darmstadt.de

Abstract

Multitier programming languages address the complexity of developing distributed systems abstracting over low level implementation details such as data representation, serialization and network protocols. Since the functionalities of different peers can be defined in the same compilation unit, multitier languages do not force developers to modularize software along network boundaries. Unfortunately, combining the code for all tiers into the same compilation unit poses a scalability challenge or forces developers to resort to traditional modularization abstractions that are agnostic to the multitier nature of the language.

In this paper, we address this issue with a module system for multitier languages. Our module system supports encapsulating each (cross-peer) functionality and defining it over abstract peer types. As a result, we disentangle modularization and distribution and we enable the definition of a distributed system as a composition of multitier modules, each representing a subsystem. Our case studies on distributed algorithms, distributed data structures, as well as on the Apache Flink task distribution system, show that multitier modules allow the definition of reusable (abstract) patterns of interaction in distributed software and enable separating the modularization and distribution concerns, properly separating functionalities in distributed systems.

2012 ACM Subject Classification Computing methodologies → Distributed programming languages; Software and its engineering → Modules / packages

Keywords and phrases Distributed Programming, Multitier Programming, Abstract Peer Types, Placement Types, Module Systems, Scala

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2019.3

Funding This work has been supported by the German Research Foundation (DFG) within the Collaborative Research Center (CRC) 1053 MAKI and 1119 CROSSING, by the DFG projects SA 2918/2-1 and SA 2918/3-1, by the Hessian LOEWE initiative within the Software-Factory 4.0 project, by the German Federal Ministry of Education and Research and by the Hessian Ministry of Science and the Arts within CRISP, and by the AWS Cloud Credits for Research program.

Acknowledgements We would like to thank Philipp Schulz for the implementation of the first prototype of this work and all reviewers of this paper for their comments and suggestions.

1 Introduction

Implementing distributed systems is notoriously hard because of a number of issues that naturally arise in this setting, such as consistency, fault tolerance, concurrency, mismatch among data formats, as well as mix of languages and execution platforms.

Multitier – or *tierless* – languages [36, 13, 26, 14, 41] address some of these problems. In *multitier* languages, peers (e.g., the client and the server in a Web application) are written in the same compilation unit. The compiler splits the code into a client unit and a server unit, adds the necessary communication code, performs the necessary translations (e.g., translating client code to JavaScript) and generates the deployable components. As

33rd European Conference on Object-Oriented Programming (ECOOP 2019).
Editor: Alastair F. Donaldson; Article No. 3; pp. 3:1–3:29
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

a result, developers need to use a single language, are not forced to worry about network communication, serialization, data formats and conversions and can focus on the application logic without breaking it down along network boundaries.

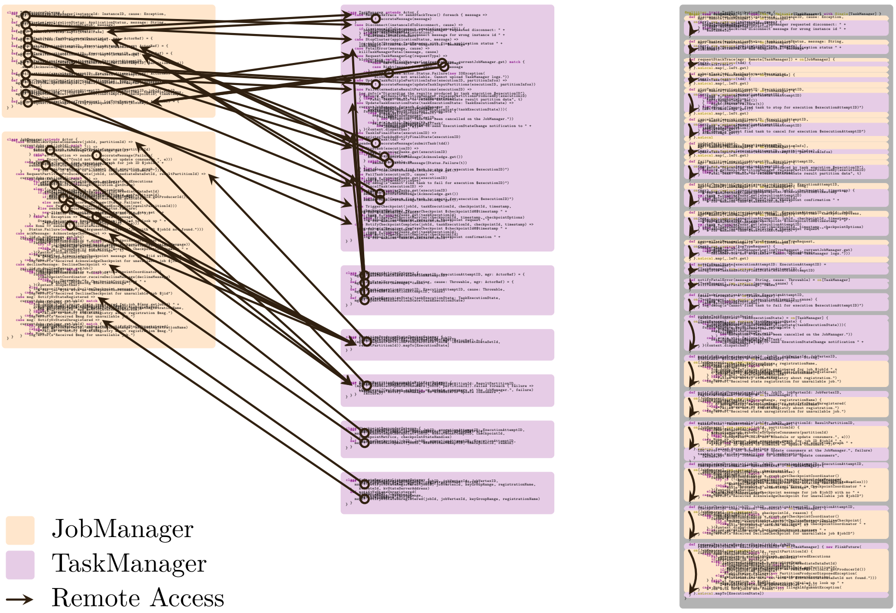
Lack of Modularization Abstractions. Scaling multitier code to large applications, however, is an open problem. Researchers have been focusing on small use cases that primarily aim to demonstrate the design of their language rather than investigating the development of complex applications that require sophisticated modularization and composition, or require abstracting over program locations and architecture. In use cases of limited size, client code and server code are nicely combined in a single compilation unit, but it is unclear what happens when one compilation unit is not enough.

As an example of this issue, Figure 1 provides an overview of the *task distribution system* in the Apache Flink stream processing framework [9]. It consists of the coordinator of the Flink instance, the *JobManager* and one or more *TaskManagers*, which execute computational tasks. Figure 1a shows the JobManager (light orange boxes, left), the TaskManager (dark violet boxes, right) and their communication (arrows). Every box is a class or an actor which is confined by network boundaries. Thus, cross-host data flow belonging to the same (distributed) functionality is scattered over multiple modules. Figure 1b shows an implementation of the same system in the ScalaLoci multitier language [48] (the figure is adapted from the same work). The data flow in the system is much more *regular*, due to the reorganization of the same code in a single unit, yet all the functionalities of the system are concentrated in a single large compilation unit with $\sim$400 LOC.

Unfortunately, adopting the traditional modularization mechanism supported by the base language (e.g., a Haskell module for a Haskell-based multitier language) is not sufficient because such modularization mechanism is not aware of multitier code and it is unclear what output code is produced after the compiler splits the code.

In summary, simplifying reasoning about distributed applications, abstracting over network communication and format conversions, and providing a single language to implement all components, multitier languages have the potential to significantly help programmers developing distributed systems. However, enabling multitier programming to scale to large code bases is still a research problem because of the lack of proper modularization mechanisms. In its current state, multitier programming does effectively defeat the *tyranny of the dominant decomposition* [45] for distributed systems, removing the need to modularize applications according to tiers and network communication. Yet, it does not offer an alternative modularization solution designed in synergy with multitier abstractions.

Contribution. In this paper, we propose LociMod, a novel *multitier module system* for ScalaLoci. Multitier modules encapsulate the interaction between distributed components of (sub)systems, allowing for (1) **decoupling modularization from distribution** and (2) **defining reusable patterns of interaction** that model the functionality of a (distributed) subsystem and that can be composed to build larger distributed systems. LociMod supports strong interfaces [22] to achieve encapsulation and information hiding, such that implementations can be easily exchanged. The main contribution of the work is to make peer types, which define the placement of a functionality in LociMod, abstract. This design choice enables the definition of abstract modules, which capture a fragment of a distributed system, can be further composed with other abstract modules, and can eventually be instantiated for a concrete software architecture.



(a) Original Flink.

(b) ScalaLoci reimplementation.

Figure 1 Flink task distribution system in ScalaLoci, adapted from [48].

Our case studies on distributed algorithms, distributed data structures, as well as on the Apache Flink task distribution system, show that LociMod multitier modules allow the definition of reusable (abstract) patterns of interaction in distributed software and enable separating the modularization and distribution concerns, properly separating functionalities in distributed systems. In summary, this paper makes the following contributions:

- We present LociMod, a novel module system for multitier languages, featuring multitier modules, which support strong interfaces and exchangeable implementations.
- We show that, thanks to LociMod abstract peer types, the interaction between multitier abstractions and modularization features results in a number of powerful abstractions to define and compose distributed systems, including multitier mixin composition and constrained modules.
- We provide an implementation of LociMod as an extension to ScalaLoci, a multitier language embedded into Scala. The implementation supports separate compilation and is publicly available.[1]
- We evaluate LociMod with case studies, including distributed algorithms, distributed data structures and the Apache Flink Big Data processing framework, demonstrating the composition properties of multitier modules and how they can capture (distributed) functionalities in complex systems.

The paper is structured as follows. Section 2 provides an overview of ScalaLoci and the important features of the Scala type system. Section 3 describes the design of multitier modules. Section 4 discusses the implementation. Section 5 presents the evaluation. Section 6 discusses the related work. Section 7 concludes.

[1] http://scala-loci.github.io/

2 Background

2.1 ScalaLoci

Since LociMod is an extension of ScalaLoci [48], we first provide an overview of ScalaLoci to the reader. ScalaLoci is a *general purpose* multitier language for distributed systems – unlike most multitier languages, which focus on the Web (i.e., client-server architecture) only. To support generic distributed architectures, ScalaLoci provides language abstractions for developers to freely define the different components – called *peers* – of the distributed system and their architectural relation. Peers are defined as *peer types*, which allow for specifying the placement of data and computations at the type level using *placement types*, enabling static reasoning about placement. A placement type `T on P`[2] represents a value of a traditional type `T` which is placed on a peer of peer type `P`.

Placed Values. Placed values of type `T on P` are initialized with `placed { e }` expressions. For example, the following string is placed on the `Master` peer:

```
val name: String on Master = placed { "the one and only master" }
```

The `name` value is accessible remotely from other peers. Accessing remote values requires the `asLocal` marker, creating a local representation of the remote value by transmitting it over the network:

```
val masterName: Future[String] on Worker = placed { name.asLocal }
```

Accessing `name` remotely from a `Worker` peer returns a value of type `Future[String]`. Futures – which are part of Scala's standard library – account for network latency and possible communication failures by representing a value which may not be available immediately, but will become available in the future or produce an error.

Remote accessibility of placed values can be regulated: Local placed values denoted by the type `Local[T] on P` specify values that can only be accessed locally from the same peer instance, i.e., remote access via `asLocal` is not possible:

```
val realName: Local[String] on Master = placed { "Rumpelstiltskin" }
```

For a value definition `val v: T on P = placed { e }`, the shorthand notation `val v = on[P] { e }` can be used, inferring the placement type `T on P`.

Placed Computations. Like placed values, placed computations are declared to have a placement type and defined using a `placed` expression:

```
def execute(task: Task[T]): T on Worker = placed { task.process() }
```

Invoking a remote computation is explicit using `remote call`. If the result of a remote computation is of interest to the local peer instance, it can be made available locally using `asLocal` (as described before):

```
val result: Future[T] on Master = placed { (remote call execute(new Task()).asLocal }
```

[2] The Scala compiler treats `T on P` and `on[T, P]` equivalently.

Architecture Specification. In LociMod, the architectural scheme of the distributed system is expressed using *ties*, which specify the kind of relation among peers. Ties are encoded as structural type refinements specifying the `Tie` type for peers. Ties to multiple peers are defined by declaring a compound type (e.g., `type Tie <: Single[Master] with Multiple[Worker]`). Remote access is only possible between tied peers.

For instance, an architecture with a single master that offloads computations to a single worker is defined by a `Master` peer and a `Worker` peer (specified through the `@peer` annotation on type members):

```
@peer type Master <: { type Tie <: Single[Worker] }
@peer type Worker <: { type Tie <: Single[Master] }
```

Both peers have a *single tie* to each other, i.e., workers are always connected to a single master instance and each corresponding master instance always manages a single worker. A variant of the master–worker model, where a single master instance manages multiple workers, is modeled by a *single tie* from worker to master and a *multiple tie* from master to worker:

```
@peer type Master <: { type Tie <: Multiple[Worker] }
@peer type Worker <: { type Tie <: Single[Master] }
```

Section 5.1 presents a more systematic categorization of common distributed architectures and their encoding using peers and ties.

2.2 Scala Abstract Data Types and Path-Dependent Types

The LociMod module system leverages Scala's type system features, in particular abstract types and path-dependent types, which we quickly revise. In Scala, traits, classes and objects (i.e., singleton classes) define type members, which are either abstract (e.g., `type SomeType`) or define concrete type aliases (e.g., `type SomeType = Int`). Abstract type members can define lower and upper type bounds (e.g., `type SomeType >: LowerBound <: UpperBound`), which refine the type while keeping it abstract. Inherited abstract type members can be overridden. Such mechanism enables specializing the upper bound and generalizing the lower bound. In Section 3.2, we define the peers of the distributed system as abstract type members. Refining the upper bound enables specializing a peer as (a subtype of) another peer, enabling peer composition by combining super peers into a sub peer.

Scala types can be *dependent on an path* (of objects). For example, in the following code snippet, both objects `a` and `b` inherit the `SomeType` abstract type member defined in the `Module` trait:

```
trait Module { type SomeType }
object a extends Module
object b extends Module
```

The path-dependent type `a.SomeType` refers to `a`'s `SomeType` and the path-dependent type `b.SomeType` refers to `b`'s `SomeType`. The types depend on the objects `a` or `b`, respectively. They are distinct since their paths differ.

For instance, the following example defines a module `A` with an abstract type member `T`. Further, a module `B` defines an abstract type member `U`. The module `C` extends module `B` inheriting its abstract type member `U`. The module `C` also references an instance `a` of module

A. Module C's U type is overridden and refined as a subtype of the T type defined in the a instance by specifying that the upper bound of U is the path-dependent type a.T:

```scala
trait A { type T }

trait B { type U }

trait C extends B {
  type U <: a.T
  val a: A
}
```

In Section 3.2.1, we use this mechanism to declare references to other modules from within a module and to refer to the peers (defined as abstract type members) in the referenced modules via their path-dependent types.

3 LociMod Multitier Modules

In this section, we describe the LociMod module system. The goal of this section is twofold. On the one hand, we present the design of multitier modules. On the other hand, we demonstrate a number of examples for multitier modules and their composition mechanisms.

We first introduce (concrete) *multitier modules* and show how they can be composed into larger applications, using *module references* – references to other multitier modules. Then we introduce modules with *abstract peer types* and show their composition through module references as well as another composition mechanism, *multitier mixing*. Next we show how such composition mechanism enables defining *constrained multitier modules*.

3.1 Multitier Modules

We embed LociMod into Scala, following the same approach of ScalaLoci, which is a Scala DSL. Scala traits represent modules – adopting Scala's design that unifies object and module systems [30]. Traits can contain abstract declarations and concrete definitions for both type and value members – thus serve as both module definitions and implementations – and Scala objects can be used to instantiate traits.

Module Definition. In LociMod, multitier modules are defined by a trait with the @multitier annotation. Multitier modules can define (i) values placed on different peers and (ii) the peers on which the values are placed – including constraints on the architectural relation between peers. This approach enables modularization across peers (not necessarily along peer boundaries) combining the benefits of multitier programming and modular development. To illustrate, consider an application that allows a user to edit documents offline but also offers the possibility to backup the data to an online storage:

```scala
1 @multitier trait Editor {
2   @peer type Client <: { type Tie <: Single[Server] }
3   @peer type Server <: { type Tie <: Single[Client] }
4
5   val backup: FileBackup
6 }
```

The Editor specifies a Client (Line 2) and a Server (Line 3). The client should be able to backup/restore documents to/from the server, e.g., the client can invoke a backup.store method to backup data. Thus, the module requires an instance of the FileBackup multitier module (Line 5) providing the backup service. Section 3.2 shows how the Editor and the FileBackup module can be composed.

Encapsulation with Multitier Modules. LociMod's multitier modules *encapsulate distributed (sub)systems* with a specified distributed architecture, enabling the creation of larger distributed applications by composition. The following code shows a possible implementation for the backup service subsystem using the file system to store backups:

```
 1 @multitier trait FileBackup {
 2   @peer type Processor <: { type Tie <: Single[Storage] }
 3   @peer type Storage <: { type Tie <: Single[Processor] }
 4
 5   def store(id: Long, data: Data): Unit on Processor = placed { remote call write(id, data) }
 6   def load(id: Long): Future[Data] on Processor = placed { (remote call read(id)).asLocal }
 7
 8   private def write(id: Long, data: Data): Unit on Storage = placed {
 9     writeToFile(data, s"/storage/$id") }
10   private def read(id: Long): Data on Storage = placed {
11     readFromFile[Data](s"/storage/$id") }
12 }
```

The multitier `FileBackup` module specifies a `Processor` to compress data (of type `Data`) and a `Storage` peer to store and retrieve data associating them to an ID. The `store` (Line 5) and `load` (Line 6) methods can be called on the `Processor` peer, invoking `write` (Line 8) and `read` (Line 10) remotely on the `Storage` peer. The implementations of the `write` and `read` methods operate on files.

LociMod multitier modules support standard access modifiers for placed values (e.g., `private`, `protected` etc.), which are used as a technique to encapsulate module functionality. In the `FileBackup` module, the `write` and the `read` methods are declared private, so other modules that use `FileBackup` cannot directly access them. Overall, the `FileBackup` module encapsulates all the functionalities related to the backup service subsystem, including the communication between `Processor` and `Storage`.

As the last example demonstrates, multitier modules enable separating modularization and distribution concerns, allowing developers to organize applications based on logical units instead of network boundaries. A multitier module abstracts over potentially multiple components and the communication between them, specifying distribution by expressing the placement of a computation on a peer in its type. Both axes are traditionally intertwined by having to implement a component of the distributed system in a module (e.g., a class, an actor, etc.) leading to cross-host functionality being scattered over multiple modules.

Multitier Modules as Interfaces and Implementations. To decouple the code that uses a multitier module from the concrete implementation of such a module, LociMod supports modules to be used as interfaces and implementations. Multitier modules can be abstract, i.e., defining only abstract members, acting as module interfaces or they can define concrete implementations. For example, applications that require a backup service can be developed against the `BackupService` module interface, which declares a `store` and a `load` method:

```
@multitier trait BackupService {
  @peer type Processor <: { type Tie <: Single[Storage] }
  @peer type Storage <: { type Tie <: Single[Processor] }

  def store(id: Long, data: Data): Unit on Processor
  def load(id: Long): Future[Data] on Processor
}
```

LociMod adopts Scala's inheritance mechanism to express the relation between the multitier modules used as interfaces and their implementations. The `FileBackup` module presented before is a possible implementation for the BackupService module interface, i.e., we can redefine FileBackup to let it implement BackupService:

```
@multitier trait FileBackup extends BackupService { ... }
```

The following example presents a different implementation for the BackupService module interface using a database backend (instead of a file system) as storage:

```
1  @multitier trait DatabaseBackup extends BackupService {
2    def store(id: Long, data: Data): Unit on Processor = placed { remote call insert(id, data) }
3    def load(id: Long): Future[Data] on Processor = placed { (remote call query(id)).asLocal }
4
5    private val db: AsyncContext = ...
6
7    private def insert(id: Long, data: Data): Unit on Storage = placed {
8      db.run(query[(Long, Data)].insert(lift(id -> data))) }
9    private def query(id: Long): Future[Data] on Storage = placed {
10     db.run(query[(Long, Data)].filter { _._1 == lift(id) }) map { _.head._2 } }
11 }
```

The implementations of the store and of the load methods invoke insert (Line 7) and query (Line 9) remotely, which insert the backup data into a database and retrieve the data from a database, respectively.[3]

Combining Multitier Modules. Thanks to the separation between module interfaces and module implementations, applications can be developed against the interface, remaining agnostic to the implementation details of a subsystem encapsulated in a multitier module. For example, the Editor presented before can be adapted to use a BackupService interface instead of the concrete FileBackup implementation (Line 5):

```
1 @multitier trait Editor {
2   @peer type Client <: { type Tie <: Single[Server] }
3   @peer type Server <: { type Tie <: Single[Client] }
4
5   val backup: BackupService
6 }
```

Finally, a multitier module can be instantiated by instantiating concrete implementations of the module interfaces it refers to. LociMod relies on the Scala approach of using an object to instantiate a module, i.e., declaring an object that extends a trait – or mixes together multiple traits – creates an instance of those traits. For example, the following code creates an editor instance of the Editor module by providing a concrete DatabaseBackup instance for the abstract backup value:

```
@multitier object editor extends Editor {
  @multitier object backup extends DatabaseBackup
}
```

The multitier module instance of a @multitier object can be used to run different peers from (non-multitier) standard Scala code (e.g., the Client and the Server peer), where every peer instance only contains the values placed on the respective peer. Peer startup is presented in Section 3.4.

[3] The example uses the Quill http://getquill.io query language to access the database

3.2 Abstract Peer Types

In the previous section, we have shown how to encapsulate a subsystem within a multitier module and how to define a module interface such that multiple implementations are possible. LociMod modules allow for going further, enabling abstraction over placement using abstract peer types. Peer types are abstract type members of traits, i.e., they can be overridden in sub traits, specializing their type. As a consequence, LociMod multitier modules are parametric on peer types. For example, the `BackupService` module of the previous section defines an abstract `Processor` peer, but the `Processor` peer does not necessarily need to refer to a physical peer in the system. Instead, it denotes a *logical* place. When running the distributed system, a `Client` peer, for example, may adopt the `Processor` role, by specializing the `Client` peer to be a `Processor` peer.

Peer types are used to distinguish places only at the type level, i.e., the placement type `T on P` represents a run time value of type `T`. The peer type `P` is used to keep track of the value's placement, but a value of type `P` is never constructed at run time. Hence, `T on P` is essentially a "phantom type" [12] due to its parameter `P`.

The next two sections describe the interaction of abstract peer types with two composition mechanisms for multitier modules. We already encountered the first mechanism, module references. The other mechanism, multitier mixing, enables combining multitier modules directly. In both cases, the peers defined in a module can be specialized with the role of other modules' peers.

3.2.1 Peer Type Specialization with Module References

Since peer types are abstract, they can be specialized by narrowing their upper type bound, augmenting peers with different roles defined by other peers. Peers can subsume the roles of other peers – similar to subtyping on classes – enabling polymorphic usage of peers. Programmers can use this feature to augment peer types with roles defined by other peer types by establishing a subtyping relation between both peers. This mechanism enables developers to define reusable patterns of interaction among peers that can be specialized later to any of the existing peers of an application.

For example, the editor application that requires the backup service (Section 3.1) needs to specialize its `Client` peer to be a `Processor` peer and its `Server` peer to be a `Storage` peer for clients to be able to perform backups on the server:

```
1 @multitier trait Editor {
2   @peer type Client <: backup.Processor { type Tie <: Single[Server] with Single[backup.Storage] }
3   @peer type Server <: backup.Storage { type Tie <: Single[Client] with Single[backup.Processor] }
4
5   val backup: BackupService
6 }
```

We specify the `Client` peer to be a (subtype of the) `backup.Processor` peer (Line 2) and the `Server` peer to be a (subtype of the) `backup.Storage` peer (Line 3). Both `backup.Processor` and `backup.Storage` refer to the peer types defined on the `BackupService` instance referenced by `backup`. We can use such *module references* to refer to (path-dependent) peer types through a reference to the multitier module.

Since the subtyping relation `Server <: backup.Storage` specifies that a server *is a* storage peer, the backup functionality (i.e., all values and methods placed on the `Storage` peer) are also placed on the `Server` peer. Super peer definitions are locally available on sub peers, making peers composable using subtyping. Abstract peer types specify such subtyping relation by declaring an upper type bound. When augmenting the server with the storage functionality using subtyping, the `Tie` type also has to be a subtype of the `backup.Storage` peer's `Tie` type. This type level encoding of the architectural relations among peers enables the Scala compiler to check that the combined architecture of the system complies to the architectural constraints of every subsystem.

Note that, for the current example, one may expect to unify the `Server` and the `Storage` peer, so they refer to the same peer, specifying type equality instead of a subtyping relation:

```
@peer type Server = backup.Storage { type Tie <: Single[Client] }
```

Since peer types, however, are never instantiated (they are used only as phantom types to keep track of placement at the type level) we can always keep peer types abstract, only specifying an upper type bound. Hence, it is sufficient to specialize `Server` to be a `backup.Storage`, keeping the `Server` peer abstract for potential further specialization.

3.2.2 Peer Type Specialization with Multitier Mixing

The previous section shows how peer types can be specialized when referring to modules through *module references*. This section presents a different composition mechanism based on composing traits – similar to *mixin composition* [4]. Since LociMod multitier modules can encapsulate distributed subsystems (Section 3.1), mixing multitier modules enables including the implementations of different subsystems into a single module.

LociMod separates modules from peers, i.e., mixing modules does not equate to unify the peers they define. Hence we need a way to coalesce different peers. We use (i) subtyping and (ii) overriding of abstract types as a mechanism to specify that a peer also comprises the placed values of (i) the super peers and (ii) the overridden peers, i.e., a peer subsumes the functionalities of its super peers (Section 3.2.1) and its overridden peers. Since peers are abstract type members, they can be overridden in sub modules. To demonstrate mixing of multitier modules we consider the case of two different functionalities.

First, we consider a computing scheme where a master offloads tasks to worker nodes:

```
1 @multitier trait MultipleMasterWorker[T] {
2   @peer type Master <: { type Tie <: Multiple[Worker] }
3   @peer type Worker <: { type Tie <: Single[Master] }
4
5   def run(task: Task[T]): Future[T] on Master = placed {
6     (remote(selectWorker()) call execute(task)).asLocal
7   }
8   private def execute(task: Task[T]): T on Worker = placed { task.process() }
9 }
```

The example defines a master that has a multiple tie to workers (Line 2) and a worker that has a single tie to a master (Line 3). The `run` method has the placed type `Future[T] on Master` (Line 5), placing `run` on the `Master` peer. Running a task remotely results in a `Future` [3] to account for processing time and network delays. The remote call to `execute` – to be executed on the worker – (Line 6) starts processing the task (Line 8). The remote result is transferred back to the master as `Future[T]` using `asLocal` (Line 6). A *single* worker instance in a pool of workers is selected for processing the task via the `selectWorker` method (Line 6, the implementation of `selectWorker` is omitted, for simplicity).

Second, we consider the case of monitoring, a functionality that is required in many distributed applications to react to possible failures [23]. In LociMod, a heartbeat mechanism can be defined across a `Monitored` and a `Monitor` peer in a multitier module:

```
1 @multitier trait Monitoring {
2   @peer type Monitor <: { type Tie <: Multiple[Monitored] }
3   @peer type Monitored <: { type Tie <: Single[Monitor] }
4 
5   def monitoredTimedOut(monitored: Remote[Monitored]): Unit on Monitor
6 
7   ...
8 }
```

The module defines the architecture with a single monitor and multiple monitored peers (Line 2 and 3). The `monitoredTimedOut` method (Line 5) is invoked by `Monitoring` implementations whenever a heartbeat was not received from a monitored peer instance for some time. We leave out the actual implementation of the monitoring logic for brevity.

To add monitoring to an application, such application has to be mixed with the `Monitoring` module. Mixing composition brings the members declared in all mixed-in modules into the local scope of the module that mixes in the other modules, i.e., all peer types of the mixed-in modules are in scope. However, the peer types of different modules define separate architectures, which can then be combined by specializing the peers of one module to the peers of other modules. For example, to add monitoring to the the `MultipleMasterWorker` functionality, `MultipleMasterWorker` needs to be mixed with `Monitoring` and the `Master` and `Worker` peers need to be overridden to be (subtypes of) `Monitor` and `Monitored` peers:

```
@multitier trait MonitoredMasterWorker[T] extends MultipleMasterWorker[T] with Monitoring {
  @peer type Master <: Monitor { type Tie <: Multiple[Worker] with Multiple[Monitored] }
  @peer type Worker <: Monitored { type Tie <: Single[Master] with Single[Monitor] }
}
```

Specializing peers of mixed modules follows the same approach as specializing peers accessible through module references (Section 3.2.1), i.e., `Master <: Monitor` specifies that a master is a monitor peer, augmenting the master with the monitor functionality. Also, for specialization using peers of mixed-in modules, the compiler checks that the combined architecture of the system complies to the architectural constraints of every subsystem.

3.2.3 Properties of Abstract Peer Types

LociMod abstract peer types share commonalities with both parametric polymorphism – considering type parameters as type members [29, 46] – like ML parameterized types [25] or Java generics [5], as well as subtyping in object-oriented languages. Similar to parametric polymorphism, abstract peer types allow parametric usage of peer types as shown for the `BackupService` module defining a `Storage` peer parameter. Distinctive from parametric polymorphism, however, with abstract peer types, peer parameters remain abstract, i.e., specializing peers does not unify peer types. Instead, similar to subtyping, specializing peers establishes an *is-a* relation.

Placement types `T on P` support suptyping between peers by being covariant in the type of the placed value and contravariant in the peer (i.e., the `on` type is defined as `type on[+T, -P]`), which allows values to be used in a context where a value of a super type placed on a sub peer is expected. This encoding is sound since a subtype can be used where a super type is expected and values placed on super peers are available on all sub peers. For example, we can extend the `Editor` with a `WebClient`, which is a special kind of client (i.e., `WebClient <: Client`, Line 5) with a Web user interface (Line 8), and a `MobileClient` (i.e., Line 6):

```
 1 @multitier trait Editor {
 2   @peer type Server <: { type Tie <: Multiple[Client] }
 3   ...
 4   @peer type Client <: { type Tie <: Single[Server] }
 5   @peer type WebClient <: Client { type Tie <: Single[Server] }
 6   @peer type MobileClient <: Client { type Tie <: Single[Server] }
 7
 8   val webUI: UI on WebClient
 9   val ui: UI on Client = placed { webUI }  // ✗ Error: `Client` not a subtype of `WebClient`
10 }
```

By using subtyping on peer types, not unifying the types, we are able to distinguish between the general `Client` peer, which can have different specializations (e.g., `WebClient` and `MobileClient`), i.e., every Web client is a client but not every client is a Web client. By keeping the types distinguishable, the `ui` binding (Line 9) is rejected by the compiler since it defines a value on the `Client` peer, i.e., the access to `webUI` inside the placed expression is computed on the `Client` peer. However, `webUI` is not available on `Client` since it is placed on `WebClient` and a client is not necessarily a Web client.

3.3 Constrained Multitier Modules

LociMod multitier modules not only allow abstraction over placement, but also the definition of *constrained multitier modules* that refer to other modules. This feature enables expressing constraints among the modules of a system, such as that one functionality is required to enable another. In LociMod, Scala's self-type annotations express such constraints, indicating which other module is required during mixin composition. To improve decoupling, constraints are often defined on module interfaces, such that multiple module implementations are possible.

Applications requiring constrained modules include distributed algorithms, discussed in more detail in the evaluation (Section 5.1). For example, a global locking scheme ensuring mutual exclusion for a shared resource can be implemented based on a central coordinator. Choosing a coordinator among connected peers requires a leader election algorithm. The `MutualExclusion` module declares a `lock` (Line 2) and `unlock` (Line 3) method for regulating access to a shared resource. `MutualExclusion` is constrained over `LeaderElection` since our locking scheme requires the leader election functionality:

```
1 @multitier trait MutualExclusion { this: LeaderElection =>
2   def lock(id: T): Boolean on Node
3   def unlock(id: Id): Unit on Node
4 }
```

Such requirement, expressed as a Scala self-type (Line 1), forces the developer to mix in a `LeaderElection` implementation to create instances of the `MutualExclusion` module.

A leader election algorithm can be defined by the following module interface:

```
1 @multitier trait LeaderElection[T] {
2   @peer type Node
3
4   def electLeader(): Unit on Node
5   def electedAsLeader(): Unit on Node
6 }
```

The module defines an `electLeader` method (Line 4) to initiate the leader election. The `electedAsLeader` method (Line 5) is called by `LeaderElection` module implementations on the peer instance that has been elected to be the leader.

All definitions of the `LeaderElection` module required by the self-type annotation are available in the local scope of the `MutualExclusion` module, which includes peer types and placed values. A self-type expresses a requirement but not a subtyping relation, i.e., we express the requirement on `LeaderElection` in the example as self-type since the `MutualExclusion` functionality requires leader election but is not a leader election module itself.

Multiple constraints can be expressed by using a compound type. For example, different peer instances often need to have unique identifiers to distinguish among them. Assuming an `Id` module provides such mechanism, a module which requires both the leader election and the identification functionality can specify both required modules as compound self-type `this: LeaderElection with Id`. Such requirement makes the definitions of both the `LeaderElection` and the `Id` module available in the module's local scope and forces the developer to mix in implementations for both modules.

Mixin composition is guaranteed by the compiler to conform to the self-type (which is the essence of the Scala *cake pattern*). Assuming a `YoYo` implementation of the `LeaderElection` interface which implements the Yo-Yo algorithm [39] (Section 5.1 presents different leader election implementations), the following code shows how a `MutualExclusion` instance can be created by mixing together `MutualExclusion` and `YoYo`:

```
@multitier object mutualExclusion extends MutualExclusion with YoYo
```

The `YoYo` implementation of the `LeaderElection` interface satisfies the `MutualExclusion` module's self-type constraint on the `LeaderElection` interface. Since mixing together `MutualExclusion` and `YoYo` fulfills all constraints and leaves no values abstract, the module can be instantiated.

3.4 Peer Startup

In the previous sections, we have shown how LociMod multitier *modules are instantiated*. To start up a distributed system, however, we also need to *start peers* defined in the modules. Different peer instances are typically started on different hosts and connect with each other over a network according to the architecture specification. As a consequence, an additional step is required to start the peers of (already instantiated) modules. For the master–worker example, the master and the worker peers are started as follows:

```
1  @multitier object masterWorker extends MultipleMasterWorker[Int]
2
3  object Master extends App {
4    multitier start new Instance[masterWorker.Master](
5      listen[masterWorker.Worker] { TCP(1099) })
6  }
7
8  object Worker extends App {
9    multitier start new Instance[masterWorker.Worker](
10     connect[masterWorker.Master] { TCP("localhost", 1099) })
11 }
```

We follow the idiomatic way of defining an executable Scala application, where an `object` extends `App` (Line 3 and 8). The object body is executed when the application starts. The code executed when staring a Scala application is standard (non-multitier) Scala, which, in our example, uses `multitier start Instance[...]` to start a peer of an instantiated multitier module. Line 1 instantiates a `MasterWorker` module using the `MultipleMasterWorker` implementation. Line 4 starts a `Master` peer of the module, which uses TCP to listen for connections from `Worker` peer instances. Line 9 starts a `Worker` peer of the module, which uses TCP to connect to a running `Master` peer instance.

4 Implementation

The implementation of LociMod required to modify ∼5 K LOCs of the ScalaLoci codebase. The ScalaLoci compilation process entails three main aspects [48]: (1) the type-level encoding of placement types into the Scala type system, (2) the compile-time macro-driven code separation of code belonging to different peers and (3) the injection of the communication code. The implementation of LociMod requires plugging into the steps above to introduce functionalities for module definition and composition as well as checks for architectural conformance. Both are discussed hereafter.

We preserve Scala's separate compilation because our implementation is based on Scala macros, which expand locally and cannot transform any other code than the annotated trait, class or object under expansion. Once modules are compiled, they are not recompiled unless their code or interfaces on which they depend change.

Macro Expansion. To enable distributed functionalities bundled in a multitier module (Section 3.1) to be executed on different machines (Section 3.4), our implementation separates multitier modules into peer-specific parts and replaces remote accesses with calls to the communication runtime, auto-generating the transmission boilerplate code. For the splitting, we rely on Scala annotation macros [8] (traits and objects are annotated with `@multitier`), transforming the type-checked abstract syntax tree[4] of the module. Placement types, specifying which values belong to which peer, have no direct semantic equivalent in plain Scala. The implementation splits multitier code based on placement types, thereby effectively erasing placement types from the generated code.

Listing 1 provides an intuition of how the macro expansion works, demonstrating module and peer composition as well as remote access. The LociMod code (Listing 1a) defines a module `A` with a peer `Peer` and a placed value `value`. Module `B` mixes in module `A` (Line 6), defines a reference to an instance of module `A` (Line 9), and accesses a remote value through the reference (Line 13).

In the expanded code (Listing 1b, simplified excerpt), placed values are annotated with `compileTimeOnly` (Line 3), which instructs the Scala compiler to issue an error in case such value is referenced in user code after macro expansion. The code generation creates `Marshallable` instances (Line 4) for network transmission of placed values and runtime identifiers for placed values (Line 5), modules (Line 16) and peers (Line 17) for dispatching remote accesses. The splitting process generates a `<placed values>` trait, which contains all placed values in the same order in which they appear in the multitier module to retain the initialization order. Values, however, are nulled (Line 9) and only initialized for the peer on which they are placed. Therefore, the splitting process generates an additional *peer trait* for every peer (Line 10), thus splitting multitier code into peer-specific components. Peer traits also handle local dispatching of remote requests, unmarshalling arguments and marshalling the return value (Line 14).

The example illustrates our module composition mechanisms. Mixing module `A` into module `B` results in the respective `<placed values>` and peer traits being mixed in (Line 25 and 34), using Scala mixin composition. For the `module` reference (Line 22, largely left out for brevity), both the generated module identifier (Line 22) and the dispatching logic for remote requests (Line 32) keep the path of the module reference (`"module"`) into account, to

[4] Annotation macros are expanded before type-checking but can explicitly invoke the type checker to obtain typed abstract syntax trees

Listing 1 Macro Expansion.

(a) LociMod user code.

```
@multitier trait A {
  @peer type Peer
  val value: Int on Peer
}

@multitier trait B extends A {
  @peer type Peer <: module.Peer { type Tie <: Single[module.Peer] }

  @multitier object module extends A {
    val value: Int on Peer = placed { 42 }
  }

  val localValue: Local[Future[Int]] on Peer = placed { module.value.asLocal }
}
```

(b) Scala code after LociMod expansion.

```
 1 trait A {
 2   @peer type Peer
 3   @compileTimeOnly("Remote access must be explicit.") val value: Int on Peer
 4   @MarshallableInfo final val $loci$mar$A$0 = Marshallable[Int]
 5   @PlacedValueInfo("value:scala.Int", null, $loci$mar$A$0) final val $loci$val$A$0 =
 6     new PlacedValue[Unit, Unit, Future[Unit], Int, Int, Future[Int]](
 7       Value.Signature("value:scala.Int", $loci$sig.path), true, null, $loci$mar$A$0)
 8
 9   trait `<placed values>` extends PlacedValues { val value: Int = null.asInstanceOf[Int] }
10   trait $loci$peer$Peer extends `<placed values>` {
11     def $loci$dispatch(req: MessageBuffer, sig: Value.Signature, ref: Value.Reference) =
12       if (sig.path.isEmpty) sig.name match {
13         case $loci$val$A$0.sig.name =>
14           Try(value) map { response => $loci$mar$A$0.marshal(response, ref) } ... } else ... }
15
16   lazy val $loci$sig = Module.Signature("A")
17   lazy val $loci$peer$sig$Peer = Peer.Signature("Peer", collection.immutable.Nil, $loci$sig)
18 }
19
20 trait B extends A {
21   @peer type Peer <: module.Peer { type Tie <: Single[module.Peer] }
22   object module extends A { lazy val $loci$sig = Module.Signature("B#module", "module") ... }
23   @compileTimeOnly("...") val remoteValue = null.asInstanceOf[Local[Future[Int]] on Peer]
24
25   trait `<placed values>` extends PlacedValues with super[A].`<placed values>` {
26     final lazy val module: B.this.module.`<placed values>` = $loci$multitier$module()
27     val remoteValue: Future[Int] = $loci$expr$B$0()
28     protected[this] def $loci$expr$B$0(): Future[Int] = null.asInstanceOf[Future[Int]]
29
30     def $loci$dispatch(req: MessageBuffer, sig: Value.Signature, ref: Value.Reference) =
31       if (sig.path.isEmpty) ... else sig.path.head match {
32         case "module" => module.$loci$dispatch(req, sig.copy(sig.name, sig.path.tail), ref) ... } }
33
34   trait $loci$peer$Peer extends `<placed values>` with super[A].$loci$peer$Peer {
35     protected[this] def $loci$multitier$module() = new B.this.module.$loci$peer$Peer { ... }
36     protected[this] def $loci$expr$B$0(): Future[Int] = SingleIntAccessor(RemoteValue)(
37       new RemoteRequest[Int from B.this.module.Peer, Future[Int], Peer, Single, Unit](
38         (), B.this.module.$loci$val$A$0, B.this.module.$loci$peer$sig$Peer, ...)).asLocal }
39 ... }
```

handle remote access to path-dependent modules. The `module` reference for the peer trait generated for module `B`'s `Peer` (Line 34) is instantiated to the peer trait generated for module `A`'s `Peer` (Line 35), so that values placed on module `B`'s `Peer` can access values placed on module `A`'s `Peer` since module `B` defines `Peer <: module.Peer`. Since peer types are used to guide the splitting and define the composition scheme of the synthesized peer traits, peer types themselves are never instantiated. Hence, they can be abstract.

Like value of module A, localValue of module B is nulled in the <placed values> trait (Line 27 and 28) and initialized in the generated peer trait (Line 36). Since localValue is defined local (i.e., not remotely accessible), no Marshallable instance or runtime identifier is generated for localValue. The remote access module.value.asLocal is expanded into a call to the communication backend with the remote value and remote peer identifiers as arguments (Lines 36–38).

As illustrated by the example, the code generation solely replaces the code of the annotated trait, class or object and only depends on the super traits and classes and the definitions in the multitier modules' body, thus retaining the same support for separate compilation offered by standard Scala traits, classes and objects.

Correctness Checks. Abstract peer types can be specialized, introducing further constraints on the architecture in which they are already involved. Our approach ensures that the architecture of the specialized peers does not violate the architectural constraints of the more general peers. Specifically, ties defined for a peer also need to be defined when specializing the peer, i.e., the tie of a peer needs to be a subtype of the ties of all super and overridden peers. It is, however, possible to refine a tie to make it more specific (i.e., a *multiple* tie is the most general from, whereas an *optional* tie is more specific and a *single* tie is the most specific form). For example, when specializing a Server peer with a Multiple[Client] tie to a WebServer <: Server peer, the WebServer also needs to specify the tie to the Client. It can specify the type as Multiple[Client] (like its super peer), but it can also specify a more specific tie, e.g., Single[Client]. Refining ties is sound since, if code placed on a peer is able to handle any number of connected remote instances (multiple tie), particularly, it can also handle the case when at most one instance is connected (optional or single tie) – but not the other way around.

5 Evaluation

The objective of the evaluation is to assess the design goals established in Section 3, answering the following research questions:

RQ1 Do multitier modules enable defining reusable patterns of interaction in distributed software?
RQ2 Do multitier modules enable separating the modularization and distribution concerns?

For RQ1, we first consider **distributed algorithms** as a case study. Distributed algorithms are a suitable case study because – as we explain soon – they depend on each other and on the underlying architecture. Yet, one wants to keep each algorithm modularized in a way that algorithms can be freely composed. Second, we show how **distributed data structures** can be implemented in LociMod. This case study requires to hide the internal behavior of the data structure from user code as well as to provide a design that does not depend on the specific system architecture. For RQ2, we evaluate the applicability of LociMod to existing real-word software. We reimplemented the *task distribution system* of the Apache Flink **distributed stream processing** framework introduced in Section 1 using multitier modules.

5.1 Distributed Algorithms

We present a case study on a distributed algorithm for mutual exclusion through global locking to access a shared resource. As global locking requires a leader election algorithm, we implement different election algorithms as reusable multitier modules. Also, leader

Listing 2 Mutual Exclusion.

```
 1 @multitier trait MutualExclusion[T] { this: Architecture with LeaderElection[T] =>
 2   private var locked: Option[T] localOn Node = placed { None }
 3
 4   def lock(id: T): Boolean on Node = placed {
 5     if (state == Leader && locked.isEmpty) {
 6       locked = Some(id)
 7       true
 8     }
 9     else
10       false
11   }
12
13   def unlock(id: Id): Unit on Node = placed {
14     if(state == Leader && locked == Some(id))
15       locked = None
16   }
17 }
```

election algorithms assume different distributed architectures, which we represent as multitier modules, too. The implemented mechanism relies on a central coordinator (Listing 2). The MutualExclusion module is parameterized over the leader election algorithm using constrained multitier mixing by specifying a requirement on the LeaderElection interface (Line 1) abstracting over concrete leader election implementations. LeaderElection provides the state method (Line 5 and 14) indicating whether the local node is the elected leader. The MutualExclusion module defines the lock (Line 4) and the unlock (Line 13) methods, to acquire and release the lock.

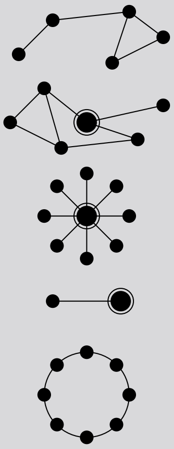
System Architectures. The MutualExclusion module (Listing 2) specifies a constraint on Architecture (Line 1) requiring any distributed architecture for the system abstracting over a concrete one. Architecture is the base trait for different distributed architectures expressed as reusable modules. Listing 3 shows the definitions for different architectures with their iconification on the right. The Architecture module defines the general Node peer and the constraint that peers of type Node are connected to an arbitrary number of other Node peers. The P2P module defines a Peer that can connect to arbitrary many other peers. Thus, the P2P is essentially the general architecture since nodes connecting in a P2P fashion do not impose any additional architectural constraints. The P2PRegistry module adds a central registry, to which peers can connect. The MultiClientServer module defines a client that is always connected to single server, while the server can handle multiple clients simultaneously. The ClientServer module specifies a server that always handles a single client instance. For the Ring module, we define a Prev and a Next peer. A RingNode itself is both a predecessor and a successor. All Node peers have a single tie to their predecessor and a single tie to their successor.

Leader Election. We present the LeaderElection interface for a generic leader election algorithm in LociMod. Since leader election differs depending on the network architecture, the interface defines a self-type constraint on Architecture, abstracting over the concrete network architecture constraining multitier mixing:

```
@multitier trait LeaderElection[T] { this: Architecture with Id[T] =>
  def state: State on Node
  def electLeader(): Unit on Node
  def electedAsLeader(): Unit on Node
}
```

Listing 3 Distributed Architectures.

```
@multitier trait Architecture {
  @peer type Node <: { type Tie <: Multiple[Node] }
}
@multitier trait P2P extends Architecture {
  @peer type Peer <: Node { type Tie <: Multiple[Peer] }
}
@multitier trait P2PRegistry extends P2P {
  @peer type Registry <: Node { type Tie <: Multiple[Peer] }
  @peer type Peer <: Node { type Tie <: Optional[Registry] with Multiple[Peer] }
}
@multitier trait MultiClientServer extends Architecture {
  @peer type Server <: Node { type Tie <: Multiple[Client] }
  @peer type Client <: Node { type Tie <: Single[Server] with Single[Node] }
}
@multitier trait ClientServer extends MultiClientServer {
  @peer type Server <: Node { type Tie <: Single[Client] }
  @peer type Client <: Node { type Tie <: Single[Server] with Single[Node] }
}
@multitier trait Ring extends Architecture {
  @peer type Node <: { type Tie <: Single[Prev] with Single[Next] }
  @peer type Prev <: Node
  @peer type Next <: Node
  @peer type RingNode <: Prev with Next
}
```



Further, the interface abstracts over a mechanism for assigning IDs to nodes implemented by the `Id[T]` module, where `T` is the type of the IDs. The `Id` module interface defines a local `id` value on every node and requires an ordering relation for IDs:

```
@multitier abstract class Id[T: Ordering] { this: Architecture =>
  val id: Local[T] on Node
}
```

The `LeaderElection` module defines a local variable `state` that captures the state of each peer (e.g., `Candidate`, `Leader` or `Follower`). The `electLeader` method is kept abstract to be implemented by a concrete implementation of the interface. After a peer instance has been elected to be the leader, implementations of `LeaderElection` call `electedAsLeader`. We consider three leader election algorithms:

Hirschberg-Sinclair Leader Election. The Hirschberg-Sinclair algorithm [21] implements leader election for a ring topology. In every algorithm phase, each peer instance sends its ID to both of its neighbors in the ring. IDs circulate and each node compares the ID with its own. The peer with the greatest ID becomes the leader. The logic of the algorithm is encapsulated into the `HirschbergSinclair` module, which extends `LeaderElection`:

```
1 @multitier trait HirschbergSinclair[T] extends LeaderElection[T] { this: Ring with Id[T] =>
2   def electLeader() = on[Node] { elect(0) }
3   private def elect(phase: Int) = on[Node] { /* ... */ }
4   private def propagate(remoteId: T, hops: Int, direction: Direction) = on[Node] { /* ... */ }
5 }
```

The module's self-type encodes that the algorithm is designed for ring networks (Line 1). When a new leader election is initiated by calling `electLeader` (Line 2), the `elect` method is invoked (Line 3). The `propagate` method passes the IDs of peer instances along the ring and compares them with the local ID.

Yo-Yo Leader Election. The Yo-Yo algorithm [39] is a universal leader election protocol, i.e., it is independent of the network architecture. For this reason, the self-type constraint of the `YoYo` implementation is simply `Architecture with Id[T]`. In the Yo-Yo algorithm, each node exchanges its ID with all neighbors, progressively pruning subgraphs where there is no lower ID. The node with the lower ID becomes the leader.

Raft Leader Election. The Raft consensus algorithm [32] elects a leader by making use of randomized timeouts. Once a leader is elected, it maintains its leadership by sending heartbeat messages to all peer instances. If instances do not receive a heartbeat message from the current leader for a certain amount of time, they initiate a new election.

Instantiating Global Locking. The following code instantiates a `MutualExclusion` module using the Hirschberg-Sinclair leader election algorithm for a ring architecture:

```scala
@multitier object locking extends
  MutualExclusion[Int] with HirschbergSinclair[Int] with Ring with RandomIntId
```

The example mixes in a module implementation for every module over which other modules are parameterized, i.e., `MutualExclusion` is parameterized over `LeaderElection`, which is instantiated to `HirschbergSinclair`. `HirschbergSinclair` requires a `Ring` architecture and an `Id` implementation, which is instantiated to a `RandomIntId` module (whose implementation is left out for brevity). The following code, instead, instantiates a `MutualExclusion` module using the Yo-Yo leader election algorithm for a P2P architecture:

```scala
@multitier object locking extends
  MutualExclusion[Int] with YoYo[Int] with P2P with RandomIntId
```

Summary. The case study demonstrates how module implementations for concrete architectures and leader election algorithms can be composed into a module providing global locking and made reusable. Since modules encapsulate a functionality within a well-defined interface, leader election algorithms can be easily exchanged. Our approach allows for simply mixing different cross-peer functionality together without changing any multitier code that is encapsulated into the modules (RQ1).

5.2 Distributed Data Structures

This section demonstrates how distributed data structures can be implemented in LociMod. First, we reimplement non-multitier conflict-free replicated data types (CRDTs) as multitier modules in LociMod. Second, we compare to an existing multitier cache originally implemented in Eliom [36].

Conflict-Free Replicated Data Types. Conflict-free replicated data types (CRDT) [42, 43] offer eventual consistency across replicated components for specific data structures, avoiding conflicting updates by design. With CRDTs, updates to shared data are sent asynchronously to the replicas and *eventually* affect all copies. Such *eventually consistent* model [47] provides better performance (no synchronization is required) and higher availability (each replica has a local copy which is ready to use). We reimplemented several CRDTs, publicly available in Scala[5], in LociMod (Table 1).

We discuss the representative case of the `GSet`. G-Sets (grow-only sets) are sets which only support adding elements; elements cannot be removed. A *merge* operation computes the union of two G-Sets. Listing 4a1 shows the G-Set in Scala. `GSet` defines a set `content` (Line 3) and a method to check if an element is in the set (Line 5). Adding an element

[5] `http://github.com/lihaoyi/crdt`

Table 1 Common Conflict-Free Replicated Data Types.

CRDT	Description	Lines of Code		Remote Accesses
		Scala	LociMod	
		local	*distrib.*	
G-Counter	Grow-only counter. Only supports incrementing.	14	15	1
PN-Counter	Positive-negative counter. Supports incrementing and decrementing.	13	14	1
LWW-Register	Last-write-wins register. Supports reading and writing a single value.	10	11	1
MV-Register	Multi-value register. Supports writing a single value. Reading may return a set of multiple values that were written concurrently.	12	13	1
G-Set	Grow-only set. Only supports addition.	7	9	1
2P-Set	Two-phase set. Supports addition and removal. Removed elements cannot be added again.	13	17	2
LWW-Element-Set	Last-write-wins set. Supports addition and removal. Associates each added and removed element to a time stamp.	15	19	2
PN-Set	Positive-negative set. Supports addition and removal. Associates a counter to each element, incrementing/decrementing the counter upon addition/removal.	12	16	2
OR-Set	Observed-removed set. Supports addition and removal. Associates a set of added and of removed (unique) tags to each element. Adding inserts a new tag to the added tags. Removing moves all tags associated to an element to the set of removed tags.	15	18	2

inserts it into the local `content` set (Line 7). Listing 4b presents a multitier module for a multitier G-Set. The implementations are largely similar despite that the LociMod version is distributed and the Scala version is not. The Scala CRDTs are only local. Distributed data replication has to be implemented by the developer (Listing 4a2).

In the LociMod variant, the peer type of the interacting nodes is abstract, hence it is valid for any distributed architecture. The LociMod multitier module can be instantiated by applications for their architecture:

```
 1 @multitier trait EventualConsistencyApp {
 2   @peer type Server <: ints.Node with strings.Node {
 3     type Tie <: Single[Client] with Single[ints.Node] with Single[strings.Node] }
 4   @peer type Client <: ints.Node with strings.Node {
 5     type Tie <: Single[Server] with Single[ints.Node] with Single[strings.Node] }
 6 
 7   @multitier object ints extends GSet[Int]
 8   @multitier object strings extends GSet[String]
 9 
10   on[Server] { ints.add(42) }
11   on[Client] { strings.add("forty-two") }
12 }
```

The example defines a `GSet[Int]` (Line 7) and a `GSet[String]` (Line 8) instance. The `Server` and a `Client` peer are also `ints.Node` and `strings.Node` peers (Line 2 and 4) and are tied to other `ints.Node` and `strings.Node` peers (Line 3 and 5). Thus, both the server (Line 10) and the client (Line 11) can use the multitier module references `ints` and `strings` to add elements to both sets, which (eventually) updates the sets on the connected nodes. The plain Scala version, in contrast, does not offer abstraction over placement.

Listing 4 Conflict-Free Replicated Grow-Only Set.

(a) Scala implementation.

(a1) Traditional G-Set implementation.

```
class GSet[T] {
  val content =
    mutable.Set.empty[T]
  def contains(v: T) =
    content.contains(v)
  def add(v: T) =
    content += v

  def merge(other: GSet[T]) =
    content ++= other.content
}
```

(a2) Example of user code for distribution.

```
1  trait Host[T] {
2    val set = new GSet[T]
3    def add(v: T) = {
4      set.add(v)
5      send(set.content) }
6    def receive(content: T) =
7      set.merge(content)
8  }
```

(b) LociMod implementation.

```
1  @multitier trait GSet[T] extends Architecture {
2    val content = on[Node] {
3      mutable.Set.empty[T] }
4    def contains(v: T) = on[Node] {
5      content.contains(v) }
6    def add(v: T) = on[Node] {
7      content += v
8      remote call merge(content.toSet) }
9    private def merge(content: Set[T]) = on[Node] {
10     this.content ++= content }
11 }
```

In addition to be more concise, the LociMod version exhibits a better design thanks to the combination of multitier programming and modules. In plain Scala, the actual replication of added elements by propagating them to remote nodes is mingled with the user code: The Scala versions of all CRDTs transfer updated values explicitly to merge them on the replicas, i.e., `merge` needs to be public to user code. The *Remote Accesses* column in Table 1 counts the methods that mix replication logic and user code.

Listing 4a2 shows the user code adding an element to the local G-Set (Line 4), sending the content to a remote host (Line 5), receiving the content remotely (Line 6) and merging it into the remote G-Set (Line 7). In contrast, adding an element to the LociMod `GSet` (Listing 4b) directly merges the updated set into all connected remote nodes (Line 8 and 10). The multitier module implicitly performs remote communication between different peers (Line 8), encapsulating the remote access to the replicas, i.e., `merge` is private.

Distributed Caching. We implement a cache that is shared between a client–server architecture. It is modeled after Eliom's multitier cache holding the values already computed on a server [36]. The following code presents the cache using LociMod multitier modules:

```
1  @multitier trait Cache[K, V] extends MultiClientServer {
2    private val table = on[Node] { mutable.Map.empty[K, V] }
3
4    on[Client] {
5      table.asLocal foreach { serverTable =>
6        table ++= serverTable } }
7
8    def add(key: K, value: V) = on[Node] { table += key -> value }
9  }
```

The `Cache` module is implemented for a client–server architecture (Line 1). The `table` map (Line 2) is placed on every `Node`, i.e., on the client and the server peer. The `add` method adds an entry to the map (Line 8). As soon as the client instance starts, the client populates its local map with the content of the server's map (Line 6).

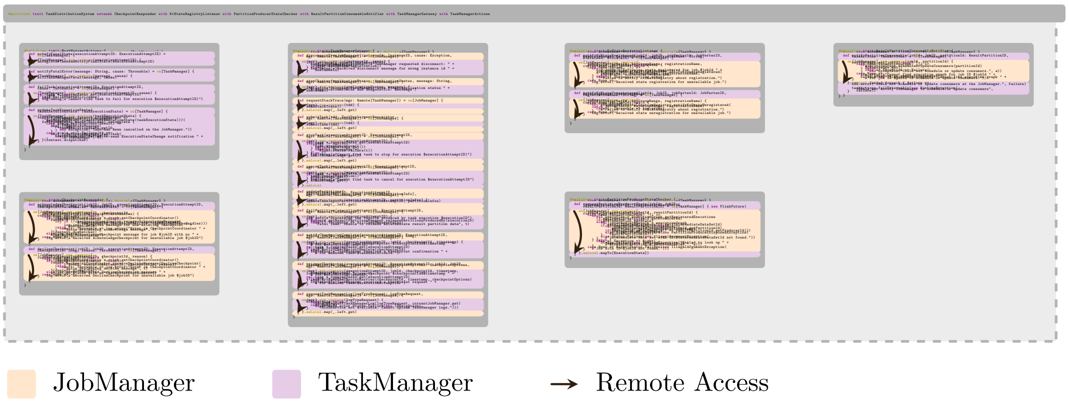


Figure 2 Example communication in Flink using LociMod multitier modules.

LociMod's multitier model is more expressive than Eliom's as it allows the definition of arbitrary peers through placement types. Placement types enable abstraction over placement, as opposed to Eliom, which only supports two fixed predefined places (server and client). LociMod supports Eliom's client–server model (Line 1) as a special case. Thanks to LociMod's abstract peer types, the `Cache` module can also be used for other architectures. For example, we can enhance the `Peer` and `Registry` peers of a P2P architecture with the roles of the client and the server of the `Cache` module by mixing `Cache` and `P2PRegistry` and composing both architectures:

```
@multitier trait P2PCache[K, V] extends Cache[K, V] with P2PRegistry {
  @peer type Registry <: Server { type Tie <: Multiple[Peer] with Multiple[Client] }
  @peer type Peer <: Client { type Tie <: Single[Registry] with Single[Server] with Multiple[Peer] }
}
```

Summary. The case studies demonstrate that, thanks to the multitier module system, distributed data structures can be expressed as reusable modules that can be instantiated for different architectures encapsulating all functionalities needed for the implementation of the data structure (RQ1).

5.3 Apache Flink

The *task distribution system* of the Apache Flink stream processing framework [9], provides Flink's core task scheduling and deployment logic. It is based on Akka actors and consists of six *gateways* (an API that encapsulates sending and receiving actor messages) amounting to ~ 500 highly complex Scala LOC. Gateways wrap method arguments into messages, sending the message and (potentially) receiving a different message carrying a result.

With the current Flink design, however, code fragments that are executed on different distributed components (i.e., for sending and receiving a message), inevitably belong to different actors. The functionalities that conceptually belong to a single gateway are scattered over multiple files in the Flink implementation, breaking modularization. The messages sent by the actors in every gateway are hard to follow for developers because matching sending and receiving operations are completely separated in the code. 19 out of the 23 sent messages are processed in a different compilation unit within another package, hindering the correlation of messages with the remote computations they trigger.

We reimplemented the task distribution system using multitier modules, to cover the complete cross-peer functionalities that belong to each gateway. The resulting modules

Listing 5 Remote communication in Flink.

(a) Original Flink implementation.

(a1) Message definition.

```
package flink.runtime

case class SubmitTask(td: TaskDeployment)
```

(a2) Calling side.

```
package flink.runtime.job

case class SubmitTask(td: TaskDeployment)

class TaskManagerGateway {
  def submitTask(
      td: TaskDeployment,
      mgr: ActorRef) =
    (mgr ? SubmitTask(td))
     .mapTo[Acknowledge]
}
```

(a3) Responding side.

```
package flink.runtime.task

class TaskManager extends Actor {
  // standard Akka message loop
  def receive = {
    case SubmitTask(td) =>
      val task = new Task(td)
      task.start()
      sender ! Acknowledge()
} }
```

(b) Refactored LociMod implementation.

```
package flink.runtime.multitier

@multitier object TaskManagerGateway {
  @peer type JobManager <: {
    type Tie <: Multiple[TaskManager] }
  @peer type TaskManager <: {
    type Tie <: Single[JobManager] }

  def submitTask(
      td: TaskDeployment, tm: Remote[TaskManager]) =
    on[JobManager] {
      (remote(tm) call startTask(td)).asLocal
    }
  def startTask(td: TaskDeployment) =
    on[TaskManager] {
      val task = new Task(td)
      task.start()
      Acknowledge()
    }
}
```

are (1) the TaskManagerGateway to control task execution, (2) the TaskManagerActions to notify of task state changes, (3) the CheckpointResponder to acknowledge checkpoints, (4) the KvStateRegistryListener to notify key-value store changes, (5) the PartitionProducerStateChecker to check of the state of producers and of result partitions and (6) the ResultPartitionConsumableNotifier to notify of available partitions. Since the different cross-peer functionalities of the task distribution system are cleanly separated into different modules, the complete TaskDistributionSystem application is simply the composition of the modules 1–6 that implement each subsystem:

```
1  @multitier trait TaskDistributionSystem extends
2      CheckpointResponder with KvStateRegistryListener with PartitionProducerStateChecker with
3      ResultPartitionConsumableNotifier with TaskManagerGateway with TaskManagerActions {
4    @peer type JobManager <: { type Tie <: Multiple[TaskManager] }
5    @peer type TaskManager <: { type Tie <: Single[JobManager] with Single[TaskManager] }
6  }
```

We mix together the subsystem modules (Line 2 and 3) and specify the architecture of the complete task distribution system (Line 4 and 5). As all subsystems share the same architecture, it is not necessary to specify the architecture in the TaskDistributionSystem module (as we did in the example code). Instead, it suffices to specify the architecture in the mixed-in modules.

Compared to Figure 1b, which merges the functionalities of all subsystems into a single compilation unit, the LociMod version using multitier modules encapsulates each functionality into a separate module. Figure 2 shows the TaskDistributionSystem module (background), composed by mixing together the subsystem modules (foreground). The multitier modules contain code for the JobManager and the TaskManager peer. Arrows represent cross-peer data flow, which is encapsulated within modules and is not split over different modules. Importantly, even modules that place all computations on the same peer (e.g., the module containing only dark violet boxes) define remote accesses (arrows), i.e., different instances of the same peer type (e.g., the dark violet peer) communicate with each other.

It is instructive to look into the details of one of the modules. Listing 5 shows an excerpt of the – extensively simplified – TaskManagerGateway functionality for Flink (left) and its reimplementation in LociMod (right) side-by-side, focusing only on a single remote access of a single gateway. The example concerns the communication between the TaskManagerGateway used by the JobManager and the TaskManager – specifically, the job manager's submission of tasks to task managers. In the actor-based version (Listing 5a), this functionality is scattered over different modules hindering correlating sent messages (Listing 5a2, Line 9) with the remote computations they trigger (Listing 5a3, Line 7–9) by pattern-matching on the received message (Listing 5a3, Line 6). The LociMod version (Listing 5b) uses an intra-module cross-peer remote call (Line 12), explicitly stating the method for the remote computation (Line 16–18). Hence, in LociMod, there is no splitting over different actors as in the Flink version, thus keeping related functionalities inside the same module. The TaskManagerGateway multitier module contains a functionality that is executed on both the JobManager and the TaskManager peer. Further, the message loop of the TaskManager actor of Flink (Listing 5a3), does not only handle the messages belonging to the TaskManager-Gateway (shown in the code excerpt). The loop also needs to handle messages belonging to the other gateways – which execute parts of their functionality on the TaskManager – since modularization is imposed by the remote communication boundaries of an actor.

Summary. In summary, in the case study, the multitier module system enables decoupling of modularization and distribution as LociMod multitier modules capture cross-network functionalities expressed by Flink gateways without being constrained to modularization along network boundaries (RQ2).

6 Related Work

There is a long history of research concerned with proper software modularization mechanisms [33]. We organize related work as follows. First we discuss multitier languages. Second, we present recent advances in module systems. Third, we discuss approaches that partially combine the two solutions. Finally, we provide an overview of related research areas, including languages for distributed systems and component-based software development.

Multitier Languages. Multitier languages emerge in the Web context to remove the separation between client and server code, either by compiling the client side to JavaScript or by adopting JavaScript for the server, too. Hop [40] and Hop.js [41] are dynamically typed languages that follow a traditional client–server communication scheme with asynchronous callbacks. They do not ensure static guarantees for the behavior of the distributed system. In Links [14] and Opa [37], functions are annotated to specify either client- or server-side execution. Both languages also follow the client–server model and feature a static type system. Links' server is stateless for scalability reasons – limiting the spectrum of the supported domains. In StiP.js [34], annotations assign code fragments to the client or the server. Slicing detects the dependencies between each fragment and the rest of the program. In contrast, in LociMod, developers specify placement *in types*, enabling architectural reasoning.

Ur/Web [13], a multitier language for the Web, supports the standard ML module system. By requiring whole-program optimizations to slice the program into client and server parts, Ur/Web modules do not support separate compilation. The Eliom module system [35, 36] is also based on ML modules. It supports *mixed modules* – in Eliom terminology – which can contain declarations for both the server and the client and are similar to LociMod multitier modules that can also contain declarations for different peers (Section 3). Like LociMod modules, Eliom modules feature separate compilation. Due to the restriction to client–server applications, Eliom lacks language abstractions for architectural specifications and distributed system with multiple peers. More interestingly, Eliom modules do not support abstract peer types, hence it is not possible to specify the module functionalities over abstract peers and use such module to specialize the peers in another application (Section 3.2). For the same reason module composition does not support combining different architectures. All approaches above focus on the Web, contrarily to our goal of supporting other architectures. An exception is ML5 [26], a multitier language for generic software architectures: *Possible worlds*, as known from modal logic, address the purpose of placing computations and, similar to LociMod, are part of the type. ML5, however, does not support architecture specification, i.e., it does not allow for expressing different architectures in the language and was anyway applied only to the Web setting so far.

Module Systems. Rossberg and Dreyer design MixML, a module system that supports higher-order modules and modules as first-class values and combines ML modules' hierarchical composition with mixin's recursive linking of separately compiled components [38]. There are some commonalities in the way LociMod uses Scala traits as module interfaces – similar to ML signatures – and objects as module instances – similar to ML structures. Further, traits also support separate compilation. MixML signatures, like standard ML signatures, are structural types. In contrast, mixin composition in Scala operates on traits [15], which are nominal. LociMod, being a Scala embedding, inherits the modularization approach of using traits from Scala, but decouples it from distribution concerns.

Implicit resolution enables retroactive extensibility in the style of Hakell's type classes using the *concept pattern* [31]. The Genus programming language provides modularization abstractions that support generic programming and retroactive extension in the OO setting in a way similar to the concept pattern [49, 50]. Type classes do not support different instances for the same type and the concept pattern's encoding for type classes also requires unambiguous instances (or requires manual disambiguation otherwise). In contrast to Haskell type classes and similar to Genus, LociMod's approach to modularization using Scala traits as modules enables different implementations of the same trait.

Family polymorphism explores definition and composition of module hierarchies. The J& language supports hierarchical composability for nested classes in a mixin fashion [27, 28]. Nested classes are also supported by Newspeak, a dynamically typed object-oriented language [6]. Virtual classes [18] enable large-scale program composition through family polymorphism. Dependent classes [20] generalize virtual classes to multiple dispatching (i.e., class constructors are dynamically dispatched and are multimethods). Mixin composition is supported directly in Scala and it is used in LociMod as a way to compose multitier modules (Section 3.2.2). Virtual classes can be encoded in Scala [30], opening an interesting research direction that investigates multitier modules and family polymorphism.

Programming Languages and Calculi for Distributed Systems. Partitioned Global Address Space languages (PGAS), such as X10 [17], support high-performance parallel execution. These languages define a globally shared address space aiming to reduce boundaries among hosts, similar to multitier languages. In X10, dependent *placed types* [11] identify processing locations ensuring that objects do not cross the boundaries of locations. Instead, our approach abstracts over peer instances (of the same type), to refer uniformly to all similar peers.

Several formal calculi model distributed systems and abstract, to various degrees, over placement and remote communication. The Ambient calculus [10] models concurrent systems with both mobile devices and mobile computation. In Ambient, it is possible to define named, bounded places where computations occur. Ambients can be moved to other places and are nested to model administrative domains and their access control. The Join calculus [19] defines processes that communicate by asynchronous message passing over channels in a way that models an implementation without expensive global consensus. However, we are not aware of higher level modularization abstractions built on top of the Join calculus. CPL [7] is a core calculus for combining services in the cloud computing environment. CPL is event-based and provides combinators that allow safe composition of cloud applications. Similar to LociMod, such combinators are generic with respect to placement and can be parameterized and constrained over other combinators. However, in CPL, there is no notion of architectural specification that can be used to check that multitier module composition adheres to the desired architecture.

Software Architectures and Component-based Software Development. ArchJava [1] unifies architectural definition and implementation in one language. Multitier modules and ties are similar to ArchJava components and connections. Also, similar to LociMod, ArchJava supports connections over a network and gives additional type safety guarantees compared to pure Java, e.g., that values sent over a network are serializable [2]. Different from LociMod modules, which can be parametric and can be mixed in, ArchJava's components can be only composed through connections.

Architecture description languages (ADL) are DSLs designed to support architecture evolution. ADLs define components, connectors, architectural invariants and a mapping of architectural models to an implementation infrastructure [24]. Influenced by ADLs and object-oriented design, component models [16] provide techniques and technologies to build software systems starting from units of composition with contractually specified interfaces and dependencies which can be deployed independently [44]. Component-based development (CBD) aims at separating different concerns throughout the whole software system, defining component interfaces for interaction with other components and mechanisms for composing components, which is similar to LociMod's approach of separating different functionalities into modules providing strong interfaces to other modules. The encapsulated functionalities in LociMod modules, however, can be distributed themselves, whereas CBD in a distributed setting usually models the different components of the distributed system as separate components, forcing developers to modularize along network boundaries.

7 Conclusion

Current multitier languages lack abstractions to properly modularize large code bases. In this paper, we presented LociMod, a *multitier module system* for ScalaLoci that allows developers to modularize multitier code, enabling encapsulation and code reuse. Thanks to abstract peer types, LociMod multitier modules capture abstract patterns of interaction among components in the system, enabling their composition and the definition of module constraints.

Our evaluation on distributed algorithms, distributed data structures, and the Apache Flink big data processing framework, shows that the LociMod's multitier module system is effective in properly modularizing multitier code.

References

1 Jonathan Aldrich, Craig Chambers, and David Notkin. ArchJava: Connecting software architecture to implementation. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, New York, NY, USA, 2002. ACM.

2 Jonathan Aldrich, Vibha Sazawal, Craig Chambers, and David Notkin. Language Support for Connector Abstractions. In *Proceedings of the 17th European Conference on Object-Oriented Programming*, ECOOP '03, Berlin, Heidelberg, 2003. Springer.

3 Henry C. Baker, Jr. and Carl Hewitt. The Incremental Garbage Collection of Processes. *SIGPLAN Notices*, 12(8), 1977.

4 Gilad Bracha and William Cook. Mixin-based Inheritance. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications / European Conference on Object-Oriented Programming*, OOPSLA/ECOOP '90, New York, NY, USA, 1990. ACM.

5 Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the Future Safe for the Past: Adding Genericity to the Java Programming Language. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '98, New York, NY, USA, 1998. ACM.

6 Gilad Bracha, Peter von der Ahé, Vassili Bykov, Yaron Kashai, William Maddox, and Eliot Miranda. Modules As Objects in Newspeak. In *Proceedings of the 24th European Conference on Object-Oriented Programming*, ECOOP '10, Berlin, Heidelberg, 2010. Springer-Verlag.

7 Oliver Bračevac, Sebastian Erdweg, Guido Salvaneschi, and Mira Mezini. CPL: A core language for cloud computing. In *Proceedings of the 15th International Conference on Modularity*, MODULARITY '16, New York, NY, USA, 2016. ACM.

8 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, New York, NY, USA, 2013. ACM.

9 Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache Flink: Stream and batch processing in a single engine. *IEEE Data Engineering Bulletin*, 38, 2015.

10 Luca Cardelli and Andrew D. Gordon. Mobile ambients. *Theoretical Computer Science*, 240(1), 2000.

11 Satish Chandra, Vijay Saraswat, Vivek Sarkar, and Rastislav Bodik. Type Inference for Locality Analysis of Distributed Data Structures. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, New York, NY, USA, 2008. ACM.

12 James Cheney and Ralf Hinze. Phantom Types. Technical report, Cornell University, 2003.

13 Adam Chlipala. Ur/Web: A simple model for programming the web. In *Proceedings of the 42nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, New York, NY, USA, 2015. ACM.

14 Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web programming without tiers. In *Proceedings of the 5th International Conference on Formal Methods for Components and Objects*, FMCO '06, Berlin, Heidelberg, 2007. Springer-Verlag.

15 Vincent Cremet, François Garillot, Sergueï Lenglet, and Martin Odersky. A Core Calculus for Scala Type Checking. In *Proceedings of the 31st International Conference on Mathematical Foundations of Computer Science*, MFCS '06, Berlin, Heidelberg, 2006. Springer-Verlag.

16 Ivica Crnkovic, Severine Sentilles, Vulgarakis Aneta, and Michel R. V. Chaudron. A Classification Framework for Software Component Models. *IEEE Transactions on Software Engineering*, 37(5), September 2011.

17 Mattias De Wael, Stefan Marr, Bruno De Fraine, Tom Van Cutsem, and Wolfgang De Meuter. Partitioned Global Address Space Languages. *ACM Computing Surveys*, 47(4), May 2015.

18 Erik Ernst, Klaus Ostermann, and William R. Cook. A Virtual Class Calculus. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, New York, NY, USA, 2006. ACM.

19 Cédric Fournet and Georges Gonthier. The Reflexive CHAM and the Join-calculus. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, New York, NY, USA, 1996. ACM.

20 Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. Dependent Classes. In *Proceedings of the 22nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, New York, NY, USA, 2007. ACM.

21 D. S. Hirschberg and J. B. Sinclair. Decentralized Extrema-finding in Circular Configurations of Processors. *Communications of the ACM*, 23(11), 1980.

22 Barbara Liskov. Keynote Address – Data Abstraction and Hierarchy. In *Addendum to the Proceedings on Object-Oriented Programming Systems, Languages and Applications (Addendum)*, OOPSLA '87, New York, NY, USA, 1987. ACM.

23 Matthew L. Massie, Brent N. Chun, and David E. Culler. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, 30(7):817–840, 2004.

24 Nenad Medvidovic, David S. Rosenblum, and Richard N. Taylor. A Language and Environment for Architecture-based Software Development and Evolution. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, New York, NY, USA, 1999. ACM.

25 R. Milner, L. Morris, and M. Newey. A Logic for Computable Functions with Reflexive and Polymorphic Types. In *Proceedings of the Conference on Proving and Improving Programs*, Arc-et-Senans, 1975.

26 Tom Murphy, VII., Karl Crary, and Robert Harper. Type-safe Distributed Programming with ML5. In *Proceedings of the 3rd Conference on Trustworthy Global Computing*, TGC '07, Berlin, Heidelberg, 2008. Springer-Verlag.

27 Nathaniel Nystrom, Stephen Chong, and Andrew C. Myers. Scalable Extensibility via Nested Inheritance. In *Proceedings of the 19th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '04, New York, NY, USA, 2004. ACM.

28 Nathaniel Nystrom, Xin Qi, and Andrew C. Myers. J&: Nested Intersection for Scalable Software Composition. In *Proceedings of the 21st ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, New York, NY, USA, 2006. ACM.

29 Martin Odersky, Guillaume Martres, and Dmitry Petrashko. Implementing Higher-kinded Types in Dotty. In *Proceedings of the 7th ACM SIGPLAN Symposium on Scala*, SCALA '16, New York, NY, USA, 2016. ACM.

30 Martin Odersky and Matthias Zenger. Scalable Component Abstractions. In *Proceedings of the 20th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, New York, NY, USA, 2005. ACM.

31 Bruno C.d.S. Oliveira, Adriaan Moors, and Martin Odersky. Type Classes As Objects and Implicits. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, New York, NY, USA, 2010. ACM.

32 Diego Ongaro and John Ousterhout. In Search of an Understandable Consensus Algorithm. In *Proceedings of the 2014 USENIX Annual Technical Conference*, USENIX ATC '14, Berkeley, CA, USA, 2014. USENIX Association.

33 D. L. Parnas. On the Criteria to Be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12), December 1972.

34 Laure Philips, Coen De Roover, Tom Van Cutsem, and Wolfgang De Meuter. Towards Tierless Web Development Without Tierless Languages. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2014, New York, NY, USA, 2014. ACM.

35 Gabriel Radanne and Jérôme Vouillon. Tierless Modules, 2017.

36 Gabriel Radanne and Jérôme Vouillon. Tierless Web Programming in the Large. In *Companion Proceedings of the The Web Conference 2018*, WWW '18, Republic and Canton of Geneva, Switzerland, 2018. International World Wide Web Conferences Steering Committee.

37 David Rajchenbach-Teller and Franois-Régis Sinot. Opa: Language support for a sane, safe and secure web. In *Proceedings of the OWASP AppSec Research*, 2010.

38 Andreas Rossberg and Derek Dreyer. Mixin' Up the ML Module System. *ACM Transactions on Programming Languages and Systems*, 35(1), April 2013.

39 Nicola Santoro. *Design and analysis of distributed algorithms*, volume 56. John Wiley & Sons, 2006.

40 Manuel Serrano, Erick Gallesio, and Florian Loitsch. Hop: A language for programming the web 2.0. In *Companion to the 21th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Companion to OOPSLA '06, New York, NY, USA, 2006. ACM.

41 Manuel Serrano and Vincent Prunet. A Glimpse of Hopjs. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP '16, New York, NY, USA, 2016. ACM.

42 Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Research Report RR-7506, Inria – Centre Paris-Rocquencourt; INRIA, January 2011.

43 Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free Replicated Data Types. In *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*, SSS '11, Berlin, Heidelberg, 2011. Springer-Verlag.

44 Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002.

45 Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton, Jr. N Degrees of Separation: Multi-dimensional Separation of Concerns. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, New York, NY, USA, 1999. ACM.

46 Kresten Krab Thorup. Genericity in Java with Virtual Types. In *Proceedings of the 11th European Conference on Object-oriented Programming*, ECOOP '97, Berlin, Heidelberg, 1997. Springer-Verlag.

47 Werner Vogels. Eventually Consistent. *Communications of the ACM*, 52(1), January 2009.

48 Pascal Weisenburger, Mirko Köhler, and Guido Salvaneschi. Distributed System Development with ScalaLoci. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA '18), 2018.

49 Yizhou Zhang, Matthew C. Loring, Guido Salvaneschi, Barbara Liskov, and Andrew C. Myers. Lightweight, Flexible Object-oriented Generics. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, New York, NY, USA, 2015. ACM.

50 Yizhou Zhang and Andrew C. Myers. Familia: Unifying Interfaces, Type Classes, and Family Polymorphism. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA), October 2017.

Scopes and Frames Improve Meta-Interpreter Specialization

Vlad Vergu
Delft University of Technology, Delft, The Netherlands
v.a.vergu@tudelft.nl

Andrew Tolmach
Portland State University, Portland, OR, USA
tolmach@pdx.edu

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

Abstract

DynSem is a domain-specific language for concise specification of the dynamic semantics of programming languages, aimed at rapid experimentation and evolution of language designs. To maintain a short definition-to-execution cycle, DynSem specifications are meta-interpreted. Meta-interpretation introduces runtime overhead that is difficult to remove by using interpreter optimization frameworks such as the Truffle/Graal Java tools; previous work has shown order-of-magnitude improvements from applying Truffle/Graal to a meta-interpreter, but this is still far slower than what can be achieved with a language-specific interpreter. In this paper, we show how specifying the meta-interpreter using *scope graphs*, which encapsulate static name binding and resolution information, produces much better optimization results from Truffle/Graal. Furthermore, we identify that JIT compilation is hindered by large numbers of calls between small polymorphic rules and we introduce *rule cloning* to derive larger monomorphic rules at run time as a countermeasure. Our contributions improve the performance of DynSem-derived interpreters to within an order of magnitude of a handwritten language-specific interpreter.

2012 ACM Subject Classification Software and its engineering → Interpreters

Keywords and phrases Definitional interpreters, partial evaluation

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2019.4

Funding This research was partially funded by the NWO VICI *Language Designer's Workbench* project (639.023.206) and by a gift from the Oracle Corporation.

Acknowledgements We thank the anonymous reviewers for their feedback on previous versions of this paper, and we thank Laurence Tratt for his guidance on obtaining reliable runtime measurements and analyzing the resulting time series.

1 Introduction

A *language workbench* [9, 36] is a computing environment that aims to support the rapid development of programming languages with a quick turnaround time for language design experiments. Meeting that goal requires that (a) turning a language design idea into an executable prototype is easy; (b) the delay between making a change to the language and starting to execute programs in the revised prototype is short; and (c) the prototype runs programs reasonably quickly. Moreover, once the language design has stabilized, we will need a way to run programs at production speed, as defined for the particular language and application domain.

33rd European Conference on Object-Oriented Programming (ECOOP 2019).
Editor: Alastair F. Donaldson; Article No. 4; pp. 4:1–4:30
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Semantics specification languages such as Redex [10], K [30], and DynSem [34] provide abstractions for directly expressing the operational semantics rules of a language under design. For example, DynSem supports concise specification based on the implicitly modular operational semantics approach, which requires mentioning semantic components such as environments and stores only in rules that actually interact with those components [23, 22]. Such high-level specification languages reduce the effort of *defining* an object language. But how best to generate an executable prototype from such a definition?

Since we typically do not need the prototype to run especially fast, one natural approach is to generate an interpreter for the object language. For example, the original DynSem implementation [34] generates interpreters in Java. However, this approach requires a sequence of steps – generating code from the operational semantics definition, compiling that generated code, starting up a JVM, and running the generated interpreter on an object language AST – that altogether take on the order of a minute, even for very small language definitions. This delay inhibits workbench users from incorporating prototype generation and testing into their design iteration loop.

The standard solution to making a translated language more agile is to interpret it instead. An interpreter for an interpreter specification language is a *meta-interpreter*, resulting in two layers of interpretation: the meta-interpreter reads the AST of a specification and the AST of an object program, and interprets the rules from the specification, which in turn interpret the object language AST. While this reduces the code-to-run cycle, it increases the execution time of object programs by at least an order of magnitude, potentially limiting the scalability of tests or experiments. So, it seems that we either get slow interpreter generation or slow meta-interpreter execution. Can we get fast interpreter generation *and* fast interpreter execution?

There is reason to hope that we can: trace-based optimization frameworks such as RPython [4] and partial evaluation frameworks such as Truffle/Graal [38] have been successful in bringing the benefits of JIT compilation to (suitably instrumented) interpreters. We have been exploring whether such approaches will also work for meta-interpreters. In prior work [35] we demonstrated that specializing a meta-interpreter for DynSem using the Truffle/Graal framework can lead to an order of magnitude speed-up over a naive meta-interpeter. However, we were curious about whether we could do better still. Can we get close to the performance of a manual implementation of an object-language interpreter, or even of a production-quality object-language compiler?

In this paper, we report progress towards this goal. We show that the combination of the use of a uniform memory model and cloning semantics rules leads to a meta-interpreter for DynSem with a performance that is within a geometric mean factor of 4.7 of a hand-written object-language-specific interpreter for a small set of benchmarks on a simple object language. Both interpreters are implemented using the Truffle AST interpreter framework [40] and run with the Graal JIT compiler for the Java VM [38], which aggressively inlines stable method calls into efficient machine code. This work makes the following contributions:

- *Memory representation using "scopes and frames"*: The specifications of Vergu et al. [35] use environments for the representation of memory (environment and store) as is common in dynamic semantics specifications. However, this memory representation is language-specific and has high performance overhead. In this paper we use the "scopes and frames" approach [28], a uniform (language parametric) model for the representation of memory in dynamic semantics specifications based on scope graphs [25, 32]. By mapping frames onto Truffle's Object Storage Model, we can piggy-back on the optimizations for that representation.

- *Rule cloning*: The units of execution in a DynSem specification are reduction rules for language constructs. Since the same rule is used for all occurrences of a language construct in a program, the specializer considers them as *polymorphic*, with limited specialization as result. By cloning rules for each call site, rules become monomorphic, allowing Graal to inline them.
- *Evaluation*: We have evaluated the approach using the Tiger language [2]. We compare the performance of three variants of DynSem specifications for Tiger and a Tiger-specific interpreter implemented in Java, all running on the Graal VM. The variants compare memory representation (environments vs scopes-and-frames) and inlining vs not inlining. The results suggest that this is a viable approach, with performance of meta-interpretation using inlining and scopes-and-frames within an order of magnitude of the language-specific interpreter.

Outline. We proceed as follows. In the next section, we describe the DynSem specification language and review the Truffle/Graal framework. In Section 3 we discuss the design of the (hybrid) meta-interpreter. In Section 4 we review the "scopes-and-frames" approach, demonstrate its application in DynSem specifications, and discuss the mapping of frames to Truffle's Object Storage Model. In Section 5 we discuss the design of rule cloning in the meta-interpreter driven by a light-weight binding time analysis. In Section 6 we present the set-up of the evaluation experiment and discuss the results. In Section 7 we discuss related and future work.

2 Background

In this section we discuss the background on the DynSem specification language and the Truffle and Graal framework.

2.1 DynSem

DynSem [34] is a meta-DSL for specifying the dynamic semantics of programming languages. It is included in the Spoofax Language Workbench [17] and is a part of a larger effort to derive programming environments from high-level specifications [36]. In DynSem, programs are represented as terms and program execution is modeled as reduction of program terms to value terms. We illustrate the key concepts of DynSem with the example in Figure 1.

Signatures. The structure of terms is defined by means of an algebraic signature, which defines the sorts (types) of terms, term constructors, typed reduction arrows, and semantic components. Figure 1a illustrates these concepts for a subset of the term signatures of Tiger [2]. Tiger is a simple programming language originally invented for teaching about compilers; it is a statically typed language and has let bindings, functions, records, control-flow constructs and arrays. Figure 1a declares two sorts of terms: `Exp` for program expressions, and `Val` for value terms. A constructor declaration defines the arity and types of terms that a constructor can be applied to. For example, the `Plus` constructor is used to construct terms of the form `Plus(e1, e2)` where the subterms `e1` and `e2` are terms of sort `Exp`. Note that just like program expressions, value terms are represented by a sum type to represent different kinds of values, unified in the `Val` sort. The example defines integer and closure values.

An *arrow* defines the source and target sort of a reduction. For example, the `Exp` $\longrightarrow$ `Val` arrow states that `Exp` terms can be reduced to `Val` terms using the $\longrightarrow$ arrow. Semantic components are used to represent the run-time state of programs. In the example, semantic components for environments `E` (mapping identifiers to locations) and heaps (stores) `H` (mapping locations to values) are defined.

```
signature
  sorts Exp Val
  constructors
    Plus: Exp * Exp → Exp
    Call: Id * Exp → Exp
    IntV: Int → Val
    ClosureV: Id * Exp * E → Val
  arrows
    Exp ⟶ Val
  components
    E : Map(Id, Int)
    H : Map(Int, Val)
```

(a)

```
E ⊢ e1 :: H1 ⟶ IntV(i) :: H2;
E ⊢ e2 :: H2 ⟶ IntV(j) :: H3;
IntV(addI(i, j)) ⇒ v
-------------------------------------
E ⊢ Plus(e1, e2) :: H1 ⟶ v :: H3
```

(b)

```
e1 ⟶ IntV(i); e2 ⟶ IntV(j)
------------------------------------
Plus(e1, e2) ⟶ IntV(addI(i, j))
```

(c)

```
readVar(f) ⟶ ClosureV(arg, efun, E);
e ⟶ varg;
E {arg ↦ varg, E} ⊢ e ⟶ v
----------------------------------------
Call(f, e) ⟶ v
```

(d)

```
signature
  arrows
    readVar(String)  ⟶ Val
    lookup(String)   ⟶ Int
    read(Addr)       ⟶ Val
    write(Addr, Val) ⟶ Val
    allocate(Val)    ⟶ Int
```

(e)

```
readVar(x) ⟶ read(lookup(x))

E ⊢ lookup(x) ⟶ E[x]

read(a) :: H ⟶ H[a]

write(a, v)::H ⟶ v :: H {a ↦ v,H}

fresh ⇒ a; write(a, v) ⟶ _
-------------------------------
allocate(v) ⟶ a
```

(f)

Figure 1 (a) Algebraic term signatures in DynSem. (b) Fully elaborated rule for arithmetic addition and (c) its concise equivalent with implicit propagation of semantic components. (d) Semantics of a unary function call. (e) Signatures of auxiliary meta-functions for environment and store operations and (f) their corresponding rules.

DynSem specifications are statically checked with respect to signatures. The checker ensures that term patterns in rules are consistent with constructor declarations and that arrow arguments are of the right sort.

Rules. Reduction rules define the dynamic semantics of programs by reduction of program terms to value terms. A rule has the form

```
prem1; prem2; ...
-----------------
lhs ⟶ rhs
```

where the conclusion is an arrow declared in the signature. It defines that a term matching `lhs` is reduced to the instantiation of term `rhs`, provided that the premises `prem1; prem2; ...` succeed. Premises are either recursive arrow applications or pattern matches. An arrow application premise `lhs ⟶ rhs` instantiates the pattern `lhs` with the substitutions for meta-variables from the left-hand side of the conclusion or from earlier premises, reduces it with the arrow, and matches the result against the `rhs` pattern. A pattern matching premise `lhs ⇒ rhs` instantiates the pattern `lhs`, which may possible involve application of meta-functions (see below), and matches it to the pattern `rhs`. Arrows are usually defined in a big-step style [16]. That is, a rule reduces a program term to a value term in one step, using recursive invocation of arrows in the premises. This is illustrated in Figure 1c, which defines the reduction of `Plus(e1, e2)` terms with the ⟶ arrow by completely reducing the argument terms to value terms. The right-hand side of the conclusion constructs the resulting value term by using the `addI` meta-function.

Semantic Components. The rule in Figure 1c does not account for the evaluation of an expression in the context of an environment binding variables in scope and a heap storing values with longer lifetimes. DynSem supports the propagation of such contextual information by means of so called *semantic components*, which are distinguished in read-only components and read-write components. A read-only component is mentioned to the left of the ⊢ symbol, and propagates downwards (environment semantics). A read-write component is mentioned after the `::` symbol and is threaded through the evaluation of the relation.

The rule in Figure 1b propagates semantic components `E` and `H` through the evaluation of the sub-expressions of `Plus`. Semantic component `E` (representing a variable environment) propagates as a read-only semantic component, while component `H` (representing a store) is threaded through the computation and returned from the rule.

A rule only has to explicitly mention those semantic components that it modifies; other components can be left implicit. The rule of Figure 1b modifies neither environment nor store and both may therefore be left implicit, as shown in Figure 1c. A static analysis infers which semantic components must be propagated and informs a source-to-source transformation that makes all components explicit.

Meta-Functions. DynSem allows standalone units of semantics to be separately defined as meta-functions. This supports reuse across rules and promotes concise rules. The semantics of a unary function call given in Figure 1d illustrate the use of meta-functions in DynSem.

Meta-functions `readVar`, `lookup`, `read`, etc. with their signatures and semantics of Figure 1e and Figure 1f, respectively, provide a library of memory operations. The operations are used, for example, to `lookup` the heap address of a variable in the environment by its name, and to `read` the value associated with this address from the heap. The `readVar` combines these two operations in a single meta-function which is used, for example, in the `Call` rule of Figure 1d to retrieve the function closure.

2.2 Truffle and Graal

We use Truffle [40] and Graal [38] as runtime frameworks for the execution of DynSem specifications. For a definitive guide we refer the reader to the Truffle and Graal literature [40, 39, 14, 38]. Throughout this section it is useful to keep in mind that a runtime derived from a DynSem specification is an interpreter of DynSem specifications that consumes an object-language specification and a program to execute, as depicted in the architecture overview Figure 6. We provide an overview of this in Section 3.

Truffle Interpreters. Truffle [40] is a Java framework for implementing high-performance interpreters, in particular interpreters for dynamic languages. Truffle interpreters are AST interpreters. In an AST interpreter the syntactic structure of the program determines the organization of the interpreter. Each AST node implements the semantics of the language construct it represents. In a typical Truffle interpreter the parser instantiates the AST of the interpreter given a particular program. Execution in the interpreter flows downwards in the tree and results flow upwards. Truffle provides the logistics for implementing interpreter nodes and maintaining the AST.

Figure 2 shows the skeletons of the two base classes that provide the basis for implementing language-specific nodes. A `Node` is the basic building block of a Truffle interpreter. The language developer extends the `Node` class to give semantics to language constructs. The `Node` class provides facilities for constructing and modifying trees of nodes and for traversing the tree, downwards and upwards. For example, a node for binary addition has two children

```
abstract class Node ... {
  Node parent;

  Node getParent() {
    return parent;
  }

  RootNode getRootNode() {
    Node rootNode = this;
    while (rootNode.getParent() != null) {
      rootNode = rootNode.getParent();
    }
    return (RootNode) rootNode;
  }

  Node replace(Node newNode){ ... }

  Node adopt(Node child) { ... }
}

abstract class RootNode ... {
  abstract Object execute(VirtualFrame f);
}
```

Figure 2 Skeletons of Truffle Node and RootNode classes and logistics for traversing the AST upwards.

nodes, one for each of its subexpressions, and provides an execution method that performs the addition and returns the result. If the implemented language has variables, the execute method is parameterized with an environment-like data structure, called a `Frame`, that contains the variables in scope at that location of the program.

An interpreter node without a parent is a `RootNode`. Each tree of interpreter nodes has a root, which is an entry point for execution and typically corresponds to a function in the object program. Multiple interpreter trees exist at run time, typically one for each function of a program. Each root node is parameterized by a frame descriptor defining the structure of the `Frame` that is propagated downwards during evaluation. For example, if a root node corresponds to a function, its frame descriptor defines the variables bound in the body of the function. The Truffle runtime uses the frame descriptor to instantiate a frame to be used when calling the function.

Specializing Truffle Interpreters. Truffle interpreters are particularly suited to dynamic languages because the AST structure of the interpreter allows each node to self-optimize based on runtime information. The core idea is that the interpreter AST evolves at run time to a more efficient implementation based on runtime values. For example, the plus operator of a dynamic language may embed semantics for both arithmetic addition and string concatenation, and at runtime specialize itself to one of these two semantics based on the (dynamic) values of its operands. A node may replace itself by a more specific variant by using the `replace` method, which updates the node's parent to point to the new variant. Alternatively, a node may decide to replace one of its children by a more efficient one, or adopt a new child altogether, by using the `adopt` method. Truffle provides a set of class and method annotations, collectively known as the Truffle DSL [14], that reduce the implementation effort (and boilerplate) of developing node specializations. The annotations drive a (compile-time) code generator which emits highly-efficient implementations of behavior specialization and inline caching.

The Graal JIT Compiler. Graal [38] is a high-performance JIT compiler for the Java VM with powerful partial evaluation and component inlining phases. Graal aggressively inlines stable method calls in order to generate efficient machine code. Runtime decisions about what calls are inlined are based on the outcome of a cost-benefit analysis. Truffle and Graal are designed to work together to obtain JIT-compiled Truffle interpreters with little effort. Graal treats each Truffle AST root node as a single compilation unit, i.e. Graal compiles root nodes individually. Once a Truffle interpreter tree stabilizes (i.e. node rewriting has stopped) Graal inlines all method calls of the nodes which are under a common root and emits machine code for that tree. A `Frame` that is never stored in a class field can remain virtualized – `VirtualFrame`. Since all the execution methods are inlined, the virtual frame can be eliminated, resulting in highly efficient machine code. If, after compilation, a node has to be re-specialized, for example due to a specialization that is no longer valid, the VM transfers execution of the entire executing tree back to interpreted code, disregards the machine code, and the tree is recompiled to machine code once its structure has stabilized again. The size of a tree therefore greatly affects the cost-benefit analysis of JIT compilation for that subtree. As we discuss in Sections 5 and 6, small trees compile cheaply but with little benefit, whereas JIT-compiling large trees delivers better peak performance but at an increased risk of costly recompilation.

3 Meta-Interpreters

The DynSem runtime of Vergu et al. [35] is a meta-interpreter, i.e. it interprets dynamic semantics specifications of a language. Figure 3 gives a macroscopic view of the components at play in meta-interpretation. A DynSem specification undergoes lightweight source-to-source transformations (syntactic desugaring, semantic component explication, factorization, etc.) to make it amenable to interpretation. The meta-interpreter enacts the desugared DynSem specification with respect to a program's AST in order to evaluate the program. Each rule of the specification is loaded in the meta-interpreter as a callable function. The body of a function is made up of meta-interpreter nodes that implement the semantics of the DynSem instructions used within the rule. This results in two layers of interpretation: the meta-interpreter interprets the rules of the specification which in turn interpret the object language AST.

While meta-interpretation reduces the code-to-run cycle, it increases the execution time of object programs, potentially limiting the scalability of tests or experiments. So, it seems that we either get slow interpreter generation or slow interpreter execution. Motivated by the goal of having fast interpreter generation *and* fast interpreter execution, the DynSem meta-interpreter is implemented as a Truffle [40] AST interpreter and executes on an Oracle Graal VM [38]. Much of the original meta-interpretation research [35] is focused on determining an interpreter morphology and providing runtime information to the Graal JIT such that it can remove the meta-interpreter layer.

Hybrid Meta-interpretation. Because meta-interpretation is slowed down by interpretation of generic term operations (pattern matching and construction), and because term operations for an object language are specific to that language, the DynSem meta-interpreter replaces generic term operations with statically generated language-specific term operations, which are derived from the DynSem specification of the language. Vergu et al. named the combination of specification meta-interpretation and generated term operations *hybrid*

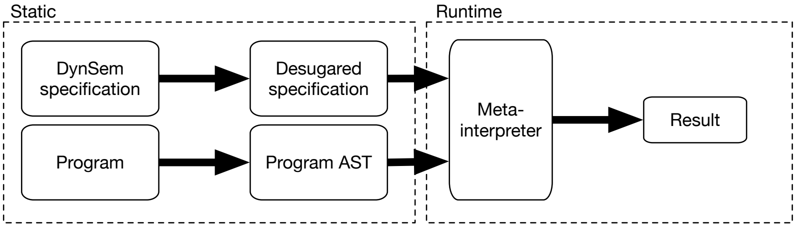
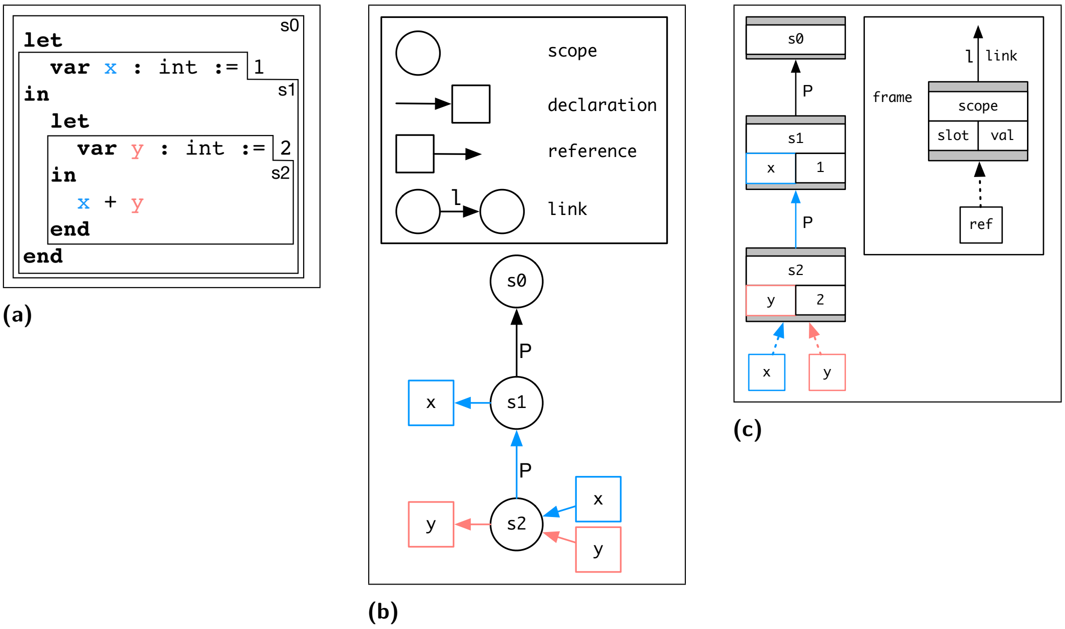


Figure 3 Overview of meta-interpretation.

meta-interpretation [35]. The original hybrid meta-interpreter starts up with generic term operations that immediately specialize themselves to the language-specific operation at their first execution, which is essentially a form of local JIT compilation.

Meta-interpreter Modifications. We apply the improvements presented in this paper to the DynSem hybrid meta-interpreter with two small modifications. First, we replace the rule dispatch mechanism by a simple rule call mechanism with an inline cache. The simplified rule call mechanism looks up the callee rule in the registry of rules and invokes it. The inline cache allows the call mechanism to remember callee rules so that the lookup is avoided in future calls. We chose to make this simplifying refactoring to allow a redesign of the rule call specialization mechanism, as we will show in Section 5. Second, we refactored the meta-interpreter to directly use the generated term operations instead of lazily replacing generic ones at run time. At best this leads to one less iteration required until warmup, but it simplifies interpreter initialization. The change does not have an effect after warmup and thus has no impact on the evaluation of the contributions of this paper.

Limitations of Name Resolution with Maps. In the original DynSem work [34], typical language specifications model name binding, resolution and program memory using abstractions for environments (mapping names to addresses) and stores (mapping addresses to values). Thus, for example, every reference to an object program variable involves a string-based lookup of the variable name in an environment data structure. Environments and stores are themselves implemented using ordinary DynSem reduction rules on top of a built-in type of persistent (i.e. functional) maps. The approach has previously been identified as a DynSem performance bottleneck [35]. The performance penalty is due in the first instance to the inherent cost of (hash-)map operations. But a more fundamental issue is that the JIT compiler cannot see the algorithms of the underlying maps, which means it cannot comprehend the operation of environments, and hence cannot comprehend name resolution in object programs. Observing and optimizing name resolution is, however, an essential ingredient in JIT compilation. Moreover, to write an environment-based DynSem specification, a language developer must define name binding and resolution in the dynamic semantics. Typically, they do this by writing higher-level DynSem meta-functions, such as variable lookup, that abstract from the low-level details of environment manipulation and encapsulate the object language's name resolution policy (Section 2.1). Unfortunately, such meta-functions are typically language-specific, making them difficult to reuse.



```
let
  var x : int := 1
in
  let
    var y : int := 2
  in
    x + y
  end
end
```

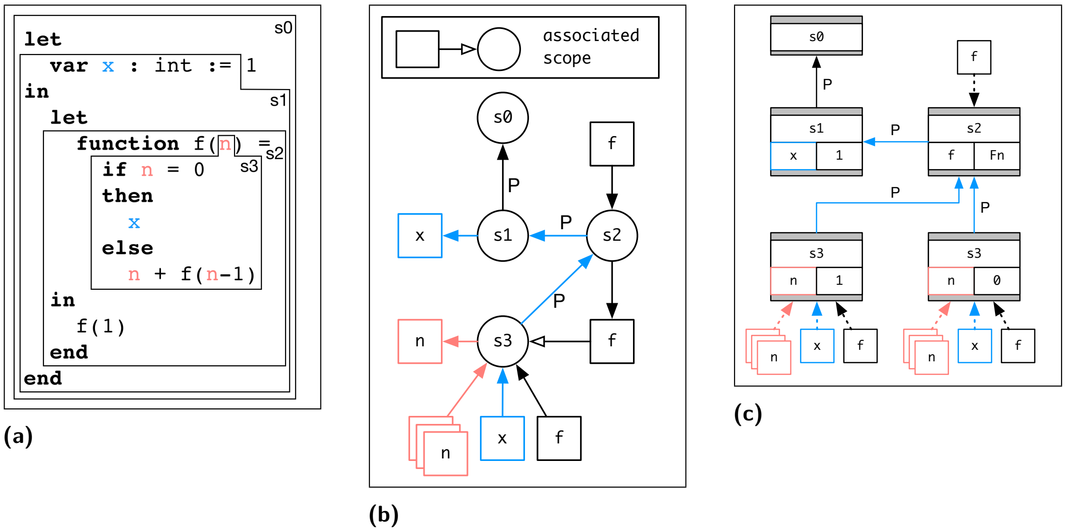
Figure 4 (a) Program with nested let bindings. The labelled box surrounding a code fragment indicates the scope the fragment resides in. Declarations and references for the same name are shown in the same color. (b) The scope graph describing the name binding structure of the program. Colors highlight name resolution paths from references to declarations. (c) Heap of frames at the end of program evaluation.

4 Scopes and Frames

To address the performance issues of the use of maps for the representation of name binding, we adopt the *scopes-and-frames* approach of Poulsen et al. [28]. In this section, we provide an overview of the previous work on *name resolution with scope graphs* and *frames* to represent scopes at run time. Then we discuss the extension of DynSem with support for scopes-and-frames and its implementation in terms of Truffle's Object Storage Model.

4.1 Name Resolution with Scope Graphs

Our approach is based on the theoretical framework of a *resolved scope graph* [25], which is a distillation of a program's name-binding structure that supports name resolution in a mostly language-independent way. Consider the small program of Figure 4a and its corresponding resolved scope graph in Figure 4b. Scopes are code regions that behave uniformly with respect to name binding and resolution. They are marked in code with labelled boxes and are shown in the scope graph as named circles. Scopes contain declarations, shown as named boxes with an incoming arrow, and references, shown as named boxes with an outgoing arrow. Visibility inclusion between scopes is shown as a labelled directed arrow between scopes. For example, the fact that declarations of the outer **let** are visible in the inner **let** is indicated by the arrow from scope s2 to s1. Arrow labels characterize visibility inclusion relationships. In this case the P label indicates a lexical parent inclusion relationship. Resolving a name involves determining a path in the graph from the scope containing the name reference to the scope containing its declaration. The reference y resolves to the local declaration by the red path in the scope graph, while reference x resolves to the declaration in the parent scope by the blue path. The name resolution of a program is the set of paths which uniquely relate each reference to a declaration.



```
let
  var x : int := 1
in
  let
    function f(n) =
      if n = 0
      then
        x
      else
        n + f(n-1)
  in
    f(1)
  end
end
```

Figure 5 (a) Program with nested let bindings and a recursive function. (b) The scope graph describing the name binding structure of the program. (c) Heap of frames at the end of the evaluation of the program.

The example in Figure 5 shows how function scopes are modeled using scope graphs. These examples demonstrate examples of lexical scope, in which declarations in inner scopes shadow declarations in outer scopes. The Tiger language, which is used for the experiments in this paper, also has records and recursive type definitions. However, scope graphs are not limited to these patterns, but rather support the formalization of a wide range of name binding patterns, including variations of let bindings (sequential, parallel, recursive), modules with (recursive and transitive) imports, classes with inheritance, packages [25, 24], type-dependent name resolution [32], and structural and generic types [33]. The framework allows modeling a variety of visibility policies by configuring path specificity and path well-formedness predicates [32].

Frames. Poulsen et al. [28] provide the theoretical foundation for using a resolved scope graph to describe the layout of frames in a heap and the semantics of the base memory operations: allocation, lookup, access, and update. Declarations and references of a scope provide a recipe for constructing a memory frame at run time. A heap of frames, for example that of Figure 4c, results from program evaluation. A new frame is created when evaluation enters a new scope. The structure of the frame is determined by the declarations and references in its describing scope, which become slots of the frame. Newly created frames are linked to existing frames in accordance to their scope links. In the frame heap, references are related to slots by the name resolution path from the scope graph. Resolving a reference to a slot is performed by traversing frame links in accordance with the path. A new frame is created each time evaluation enters a scope. We illustrate this in the program of Figure 5, where the function body is evaluated in a fresh frame for each function call. Note that for a recursive function like this, multiple frames for a single scope can exist simultaneously.

Architecture. In the rest of this section we describe how we incorporate scopes-and-frames into DynSem. Figure 6 gives an architectural overview of the approach. The static semantics of the object language is described in the constraint-based NaBL2 [32] language. Notably, it

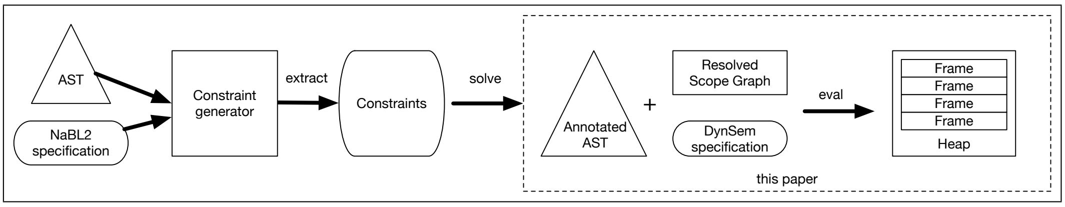


Figure 6 Architecture of the approach: static analysis on a program's AST via constraints produces an AST with explicit name and type information, which is the input for interpretation in accordance with a dynamic semantics specification.

uses scope graphs to represent the binding structure of the programs. The result of type checking with an NaBL2 specification is an annotated AST and a resolved scope graph. The DynSem specification for the object language uses frames based on scopes in the scope graph to represent memory and paths in the scope graph to resolve names to declarations in the frame heap.

4.2 Static Semantics with NaBL2

The scope graph for a program is constructed during type checking. The type checker derived from an NaBL2 specification generates constraints for an object program, which are solved by a language-independent constraint solver. We give a brief introduction to static semantics specifications with NaBL2 [32] using the rules in the left column of Figure 8 for the subset of the Tiger language used in the examples in Figure 4 and Figure 5. The signature of the abstract syntax of this subset is defined in Figure 7. (For the sake of conciseness of the presentation we have simplified the constructs in the subset to unary instead of n-ary let bindings and function definitions and calls. Furthermore, we use type equality instead of subtyping. For the experiments we have used the full Tiger language.)

An NaBL2 rule of the form `Srt[[C(e1, e2, ...) ^ (s) : t ]]:= C.` specifies that the (abstract syntax of) language construct `C(e1, e2, ...)` in the context of scope `s` has type `t` provided that the constraint `C` is satisfied. The constraint in the body of a rule is typically a conjunction of multiple simpler constraints. Constraints include recursive invocations `Srt[[C(e1, e2, ...) ^ (s) : t ]]` of constraint rules on subterms, unification constraints on constraint variables, and scope graph constraints, which support the introduction of a new scope (**`new`** `s`), the definition of a scope edge (`s` $\xrightarrow{P}$`s'`), the definition of a declaration in a scope (`o` ← `s`), the definition of a reference in a scope (`o` → `s`), the association of a type with an occurrence (`o : t`), and the resolution of a reference to a declaration (`o` ↦`d`). Here `o` denotes an *occurrence* `NS{x}` consisting of a namespace `NS` and a concrete occurrence of a name `x` in a program. The NaBL2 constraint `@l.scopeOf := s'` attaches the newly created scope `s'` as a property on the program term to make it available to the runtime.

For example, the rule for `Let` introduces a new scope `s_let`, links it to the parent scope, and passes it on as the binding scope for the declaration and as the scope of its body expression. The rule for `VarDec` introduces the variable `x` as a bound variable in the binding scope `s'` and associates the type of the initializer expression with it. The rule for `Var` declares `x` as a reference in the scope of the variable, resolves the name to a declaration `d`, and retrieves the associated type `ty`. The rule for `FunDec` creates a new scope `s_fun` for the body of the function and declares the formal parameter `x` as a declaration in that scope.

```
signature
  sorts Id
  sorts Dec constructors
    VarDec : Id * Type * Exp → Dec
    FunDec : Id * Id * Type * Type * Exp → Dec
  sorts Exp constructors
    Let    : Dec * Exp → Exp
    Var    : Id → Exp
    Call   : Id * Exp → Exp
    Plus   : Exp * Exp → Exp
    Minus  : Exp * Exp → Exp
```

Figure 7 Signature for an adapted subset of Tiger.

```
Exp[[ l@Let(dec, e) ^ (s) : ty ]] :=
  new s_let, s_let -P-> s,
  @l.scopeOf := s_let,
  Dec[[ dec ^ (s_let, s) ]],
  Exp[[ e ^ (s_let) : ty ]].
```

```
newframe(scopeOfTerm(l)) ⇒ F';
link(F', L(P(), F) ⇒ _;
Fs (F', F) ⊢ dec ⟶ _;
F' ⊢ e ⟶ v
------------------------------
F ⊢ l@Let(dec, e) ⟶ v
```

```
Dec[[ VarDec(x, t, e) ^ (s', s) ]]:=
  Tp[[ t ^ (s) : ty ]],
  Exp[[ e ^ (s) : ty ]],
  Var{x} ← s', Var{x} : ty.
```

```
F ⊢ e ⟶ v2;
set(F', x, v2) ⇒ _
----------------------------------------
Fs (F', F) ⊢ VarDec(x, _, e) ⟶ U()
```

```
Exp[[ Var(x) ^ (s) : ty ]] :=
  Var{x} → s, Var{x} ↦ d, d : ty.
```

```
F ⊢ Var(x) ⟶ get(lookup(F, x))
```

```
Dec[[ d@FunDec(f, x, t1, t2, e) ^ (s', s) ]]:=
  new s_fun, s_fun -P-> s',
  @d.scopeOf := s_fun,
  Tp[[ t1 ^ (s) : ty1 ]],
  Tp[[ t2 ^ (s) : ty2 ]],
  Var{x} ← s_fun, Var{x} : ty1,
  Exp[[ e ^ (s_fun) : ty2 ]],
  Var{f} ← s', Var{f} : FUN(ty1, ty2).
```

```
FunV(F, scopeOfTerm(d), arg, e) ⇒ clos;
set(F, f, clos) ⇒ _
--------------------------------------
Fs (F', F) ⊢ d@FunDec(f, arg, _, e) ⟶ U()
```

```
Exp[[ Call(f, e) ^ (s) : ty2 ]]:=
  Var{f} → s, Var{f} ↦ d, d : FUN(ty1, ty2),
  Exp[[ e ^ (s) : ty1 ]].
```

```
get(lookup(F, f)) ⇒
  FunV(Fp, s_fun, x, e_fun);
link(newframe(s_fun), L(P(), Fp)) ⇒ Fcall;
F ⊢ e ⟶ varg;
set(Fcall, x, varg) ⇒ _;
Fcall ⊢ e_fun ⟶ v
--------------------------------------------
F ⊢ Call(f, e) ⟶ v
```

```
Exp[[ Plus(e1, e2) ^ (s) : INT() ]]:=
  Exp[[ e1 ^ (s) : INT() ]],
  Exp[[ e2 ^ (s): INT() ]].
```

```
e1 ⟶ IntV(i1); e2 ⟶ IntV(i2)
------------------------------------
Plus(e1, e2) ⟶ IntV(plusI(i1, i2))
```

```
Exp[[ Minus(e1, e2) ^ (s) : INT() ]]:=
  Exp[[ e1 ^ (s) : INT() ]],
  Exp[[ e2 ^ (s): INT() ]].
```

```
e1 ⟶ IntV(i1); e2 ⟶ IntV(i2)
-----------------------------------
Minus(e1, e2) ⟶ IntV(subI(i1, i2))
```

Figure 8 Left: static semantics in NaBL2 for an adapted subset of Tiger. Right: corresponding dynamic semantics in DynSem using scopes and frames.

Note that the rule for `VarDec` analyzes the initializer expression using scope `s`, which is the outer scope of the corresponding `Let`. This entails that the variable declaration cannot be recursive (refer to itself). On the other hand, the rule for `FunDec` makes the scope `s'` in which the function is added as declaration, a parent scope `s_fun`, the scope of the body of the function. This entails that functions *can* be recursive.

```
sorts Val Frame Addr Occurrence

components
  F : Frame

sorts Link constructors
  L: Label * Frame→ Link

arrows
  newframe(Scope) ⟶ Frame
  link(Frame, Link) ⟶ Frame
  lookup(Frame, Occurrence) ⟶ Addr
  get(Addr) ⟶ Val
  get(Frame, Occurrence) ⟶ Val
  set(Addr, Val) ⟶ Val
  set(Frame, Occurrence, Val) ⟶ Val
```

Figure 9 DynSem API for frame operations.

4.3 DynSem with Scopes-and-Frames

Frame-based DynSem specifications rely on primitive frame operations provided as a language-independent library. Figure 9 declares the most important frame operations but elides their implementation. We discuss their semantics here; a reference dynamic semantics is given by Poulsen et al. [28].

The collection of linked frames is called the *heap*. The `newframe` operation instantiates a new frame in the heap given a `Scope`, which is a reference to a scope in the scope graph. This creates the required frame and frame slots for declarations and references but does not link the new frame. The `link` operation adds a link to a given frame. All links are labelled as in the scope graph. An `Occurrence` is a unique identification of the use of a name at a specific location in the program. Static name analysis transforms the program AST to replace each name occurrence, be it a declaration or a reference, with a unique identifier. Due to its uniqueness each occurrence is in precisely one scope. Given a reference occurrence and a frame, the `lookup` operation traverses the heap from the given frame to the frame holding a slot for the declaration occurrence by using the statically computed name resolution path. A lookup result is an `Address` specifically identifying a slot in a frame. Operations `get` and `set` read and update slots, respectively. Both operations come in a basic form operating on an address, and in a form directly operating on a frame and a slot.

Frame operations provide the building blocks for defining frame-based dynamic semantics specifications. The right column of Figure 8 shows the dynamic semantics in DynSem for the subset of Tiger discussed above. Each DynSem rule is listed next to the NaBL2 rule for the same construct. The binding in the DynSem rules follows the static semantics. Where the NaBL2 rule uses a scope, the DynSem rule uses a corresponding frame. Where the NaBL2 predicate is indexed by a scope (or scopes), the DynSem arrow is indexed by a corresponding frame (or frames). Thus, the language constructs are evaluated with the `Fs (Frame, Frame) ⊢Dec ⟶ Unit` and `F ⊢Exp ⟶ Val` arrows.

Where the NaBL2 rule creates a new scope, the DynSem rule creates a corresponding frame. There is some choice in the decision *when* to create a frame for a scope. For example, in the case of a `Let`, the frame is created as soon as the construct is evaluated. (Note that the scope is obtained from the NaBL2 `scopeOf` AST property, which is read using `scopeOfTerm` operator.) However, the evaluation rule for a function declaration does *not* create an instantiation of the scope of the function. Rather, a closure (`FunV`) is created that records the scope and the *parent* frame (F) of the function declaration. Only evaluation of the corresponding function *call* creates the function call frame and links it to the parent frame from the closure.

Where the NaBL2 rule declares a name, a DynSem rule assigns a value to the corresponding slot. For example, the VarDec rule assigns the value of the initializer expression to the slot for the variable in the binding frame. In the case of a function, the assignment of the value of the actual parameter is only done once the frame is created by the function call.

Where the NaBL2 rule resolves a name, the DynSem rule uses lookup to find the corresponding slot, using the path obtained from resolving the name in the scope graph. For example, the Var rule looks up the address of the slot for the variable and gets the value stored there. Similarly, the Call rule looks up the address of the function name and gets the closure stored there.

The systematic correspondence between static and dynamic name binding exhibited by the rules in Figure 8 extends to all name binding patterns covered by scope graphs. The Tiger language used for the evaluation of this paper has n-ary sequential let bindings, mutually recursive function declarations, type declarations, (recursive) record types, and arrays. The scope of a record describes the layout of its fields. A record instance is a frame derived from record's scope and holds field values. Record instantiation involves retrieving the scope of the record and creating a new frame from it.

4.4 Native Library for Scopes-and-Frames

A resolved scope graph is the result of static name and type analysis; once created, the graph and all the scopes it describes remain constant at run time. Thus, all frames created for a given scope will have the same structure, and the edges between frames follow the pattern fixed by scope graph edges. For example, a particular local variable reference in a program will always have the same name resolution path and will always identify the same slot in its declaration frame. This means that at run time we can partially evaluate a variable lookup to a number of frame link traversals and an offset in a declaration frame, similar to the way an optimizing compiler would optimize lookups statically.

The implementation strategy presented in this section is designed to allow the JIT compiler of the hosting VM (an Oracle Graal VM) to observe that frame structure is constant and to perform optimizations based on this observation. Our approach is to provide a Java implementation of the scopes and frames API of Figure 9, to be used in DynSem specifications. The library implements language-independent optimizations on frame operations which any language with a frame-based DynSem specification can benefit from, out of the box.

Object Storage Model. Our implementation choice is to model scopes and frames using the Truffle Object Storage Model (OSM) [37] and to implement scope and frame operations on this model. The OSM was designed as a generic framework for modeling memory in languages with dynamic name binding and typing. In particular the OSM provides a framework for modeling objects in memory that undergo shape changes, similar to objects in prototype-based languages such as Javascript. Truffle and Graal have special knowledge of the classes that make up the OSM and can perform optimizations on memory allocation and operations. Applying the OSM to a scope graph, which is by definition fixed at run time, is akin to applying it to its ideal corner case: all shapes of all objects are constant. It is however possible that the OSM introduces a certain amount of overhead that persists even in this ideal situation. As an alternative implementation strategy, one could map a scope to a Truffle

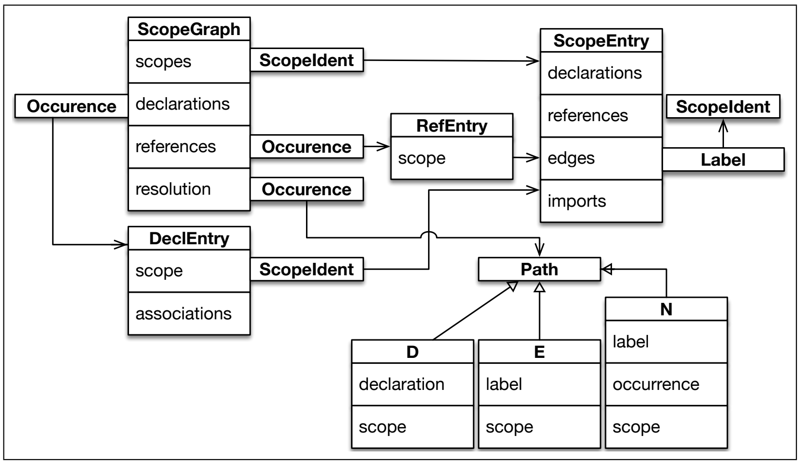


Figure 10 Components of a scope graph.

FrameDescriptor and a heap frame to a VirtualFrame. However, this mapping is intricate and would require all linked frames to be materialized in order to support frame linking. It is our understanding that materialized frames are slower than frames on the OSM.

We give a brief overview of the mapping of scopes and frames to the OSM. The OSM has three basic building blocks: objects, shapes and properties. A shape is a manifest of the properties of a family of objects and how they are laid out, akin to a prototype for an object or a class for an instance object. Shapes act as both descriptors for objects and factories for objects. A shape can be used to check whether a given object conforms to it, to retrieve properties of the object and to create new objects of that shape. A property uniquely identifies a slot and provides additional metadata to the JIT, such as whether the slot is mutable, nullable, and the type of values that it will store. The metadata informs the shape as to how the storage area for an object is to be constructed. Additionally, a property of a shape is the most efficient way to read or write the slot it identifies in an object of that shape. A property can therefore be seen as both a slot descriptor and a slot offset into an object.

Scope Graphs on OSM. Figure 10 shows the components in the makeup of a scope graph. We model them using the Truffle OSM. Declarations of layout interfaces inform the Truffle DSL to generate their implementations. A scope graph consists of scopes, declarations and references. A name resolution complements the scope graph with resolution paths from references to declarations. Paths start at the reference scope and end at the declaration scope. We use occurrences to uniquely identify declarations and references, and scope identifiers to uniquely identify scopes. Scope identifiers and occurrences are the keys to associative arrays maintained by the scope graph and are used to access detailed data. Note that we store scope graph data in a flattened representation; it is more efficient to look up scopes, declarations and references in flat associative maps than to search in graph-like structures. In the implementation, the associative arrays are instances of `DynamicObject` from the Truffle OSM. This allows Graal to optimize allocations and lookups, and gives us a set of tools for efficient access. `Occurrence` and `ScopeIdent` are optimized to have efficient hash code computation and fast equality checking.

At run time there exists precisely one scope graph. The meta-interpreter keeps a reference to the scope graph in a global interpreter context which is accessible to any interpreter node. This allows scope graph information to be accessed from anywhere in the meta-interpreter.

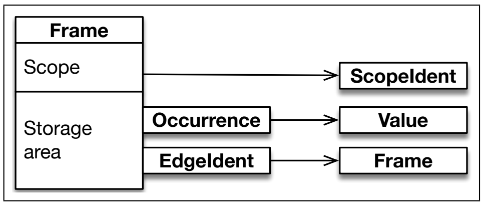


Figure 11 Structure of natively implemented frames.

Frames on OSM. We map frames and their respective operations onto the three core concepts of the OSM. Figure 11 describes the makeup of a frame. We implement a frame as an OSM object. A frame is made up of a scope uniquely identified by a `ScopeIdent` and an area for data storage. Each scope defines a unique frame shape. Each declaration is identified by its `Occurrence` and derives a frame slot property. Each edge of a scope is identified by an `EdgeIdent` – a pair of the edge label and the destination scope, and becomes a shape property and a slot in a frame. A shape dictates the structure of the storage area of a frame. Note that, by construction, all frames of a scope have the same shape. By checking whether any two frames have the same shape we effectively check whether they are frames of the same scope and vice versa.

Given a reference `Occurrence` and a starting frame, we look up the intended slot by traversing frame links as dictated by the name resolution path from the resolved scope graph. The result of the lookup is the address of the slot. The address is a pair of the frame and declaration `Occurrence` of the slot. The `Occurrence` identifies a slot property in the shape of the frame. This slot property can be used to efficiently access the slot in all frames of that shape. By definition, the relationship between a code fragment at a particular location and its surrounding scope is static. This means that code at that particular location will always execute in the context of frames derived from the same scope. This allows slot properties to be cached after their first lookup and later applied to access the slot efficiently, speeding up memory operations considerably. Such caching is particularly efficient because it can be left unguarded, since there is a static guarantee that the cached property will always be valid for that particular code location.

An advantage of mapping scopes and frames onto the Truffle OSM is that it allows the JIT compiler to observe memory operations. Since the JIT compiler can see through the memory of a running interpreter, we expect that the improvement will not be limited to just faster memory operations, but that the JIT will also optimize the running program by optimizing memory allocations. An additional advantage of using native frames is that garbage collection of frames is automatic and requires no effort from the language developer.

The native scopes and frames library makes the frame heap implicit and mutable, and does not allow it to be captured or reset. On the other hand, the vanilla DynSem library for scopes and frames uses explicit persistent data structures to model the heap. Although the heap is normally hidden from view (as an implicitly threaded semantic component), a language designer could intentionally define a semantics that observes it, captures or resets it. However, we have not encountered a language for which this would be a desirable implementation strategy. For example, even if a language needed transactional memory, capturing and resetting the entire heap would not be a good implementation approach; something finer-grained is needed. A more realistic approach would be to wrap the scopes and frames library to provide transaction support. This would work for both the vanilla DynSem and native scope and frames libraries.

5 Rule Inlining

The DynSem meta-interpreter [35] relies on Graal to optimize code within a rule and calls across rules. A rule call in the meta-interpreter corresponds to a function call in a regular interpreter. The JIT compiler will try to inline stable callees in order to reduce the number of dispatches and to generate larger compilation units. We observe that the vast majority of DynSem rules do not perform stable calls. The underlying cause is that most rules are intermediate rules, i.e. they adapt the input program term and call other rules to further reduce sub-terms. Consider, for example, the program of Figure 12a and the rule call tree of Figure 12b corresponding to its evaluation. With the exception of `FunDef`, `Var` and **`Int`**, all rules are intermediate. With the exception of meta-functions which are identified statically by their name, a callee rule is identified at runtime by the sub-term to be reduced, which in turn depends on the caller's input term. In other words a callee rule is looked up by what the JIT compiler sees as a runtime parameter to the caller. If it cannot determine that a caller's input term is constant, the JIT cannot decide to inline callees.

Not inlining of an intermediate callee rule leaves that rule exposed to calls from various callers on various program terms. We call a rule *polymorphic*, if throughout its invocations it reduces different terms. Conversely, a rule that always reduces the same term is *monomorphic*. For example, the `Call`, **`Int`** and `Var` rules of Figure 12b are polymorphic. (In this simple example, relatively many rules are monomorphic. In practice most rules in a specification are polymorphic, because the corresponding language constructs are used more than once in the program under evaluation.) Callees of polymorphic rules are not inlined, and not inlining increases the number of polymorphic rules. In larger programs, the net result is many small polymorphic rules which perform dynamic calls.

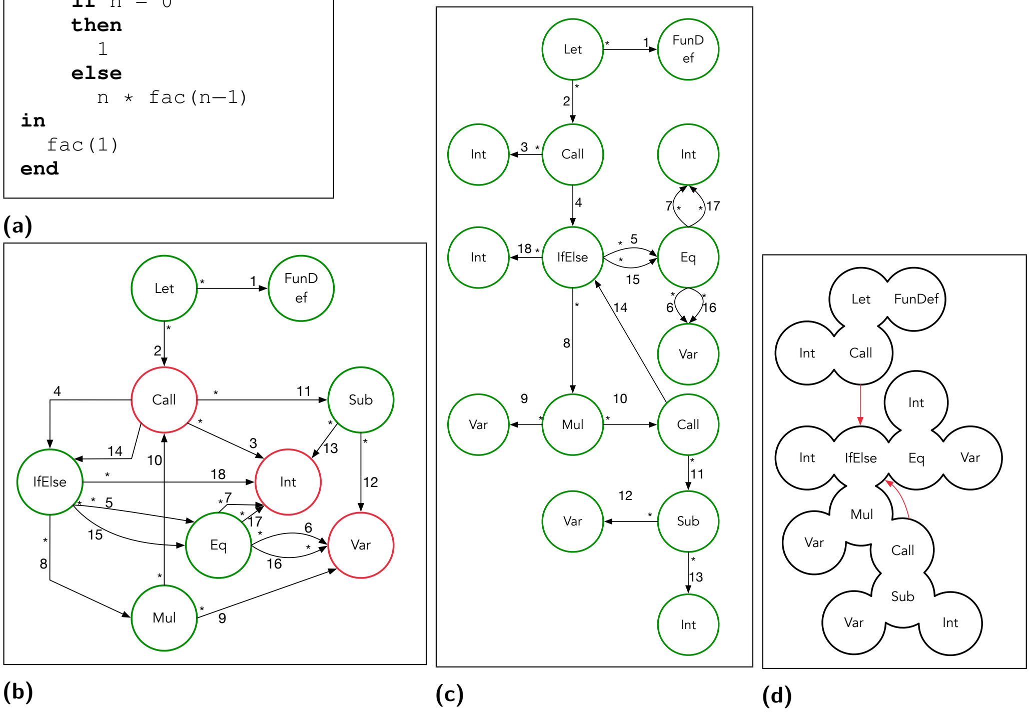
We distinguish two kinds of rule dispatch in a DynSem interpreter: *dynamic dispatch*, which depends on runtime values of the object program, and *structural dispatch*, which depends on the object program AST. In the call tree of Figure 12b all star-labeled arrows represent structural dispatch. It is desirable, and plausible, that all structural dispatch be eliminated by the JIT compiler; however, the issues outlined above prevent this. In this section we address this problem by presenting improvements to the DynSem interpreter that enable it to take explicit inlining decisions. In the ideal case the only remaining calls are those corresponding to dynamic dispatches, as illustrated in Figure 12d. The improvements consist of the following components:

- A rule-level source-to-source transformation on DynSem specifications that explicitly annotates structural rule dispatch.
- A load-time fusion of overloaded rules.
- A run-time rule-level signaling mechanism which allows any interpreter node to query whether its surrounding rule is monomorphic.
- A modified rule dispatch mechanism that can explicitly inline callee rules.

Binding-time Analysis. We introduce a lightweight source-to-source transformation of DynSem specifications that analyzes rules and identifies structural dispatches by marking meta-variables whose binding depends solely on the object program structure. Consider the arithmetic addition rule of Figure 13a where meta-variables `e1` and `e2` are annotated with `const`. The meaning of the `const` annotation on a meta-variable is twofold: (1) the meta-variable is known to stem from the rule's input without dependence on evaluation context or rule calls, and (2) the meta-variable will be bound to a term that will be constant if the surrounding rule is monomorphic. The `const` annotations of the meta-variables that are

```
let
  function fac(n) =
    if n = 0
    then
      1
    else
      n * fac(n-1)
in
  fac(1)
end
```

(a)



(b) **(c)** **(d)**

Figure 12 (a) Tiger program, (b) Rule call tree of program evaluation, (c) Rule call tree with cloned rules, (d) Rule call tree with rule inlining. Arrows marked with $*$ indicate calls on constant terms. Rules with green circles are monomorphic, those with red circles are polymorphic. Arrow numbers in figures (b) and (c) indicate execution order.

the inputs to the first two relation premises effectively mark the two rule calls as performing structural rule dispatch. It is the propagation of the `const` annotation to rule call premises that allows structural dispatch in Figure 12b to be identified and arrows labeled.

Consider the rule for a unary function call of Figure 13b. The meta-variable `e` bound to the parameter expression is `const` annotated. This identifies the evaluation of the parameter expression as requiring structural dispatch. At run time the evaluation of the parameter expression can be inlined if the surrounding rule is monomorphic. The function body `efun` retrieved from the closure is not `const` and its evaluation requires dynamic dispatch.

Fusion of Overloaded Rules. We call multiple DynSem rules that match the same pattern *overloaded* rules. Consider the six `eqV` rules of Figure 14a as an example of overloaded rules. The meta-interpreter loads overloaded rules as bundles. At rule call-time the rules in a bundle are executed one by one until the first applicable one is found and the call site caches the applicable rule at the call site. Subsequent executions of the call site first attempt the cached rules. In the event of a cache miss the remaining bundled rules are tried and the cache is grown with the newly applicable rule.

We observe that the success of a rule from the bundle is more likely to be determined by the state of the object program rather than by its structure. Consider for example a bundle of the two rules for an `if-then-else` statement. Indeed selecting one of the `if-then-else` rules depends on the result of evaluating its guard condition. By this reasoning we cannot estimate

```
const e1 ⟶ IntV(i1);
const e2 ⟶ IntV(i2);
IntV(addI(i1, i2)) ⇒ v
-------------------------------
Plus(const e1, const e2) ⟶ v
```

(a)

```
get(lookup(F, const f)) ⇒
   FunV(Fp, sfun, arg, efun);
link(newframe(sfun), L(P(), Fp)) ⇒ Fcall;
F ⊢ const e ⟶ varg;
set(Fcall, arg, varg) ⇒ _;
Fcall ⊢ efun ⟶ v
---------------------------------------------
F ⊢ Call(const f, const e) ⟶ v
```

(b)

Figure 13 DynSem rules for (a) arithmetic addition and (b) unary function call with annotated meta-variables after binding-time analysis.

the risk of a cache miss locally; and the price to pay for a cache miss is the decompilation of the caller rule. The risk of a cache miss increases further if the call is a dynamic dispatch or the caller is polymorphic.

We propose that a better strategy is to not force the caller to select a successful rule, and instead to let the callee choose the applicable rule. We do this by introducing a rule node that combines rules of a bundle into a single executable node, as shown in Figure 14b. At rule load-time, the meta-variable environments of the fused rules are concatenated and a `FusedRule` node is created for each rule bundle. The execution method of a `FusedRule` iterates through the rules, returning the result of the first applicable rule. Since the number of rules in a fused bundle is fixed at run time, the JIT compiler can completely unroll the iteration, and additional profiling can be performed on the actual number of iterations required. In addition to mitigating the risk of decompilation due to a callee cache miss, fusing rules drastically simplifies call-site logic. In the remainder of this section we refer to a rule obtained by fusion generically as a rule.

Signaling Monomorphic Rules. A structural dispatch call site (a call site which reduces a term assigned to a `const`-annotated meta-variable) must be able to query whether the surrounding caller is monomorphic or polymorphic and use this information to decide which call site optimizations can be performed. In the terms of Figure 12b, this means that a star-labelled outgoing arrow should be able to observe whether its source rule is green or red, i.e. monomorphic or polymorphic. To achieve this we install a flag at the root of each rule, as shown in the left panel of Figure 15. The flag is visible to all nodes within a rule, thus also to the nodes that implement variable reading semantics and call sites. A rule starts off as monomorphic and remains so as long as it is always invoked on the same program term. A rule becomes polymorphic, and its flag is invalidated, if and when it is invoked on a different program term. This is the case for the `Call` rule of Figure 15 which is invoked both from the body of the `let` construct, and from within the function body. We implement flag invalidation at the rule level, as shown in the left panel of Figure 15.

In the figure we describe the flag as a boolean, but in reality we implement the signal using a Truffle Assumption. Graal ensures that checking whether Assumptions are valid from JIT-ed code is very cheap, so using an assumption as a cache guard, or as a specialization guard is very efficient. While guard checking with assumptions is very cheap, the cost of decompilation and recompilation is still high.

Inlining Rules. In the call tree of Figure 12b, although dispatches to `Call`, `Int` and `Var` are all structural, the rules themselves are polymorphic because their different callees pass different input terms. However, we know that since a program is fixed, even a polymorphic

```
eqV(IntV(i), IntV(j)) ⟶ eqI(i, j)

eqV(StringV(x), StringV(y)) ⟶ eqS(x, y)

eqV(NilV(), NilV()) ⟶ 1

eqV(NilV(), RecordV(_)) ⟶ 0

eqV(RecordV(_), NilV()) ⟶ 0

eqV(RecordV(F1), RecordV(F2)) ⟶ eqFrames(F1, F2)
```

(a)

```
class FusedRule extends Rule {
  final Rule[] rules;

  FusedRule(Rule[] rules) {
    this.rules = rules;
  }
  Result execute(VirtualFrame frame) {
    for (int i = 0; i < rules.length; i++) {
      try {
        return rules[i].execute(frame);
      } catch (RuleFailure e) {}
    }
    throw new ReductionFailure("No more rules to try");
  }
}
```

(b)

Figure 14 (a) Overloaded equality rules. (b) Sketch implementation of the fused rule node.

rule has a finite set of behaviors. This set of behaviors is bound in the set of program terms that match the rule's pattern. We can create a specialized copy of the rule for each program term in this set, thereby reducing a polymorphic rule to a set of monomorphic rules. The specialized copies can be inlined to replace structural dispatches within other monomorphic rules. Applying rule cloning to the call tree of Figure 12b results in the call tree of Figure 12c; all rules in the tree are monomorphic. The dynamic dispatches that remain are those that reduce computed terms, i.e. the two closure applications (arrows 4 and 14).

We modify the meta-interpreter to inline (at run time) callees into their call site if two conditions are met: (1) the caller is monomorphic; and (2) the dispatch is structural. The right panel of Figure 15 sketches the inlining mechanism. At call time, if the conditions hold, the `uninitclone()` method copies the callee in an uninitialized state (i.e., in its state prior to any invocation), and the copy is adopted into the caller, becoming a child node. For subsequent calls, the inlined callee is executed directly as long as the rule stays monomorphic. The inlined callee is discarded and replaced by dynamic dispatch if the rule becomes polymorphic. Dynamic dispatch will attempt to cache callees locally to avoid repeated lookups; Figure 15 omits caching details for conciseness. Note that a callee is inlined without its root node, which allows calls to `getRootNode()` from within the callee to resolve to the root node of the caller. This has the advantage of sharing a single `monomorphic` flag for all inlined rules within a tree.

If we apply the cloning and inlining mechanism to the call tree of Figure 12b, the JIT will compile a monomorphic caller together with its inlined callees in a single compilation unit, thereby eliminating dispatches between rules altogether. This results in the call tree of Figure 12d where the red arrows correspond to the only two dynamic dispatches that remain. Inlining of structural dispatches creates rules which do more work locally and perform

```
class RuleRoot extends RootNode {
  boolean monomorphic = true;
  Rule rule;

  Result execute(VirtualFrame f) {
    return rule.execute(f);
  }
}

class Rule extends Node {
  Pattern patt;
  Premise[] premises;
  TermBuild output;
  Term tInit;

  Result execute(VirtualFrame f) {
    Term t = getInputTerm(f);
    patt.match(t);
    if (tInit == null) {
      tInit = t;
    } else if (getRootNode().monomorphic
                 && tInit != t) {
      getRootNode().monomorphic = false;
    }
    for (Premise p : premises) {
      p.execute(f);
    }
    return output.build(f);
  }
}
```

```
class Premise extends Node { ... }

class RelationPremise extends Premise{
  TermBuild input;
  Pattern output;
  Rule callee;

  void execute(VirtualFrame f) {
    Term t = input.build(f);
    Result res;
    if (getRootNode().monomorphic
        && input.isconst()) {
      if (callee == null) {
        callee = adopt(
          ruleRegistry().lookup(t)
            .rule.uninitclone()
        );
      }
      res = callee.execute(...);
    } else {
      callee = null;
      res = ruleRegistry().lookup(t)
               .execute(...);
    }
    output.match(res);
  }
}
```

Figure 15 Schematic implementation of rule calls with rule cloning.

fewer dynamic calls. In addition to reducing dynamic calls, this enables more intra-rule optimizations. Disadvantages of this method are longer compilation times due to larger compilation units and overhead during warmup due to rule cloning. Additionally, while larger compilation units enable better partial evaluation, this partial evaluation possibly takes longer, requiring more warmup rounds.

6 Evaluation

We evaluate our performance improvement techniques using DynSem specifications for Tiger, a simple programming language originally invented for teaching about compilers [2]. Tiger is a statically typed language with let bindings, functions, records and control-flow constructs. Our evaluation compares execution times across different flavors of Tiger implementations.

6.1 Experiment Set-up

Subjects. We evaluate four different implementations of Tiger: three meta-interpreted DynSem specifications and one hand-written Tiger interpreter. These are:

- `Meta-Env`: an environment-based DynSem specification interpreted on the runtime described in Section 3. This was the state-of-the-art DynSem runtime prior to the contributions of this paper.
- `Meta-SF`: a DynSem specification using Scopes & Frames as described in Section 4.3, interpreted on the runtime with native Scopes & Frames bindings of Section 4.4.
- `Meta-SF-Inline`: specification and runtime identical to `Meta-SF` with runtime rule inlining enabled.

- `Hand`: a Truffle-based AST interpreter using Scopes & Frames and implementing common Truffle optimization techniques (e.g. loop unrolling, polymorphic inline caches, branch profiles).

Workloads. We adapted the set of Tiger benchmark programs of Vergu et al. [35], which are translations of the Java programs of Marr et al. [19]. During earlier experimentation we discovered that benchmark runtime was too short on the faster meta-interpreters for a reliable time measurement. We addressed this by making the problems solved harder, resulting in the following six programs:

- `queens`: a solver for the 16-queens problem. The implementation uses let bindings, arrays, recursive function calls, for loops and nested conditional constructs.
- `list`: builds and traverses cons-nil lists. The program makes use of records, recursive function calls, while loops and conditionals.
- `towers`: a solver for the Towers of Hanoi game, primarily exercising records and recursive function calls.
- `sieve`: Sieve of Eratosthenes algorithm finding prime numbers smaller than 14,000. The program primarily exercises variable declarations, variable access in nested lexical scopes, and nested loops.
- `permute`: generates permutations of an array of size 8.
- `bubblesort`: performs bubble sort on a cons-nil list of 500 integers, initially in reverse order. The lists are built using records.

Methodology. We modified the four Tiger runtimes to repeat the evaluation of a program 200 times in the same process and to record the duration of each repetition. The time recorded is strictly program evaluation time, i.e. it excludes VM startup, program parsing, static analysis and interpreter instantiation. Each sequence of 200 in-process repetitions is repeated 30 times, as separate processes. We run the experiment on a Hewlett Packard ProLiant MicroServer Gen 8 with an Intel Xeon CPU E3-1265L V2 running at 2.5Ghz. The CPU has four cores; we disable one of the cores to ensure that heat dissipation is sufficient, and we disable hyper-threading to improve predictability. The machine has 16 GB of DDR3 memory, divided in two sockets, operating at a maximum frequency of 1.6Ghz, with ECC mode enabled. The operating system is a fresh minimal installation of Ubuntu Server `18.04.2` running a Linux kernel version `4.15.0-48`. All non-essential system daemons and networking are disabled before running the experiment, and we connect to the machine through out-of-band management facilities. All benchmark programs are run on the Oracle Graal Enterprise Edition VM version `1.0.0-rc9`.

We are interested in the steady state performance of each benchmark and VM combination. We use warmup_stats, part of the Krun [3] benchmarking system, to process and analyze the recorded timeseries. It performs statistical analyses to determine whether each combination of benchmark and VM shows stable performance and to compute this steady state performance.

6.2 Results

Table 1 shows the steady state runtimes, in seconds, for each configuration of benchmark and runtime. A missing measurement indicates that the configuration did not exhibit steady performance according to warmup_stats. We first consider the performance difference between traditional environment-based (`Meta-Env`) and scopes-and-frames (`Meta-SF`) specifications. For the remainder of this section, when we describe average speedup, we are referring to the geometric mean.

Table 1 Median steady state execution times, expressed in seconds, for combinations of benchmarks and VMs. The 99% confidence interval is shown in small font. Execution times for combinations which do not exhibit stable performance are excluded.

	Meta-Env	Meta-SF	Meta-SF-Inline	Hand
queens	1.7019 ±0.72583	0.0682 ±0.18626	0.0208 ±0.09366	0.0047 ±0.00085
list	0.2396 ±0.01789	0.0965 ±0.03700	0.0773 ±0.06191	
towers	9.5841 ±0.49535	0.6647 ±0.05259	0.0508 ±0.00460	0.0107 ±0.00030
sieve		0.0041 ±0.01925	0.0025 ±0.00196	0.0003 ±0.00053
permute	12.7514 ±1.91232	0.3216 ±0.02547	0.1108 ±0.00241	0.0260 ±0.00050
bubblesort	2.3551 ±0.34690	0.1164 ±0.01155	0.0147 ±0.00502	0.0060 ±0.02275

Table 2 Median number of repetitions required to reach steady state performance, and in small font the interquartile range. In parentheses (in normal font): the average duration, in seconds, of a warmup iteration.

	Meta-Env	Meta-SF	Meta-SF-Inline	Hand
queens	1	10.5 (1.0, 75.4) (1.78s)	51 (1.0, 77.1) (0.49s)	20 (18.5, 40.0) (0.25s)
list	98.5 (56.7, 121.5) (0.51s)	38.5 (25.0, 125.3) (0.49s)	81 (1.0, 106.5) (0.18s)	
towers	1	18 (18.0, 25.0) (2.49s)	89.5 (75.3, 119.5) (0.18s)	50.5 (42.4, 58.0) (0.09s)
sieve		106 (73.4, 146.6) (0.12s)	126 (5.5, 143.1) (0.04s)	9 (8.0, 17.6) (0.18s)
permute	1	68.5 (65.0, 84.5) (0.60s)	44 (40.0, 52.0) (0.28s)	30 (30.0, 43.5) (0.09s)
bubblesort	1	49 (31.4, 89.1) (1.42s)	67.5 (57.0, 85.5) (0.13s)	1

The Meta-SF interpreter improves on Meta-Env performance by an average 15x, with the highest gains for permute (39x) and smallest gains for list (2.5x). The runtimes on the two VMs are strongly correlated (correlation coefficient of 0.75), suggesting that adopting scopes and frames improves all benchmarks fairly uniformly. However, we also find a moderate correlation (correlation coefficient of 0.64) between the runtimes of Meta-Env and speedup gains exhibited by Meta-SF, suggesting that the longer the benchmark runtime on Meta-Env, the higher the speedup offered by Meta-SF. This may be due either to Meta-SF optimizing precisely the bottlenecks in Meta-Env, or simply to more complex programs benefiting more.

The Meta-SF-Inline VM improves on the performance of Meta-SF in 50% of the cases, while in the other 50% of the cases they are statistically indistinguishable. Meta-SF-Inline is always faster than Meta-Env by at least an order of magnitude and typically by two orders of magnitude, with the exception of queens for which it is at least 8.5x faster. There is strong correlation (0.79) between the runtime of benchmarks on Meta-SF and the speedup on Meta-SF-Inline. Coupled with only a moderate correlation (0.42) of runtimes on the two VMs, this suggests that, for the programs benchmarked, inlining addresses precisely the bottlenecks in Meta-SF. We do note the overlap in confidence intervals of runtime on Meta-SF and on Meta-SF-Inline for benchmarks queens, sieve and list which makes them statistically indistinguishable.

The handwritten interpreter Hand is on average 4.7x faster than Meta-SF-Inline, but not more than 30x faster. Some of these benchmarks have very short runtimes, but focusing on the two benchmarks with longest runtimes on Hand, permute and towers, produces a very similar overhead figure of 4.5x.

The number of iterations that are required until reaching steady state is an indication both of how JIT-able a benchmark/VM combination is and of how much particular optimizations compromise warmup time for maximum performance. Table 2 shows the median number of warmup iterations required until steady state is reached and the median duration of an iteration during warmup. With the exception of `list`, benchmarks on the environment-based VM do not seem to warm up well: they reach steady state performance in one iteration and never improve after that. It is noteworthy that `list`, the only benchmark that warms up on `Meta-Env`, is also the one least improved on by `Meta-SF`. In contrast to `Meta-Env`, the JIT compiler is able to optimize programs on the `Meta-SF` VM, but requires an average of 37 iterations to do so. We find a similar pattern for `Meta-SF-Inline`, typically requiring more warmup iterations than `Meta-SF` but resulting in faster code. We observe that even when the median warmup round on `Meta-SF-Inline` is slower than the steady-state performance on `Meta-SF`, it is within an order of magnitude slower, and that the average median warmup time on `Meta-SF-Inline` is shorter than on `Meta-SF`. From Table 1 we note that runtime confidence intervals are wider for the `Meta-SF` and `Meta-SF-Inline` VMs than they are for `Hand`; in particular for benchmark `queens` on `Meta-SF`, and for benchmarks `queens` and `list` on `Meta-SF-Inline`. The wide confidence intervals appear correlated with benchmark-VM combinations that have one or more non-warmup process executions (Table 2, combinations for which the 25th quantile is 1.0). This suggests some non-determinism over which we have little current understanding.

We find that replacing environments and stores by scopes and frames has a strictly beneficial effect on the execution time, and that meta-interpreters derived from scopes-and-frames specifications have better warm up characteristics. Adopting scopes and frames “out of the box” allows the JIT compiler to optimize the executing code. The JIT can see through memory operations and examine the memory layout of the program which enables partial evaluation of memory operations. Since our experiment does not measure the garbage collection activity, it is unclear to what degree the reported performance numbers are affected, positively or negatively, by garbage collection activity. We proposed in Section 5 that the fine granularity of code that the JIT is optimizing in the meta-interpreter case is a bottleneck in the optimizations that it can perform, and we introduced cloning and inlining of monomorphic rule calls at run time to attempt to improve on this situation. The expectation was that increasing the size of the rules, and thereby minimizing the number of calls across rules, would make the program easier to optimize. This expectation is borne out: in 50% of cases `Meta-SF-Inline` faster than `Meta-SF`, and in the other cases it is not slower. Inlining of rules increases the size of compilation units, aligns the structure of the rule call tree with the syntactic structure of the executing program, and the JIT can produce faster code.

Overall the combination of scopes and frames with inlining delivers a meta-interpreter that is always faster than using environments and stores. The speedup is at least one and typically two orders of magnitude. Moreover, the best meta-interpreter is within 10x slower (approximately 4.7x) than our optimized handwritten interpreter.

7 Discussion; Related and Future Work

The work presented in this paper is a performance improvement on the state of the art DynSem meta-interpreter. The improvement is achieved by (1) using scope and frames to model memory in dynamic semantics and (2) applying inline expansion of DynSem rules at run time.

Our work demonstrates a significant reduction in the execution time of meta-interpreted specifications of dynamic semantics using two techniques. The first exploits the systematic correspondence between static and run-time name binding exhibited by scopes and frames [28]. The second inlines reduction rules at run time to obtain coarser-grained rules that reflect the structure of the interpreted program. Combining these two techniques results in meta-interpreters that are at least one order of magnitude and generally two orders of magnitude faster than the state of the art DynSem meta-interpreter; and within a factor 5 from an optimized handwritten interpreter.

We remark that optimizations made to frame operations are in fact optimizations made to the executing program, not to the meta-interpreter. A resolved scope graph and paths in the scope graph representing the results of name resolution are program specific. Using the scope graph to inform optimizations of frame operations results in optimizations that are program specific. The JIT of the hosting VM, which hosts the meta-interpreter, is thus traversing the meta-interpreter layer to operate on the top-level interpreter. In the end, the program-specific optimizations performed by the JIT unlock further meta-interpreter optimizations than those limited to syntax-driven optimizations. Another indication that this is happening is, aside from the increased performance, the number of iterations required for code to warm up.

Related Work. DynSem [34], as a dynamic semantics framework, is part of the family of structural operational semantics (SOS) frameworks. This family contains big-step SOS (or natural semantics [16]); small-step SOS as originally introduced by Plotkin [27]; and reduction semantics with evaluation contexts (e.g. [11]), of which PLT Redex [10] is an instantiation. MSOS [22] and its extension I-MSOS [23] improve on the modularity and conciseness of traditional SOS by allowing *semantic components* such as environments and stores to be propagated implicitly through rules that do not modify those components. DynSem borrows the notion of implicit semantic propagation from I-MSOS and implements a systematic transformation of specifications with implicit components into equivalent specifications with explicit components. Typical DynSem specifications are in big-step style with implicit propagation of semantic components.

Dynamic semantics specifications take one of two approaches to specifying name binding: (a) eagerly substituting values for names or (b) propagating semantic components such as environments or stores that associate values with names. Specifications in Redex [18] and Ott [31] typically use substitution, while specifications in K [30] and funcons [7] typically use semantic components. Prior to the developments presented in this paper, DynSem specifications modeled name binding using semantic components that map identifiers to addresses and addresses to values and embedded name resolution semantics in terms of operations on these components. The DynSem extensions of Section 4 use scope graph [25] information to automatically derive a memory layout in terms of frames [28] and provide a set of primitives for operating on memory. The approach replaces environments, stores and other custom semantic components with a generic representation of memory stored in an implicitly propagated store. The only components passed in rules are frame references into the store.

Given a dynamic semantics for an object language there are three conceptual approaches to obtaining a execution engine for that language: (1) compile the semantics to an interpreter, (2) compile the semantics to a compiler or (3) interpret the semantics. DynSem, Redex [18] fall into the final category, i.e. a runtime is obtained by (meta-)interpreting a semantics. An older runtime for K [30] generated an interpreter for object language, but more recently K specifications can be directly interpreted. Significant amounts of research have gone into generating compilers from semantics [21, 26, 8] with varying degrees of applicability and usually with slow compilation or slow execution or both. For example, the SIS compiler generator of Mosses [21] compiled denotational semantics to a code generator, demonstrating that it was possible to compile code generators from declarative specifications. However, both the generated compiler and its emitted code were quite slow.

Translating a dynamic semantics specification to an efficient (and optimizing) compiler requires some form of offline partial evaluation [15]. The three approaches to make semantics specification executable are conceptually related to partial evaluation [15] and the Futamura projections [12, 13]. The first Futamura projection of a meta-interpreter and a semantics specification yields an interpreter, and the first Futamura projection of that interpreter and a program yields an executable. The second Futamura projection of a meta-interpreter and a semantics yields a compiler derived from the semantics. Amin et al. [1] describe the construction of a one-pass compiler that collapses all interpreter layers in a hierarchy-of-layers, thus eliminating the overhead of stacked interpretation.

Our approach to make DynSem specifications executable is through meta-interpretation with minimal pre-compilation. This raises the challenge of eliminating the overhead of meta-interpretation. The problem is more complicated than just optimizing an interpreter at runtime (as is done in just-in-time (JIT) compilation), because both the hosting and the hosted interpreters must be optimized simultaneously. The hosting meta-interpreter cannot effectively be partially evaluated without the hosted object interpreter, whose optimization in turn requires the program input.

There are two mainstream directions for implementing efficient interpreters, both relying on JIT compilation: meta-tracing and online partial evaluation. Meta-tracing, as provided by RPython [4] and applied to PyPy [5, 6] traces the execution of an interpreter to obtain a JIT compiler specific to that interpreter. The obtained JIT monitors the execution of the interpreter and compiles frequently executed code (of the interpreter) into highly efficient machine code. Only recently has online partial evaluation been shown as a practical meta-compilation technique of AST interpreters. Würthinger et al. [40] have developed Truffle, a framework for implementing interpreters. Truffle interpreters are AST interpreters, i.e. the control-flow of the interpreter follows the syntactic structure of the executing program. The Graal partial evaluator [39, 38] determines compilation units by resolving control-flow jumps across parts of the AST. For a practical comparison and evaluation of both meta-tracing and online partial evaluation of interpreters, we refer the reader to the research of Marr et al. [20].

To the best of our knowledge, neither meta-tracing nor online partial evaluation have been applied to two stacked layers of interpretation. Conceptually, meta-interpretation of a program with respect to a semantics specification involves a syntax-directed sequence of rule applications. A fixed program informs a fixed arrangement of rule applications, i.e. the rules of a specification are arranged such that they follow the AST of the program. This observation has motivated the choice of Truffle as an implementation target for the DynSem meta-interpreter. Conceptually, the Graal JIT has sufficient information to construct a tree of rules that strictly mimics the program AST. Construction of such a tree requires inlining of structural dispatch to rules, as discussed in Section 5. The inlining introduced in Section 5 is designed to aid the JIT in identifying control-flow jumps in the hosting meta-interpreter that are known to be stable but that the JIT cannot observe as such due to the intermediate interpreter layer.

Future Work. In the future we plan to investigate using Graal to perform optimizations with respect to program values. To some limited extent this is happening already: checks on value terms from within DynSem rules are observable by the JIT, and frame slot allocation takes into consideration the type of the declaration. There also still are opportunities for optimization with respect to rule inlining. Currently not all static bindings in rules are recognized as monomorphic. For example, while for a particular object language a function call is known to always resolve to a specific closure, the DynSem static analysis cannot currently determine this. While we can allow the language developer to explicitly annotate `const` meta-variables, we believe a better solution would be to uncover more static bindings automatically. We expect that combining a program, its scope graph, and a DynSem specification provides sufficient information to determine this. The scopes-and-frames approach may also apply to dynamic languages. We plan to investigate if by building frame structures dynamically and caching results of run-time name resolution we can obtain similar performance gains. Yet another research avenue is to explore whether using DynSem to define intrinsically-typed interpreters [29] for object languages provides further benefits for specialization.

References

1 Nada Amin and Tiark Rompf. Collapsing towers of interpreters. *Proceedings of the ACM on Programming Languages*, 2(POPL), 2018. `doi:10.1145/3158140`.

2 Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1998.

3 Edd Barrett, Carl Friedrich Bolz-Tereick, Rebecca Killick, Sarah Mount, and Laurence Tratt. Virtual machine warmup blows hot and cold. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA), 2017. `doi:10.1145/3133876`.

4 Carl Friedrich Bolz. *Meta-Tracing Just-in-Time Compilation for RPython*. PhD thesis, Heinrich Heine University Düsseldorf, 2014. URL: `http://d-nb.info/1057957054`.

5 Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. Tracing the meta-level: PyPy's tracing JIT compiler. In Ian Rogers, editor, *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems, ICOOOLPS 2009, Genova, Italy, July 6, 2009*, pages 18–25. ACM, 2009. `doi:10.1145/1565824.1565827`.

6 Carl Friedrich Bolz and Laurence Tratt. The impact of meta-tracing on VM design and implementation. *Science of Computer Programming*, 98:408–421, 2015. `doi:10.1016/j.scico.2013.02.001`.

7 Martin Churchill, Peter D. Mosses, Neil Sculthorpe, and Paolo Torrini. Reusable Components of Semantic Specifications. *Transactions on Aspect-Oriented Software Development*, 12:132–179, 2015. `doi:10.1007/978-3-662-46734-3_4`.

8 Olivier Danvy and René Vestergaard. Semantics-Based Compiling: A Case Study in Type-Directed Partial Evaluation. In Herbert Kuchen and S. Doaitse Swierstra, editors, *Programming Languages: Implementations, Logics, and Programs, 8th International Symposium, PLILP 96, Aachen, Germany, September 24-27, 1996, Proceedings*, volume 1140 of *Lecture Notes in Computer Science*, pages 182–197. Springer, 1996.

9 Sebastian Erdweg, Tijs van der Storm, Markus Völter, Laurence Tratt, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad A. Vergu, Eelco Visser, Kevin van der Vlist, Guido Wachsmuth, and Jimi van der Woning. Evaluating and comparing language workbenches: Existing results and benchmarks for the future. *Computer Languages, Systems & Structures*, 44:24–47, 2015. `doi:10.1016/j.cl.2015.08.007`.

10 Matthias Felleisen, Robby Findler, and Matthew Flatt. *Semantics Engineering with PLT Redex*. MIT Press, 2009.

11 Matthias Felleisen and Robert Hieb. The Revised Report on the Syntactic Theories of Sequential Control and State. *Theoretical Computer Science*, 103(2):235–271, 1992.

12 Yoshihiko Futamura. Partial Evaluation of Computation Process - An Approach to a Compiler-Compiler. *Higher-Order and Symbolic Computation*, 12(4):381–391, 1999. URL: `http://www.springerlink.com/content/l46w6q3720n57607/`.

13 Yoshihiko Futamura. Partial Evaluation of Computation Process, Revisited. *Higher-Order and Symbolic Computation*, 12(4):377–380, 1999.

14 Christian Humer, Christian Wimmer, Christian Wirth, Andreas Wöß, and Thomas Würthinger. A domain-specific language for building self-optimizing AST interpreters. In Ulrik Pagh Schultz and Matthew Flatt, editors, *Generative Programming: Concepts and Experiences, GPCE'14, Vasteras, Sweden, September 15-16, 2014*, pages 123–132. ACM, 2014. `doi:10.1145/2658761.2658776`.

15 Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial evaluation and automatic program generation*. Prentice Hall international series in computer science. Prentice Hall, 1993.

16 Gilles Kahn. Natural Semantics. In Franz-Josef Brandenburg, Guy Vidal-Naquet, and Martin Wirsing, editors, *STACS 87, 4th Annual Symposium on Theoretical Aspects of Computer Science, Passau, Germany, February 19-21, 1987, Proceedings*, volume 247 of *Lecture Notes in Computer Science*, pages 22–39. Springer, 1987.

17 Lennart C. L. Kats and Eelco Visser. The Spoofax language workbench: rules for declarative specification of languages and IDEs. In William R. Cook, Siobhán Clarke, and Martin C. Rinard, editors, *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010*, pages 444–463, Reno/Tahoe, Nevada, 2010. ACM. `doi:10.1145/1869459.1869497`.

18 Casey Klein, John Clements, Christos Dimoulas, Carl Eastlund, Matthias Felleisen, Matthew Flatt, Jay A. McCarthy, Jon Rafkind, Sam Tobin-Hochstadt, and Robby Findler. Run your research: on the effectiveness of lightweight mechanization. In John Field and Michael Hicks, editors, *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, pages 285–296. ACM, 2012. `doi:10.1145/2103656.2103691`.

19 Stefan Marr, Benoit Daloze, and Hanspeter Mössenböck. Cross-language compiler benchmarking: are we fast yet? In Roberto Ierusalimschy, editor, *Proceedings of the 12th Symposium on Dynamic Languages, DLS 2016, Amsterdam, The Netherlands, November 1, 2016*, pages 120–131. ACM, 2016. `doi:10.1145/2989225.2989232`.

20 Stefan Marr and Stéphane Ducasse. Tracing vs. partial evaluation: comparing meta-compilation approaches for self-optimizing interpreters. In Jonathan Aldrich and Patrick Eugster, editors, *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, pages 821–839. ACM, 2015. `doi:10.1145/2814270.2814275`.

21 Peter D. Mosses. Compiler Generation Using Denotational Semantics. In Antoni W. Mazurkiewicz, editor, *Mathematical Foundations of Computer Science 1976, 5th Symposium, Gdansk, Poland, September 6-10, 1976, Proceedings*, volume 45 of *Lecture Notes in Computer Science*, pages 436–441. Springer, 1976.

22 Peter D. Mosses. Modular structural operational semantics. *Journal of Logic and Algebraic Programming*, 60-61:195–228, 2004. `doi:10.1016/j.jlap.2004.03.008`.

23 Peter D. Mosses and Mark J. New. Implicit Propagation in Structural Operational Semantics. *Electronic Notes in Theoretical Computer Science*, 229(4):49–66, 2009. `doi:10.1016/j.entcs.2009.07.073`.

24 Pierre Neron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. A Theory of Name Resolution with extended Coverage and Proofs. Technical Report TUD-SERG-2015-001,

Software Engineering Research Group. Delft University of Technology, January 2015. Extended version of ESOP 2015 paper "A Theory of Name Resolution".

25 Pierre Néron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. A Theory of Name Resolution. In Jan Vitek, editor, *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9032 of *Lecture Notes in Computer Science*, pages 205–231. Springer, 2015. `doi:10.1007/978-3-662-46669-8_9`.

26 Lawrence C. Paulson. A Semantics-Directed Compiler Generator. In *POPL*, pages 224–233, 1982.

27 Gordon D. Plotkin. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming*, 60-61:17–139, 2004.

28 Casper Bach Poulsen, Pierre Néron, Andrew P. Tolmach, and Eelco Visser. Scopes Describe Frames: A Uniform Model for Memory Layout in Dynamic Semantics. In Shriram Krishnamurthi and Benjamin S. Lerner, editors, *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy*, volume 56 of *LIPIcs*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016. `doi:10.4230/LIPIcs.ECOOP.2016.20`.

29 Casper Bach Poulsen, Arjen Rouvoet, Andrew P. Tolmach, Robbert Krebbers, and Eelco Visser. Intrinsically-typed definitional interpreters for imperative languages. *Proceedings of the ACM on Programming Languages*, 2(POPL), 2018. `doi:10.1145/3158104`.

30 Grigore Rosu and Traian-Florin Serbanuta. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010. `doi:10.1016/j.jlap.2010.03.012`.

31 Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strnisa. Ott: Effective tool support for the working semanticist. *Journal of Functional Programming*, 20(1):71–122, 2010. `doi:10.1017/S0956796809990293`.

32 Hendrik van Antwerpen, Pierre Néron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. A constraint language for static semantic analysis based on scope graphs. In Martin Erwig and Tiark Rompf, editors, *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 49–60. ACM, 2016. `doi:10.1145/2847538.2847543`.

33 Hendrik van Antwerpen, Casper Bach Poulsen, Arjen Rouvoet, and Eelco Visser. Scopes as types. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA), 2018. `doi:10.1145/3276484`.

34 Vlad A. Vergu, Pierre Néron, and Eelco Visser. DynSem: A DSL for Dynamic Semantics Specification. In Maribel Fernández, editor, *26th International Conference on Rewriting Techniques and Applications, RTA 2015, June 29 to July 1, 2015, Warsaw, Poland*, volume 36 of *LIPIcs*, pages 365–378. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015. `doi:10.4230/LIPIcs.RTA.2015.365`.

35 Vlad A. Vergu and Eelco Visser. Specializing a meta-interpreter: JIT compilation of Dynsem specifications on the Graal VM. In Eli Tilevich and Hanspeter Mössenböck, editors, *Proceedings of the 15th International Conference on Managed Languages & Runtimes, ManLang 2018, Linz, Austria, September 12-14, 2018.* ACM, 2018. `doi:10.1145/3237009.3237018`.

36 Eelco Visser, Guido Wachsmuth, Andrew P. Tolmach, Pierre Néron, Vlad A. Vergu, Augusto Passalaqua, and Gabriël Konat. A Language Designer's Workbench: A One-Stop-Shop for Implementation and Verification of Language Designs. In Andrew P. Black, Shriram Krishnamurthi, Bernd Bruegge, and Joseph N. Ruskiewicz, editors, *Onward! 2014, Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software, part of SPLASH '14, Portland, OR, USA, October 20-24, 2014*, pages 95–111. ACM, 2014. `doi:10.1145/2661136.2661149`.

37 Andreas Wöß, Christian Wirth, Daniele Bonetta, Chris Seaton, Christian Humer, and Hanspeter Mössenböck. An object storage model for the truffle language implementation framework. In Joanna Kolodziej and Bruce R. Childers, editors, *2014 International Conference* *on Principles and Practices of Programming on the Java Platform Virtual Machines, Languages and Tools, PPPJ '14, Cracow, Poland, September 23-26, 2014*, pages 133–144. ACM, 2014. doi:10.1145/2647508.2647517.

38 Thomas Würthinger, Christian Wimmer, Christian Humer, Andreas Wöß, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon, and Matthias Grimmer. Practical partial evaluation for high-performance dynamic language runtimes. In Albert Cohen 0001 and Martin T. Vechev, editors, *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 662–676. ACM, 2017. doi:10.1145/3062341.3062381.

39 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 Antony L. Hosking, Patrick Th. Eugster, and Robert Hirschfeld, editors, *ACM Symposium on New Ideas in Programming and Reflections on Software, Onward! 2013, part of SPLASH '13, Indianapolis, IN, USA, October 26-31, 2013*, pages 187–204. ACM, 2013. doi:10.1145/2509578.2509581.

40 Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. Self-optimizing AST interpreters. In Alessandro Warth, editor, *Proceedings of the 8th Symposium on Dynamic Languages, DLS '12, Tucson, AZ, USA, October 22, 2012*, pages 73–82. ACM, 2012. doi:10.1145/2384577.2384587.

Transient Typechecks Are (Almost) Free

Richard Roberts
School of Engineering and Computer Science, Victoria University of Wellington, New Zealand
richard.andrew.roberts@gmail.com

Stefan Marr
School of Computing, University of Kent, UK
s.marr@kent.ac.uk

Michael Homer
School of Engineering and Computer Science, Victoria University of Wellington, New Zealand
mwh@ecs.vuw.ac.nz

James Noble
School of Engineering and Computer Science, Victoria University of Wellington, New Zealand
kjx@ecs.vuw.ac.nz

Abstract

Transient gradual typing imposes run-time type tests that typically cause a linear slowdown. This performance impact discourages the use of type annotations because adding types to a program makes the program slower. A virtual machine can employ standard just-in-time optimizations to reduce the overhead of transient checks to near zero. These optimizations can give gradually-typed languages performance comparable to state-of-the-art dynamic languages, so programmers can add types to their code without affecting their programs' performance.

2012 ACM Subject Classification Software and its engineering → Just-in-time compilers; Software and its engineering → Object oriented languages; Software and its engineering → Interpreters

Keywords and phrases dynamic type checking, gradual types, optional types, Grace, Moth, object-oriented programming

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2019.5

Funding This work is supported by the Royal Society of New Zealand Marsden Fund.

1 Introduction

> *"It is a truth universally acknowledged, that a dynamic language in possession of a good user base, must be in want of a type system."*
> with apologies to Jane Austen.

Dynamic languages are increasingly prominent in the software industry. Building on the pioneering work of Self [20], much work in academia and industry has gone into making them more efficient [13, 14, 66, 24, 23, 25]. Just-in-time compilers have, for example, turned JavaScript from a naïvely interpreted language barely suitable for browser scripting, into a highly efficient ecosystem, eagerly adopted by professional programmers for a wide range of tasks [44].

A key advantage of these dynamic languages is the flexibility offered by the lack of a static type system. From the perspective of many computer scientists, software engineers, and computational theologists, this flexibility has the disadvantage that programs without types are more difficult to read, to understand, and to analyze than programs with types. Gradual Typing aims to remedy this disadvantage, adding types to dynamic languages while maintaining their flexibility [16, 48, 50].

33rd European Conference on Object-Oriented Programming (ECOOP 2019).
Editor: Alastair F. Donaldson; Article No. 5; pp. 5:1–5:28
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

There is a spectrum of different approaches to gradual typing [22, 28]. At one end – "pluggable types" as in Strongtalk [17] or "erasure semantics" as in TypeScript [8] – all types are erased before the execution, limiting the benefit of types to the statically typed parts of programs, and preventing programs from depending on type checks at run time. In the middle, "transient" or "type-tag" checks as in Reticulated Python offer first-order semantics, checking whether an object's type constructor or supported methods match explicit type declarations [49, 11, 46, 60, 29]. Reticulated Python also supports an alternative "monotonic" semantics which mutates an object to narrow its concrete type when it is passed into a more specific type context. At the other end of the spectrum, behavioral typechecks as in Typed Racket [59, 57], Gradualtalk [3], and Reticulated Python's proxies, support higher-order semantics, retaining types until run time, performing the checks eagerly, and giving detailed information about type violations as soon as possible via blame tracking [63, 2]. Finally, Ductile typing dynamically interprets a static type system at runtime [7]. Unfortunately, any gradual system with run-time semantics (i.e. everything more complex than erasure) currently imposes a significant run-time performance overhead to provide those semantics [56, 62, 42, 6, 45, 55, 29, 30].

The performance cost of run-time checks is problematic in itself, but also creates perverse incentives. Rather than the ideal of gradually adding types in the process of hardening a developing program, the programmer is incentivized to leave the program untyped or even to *remove* existing types in search of speed. While the Gradual Guarantee [50] requires that removing a type annotation does not affect the result of the program, the performance profile can be drastically shifted by the overhead of ill-placed checks. For programs with crucial performance constraints, for new programmers, and for gradual language designers, juggling this overhead can lead to increased complexity, suboptimal software-engineering choices, and code that is harder to maintain, debug, and analyze.

In this paper, we focus on the centre of the gradual typing spectrum: the transient, first-order, type-tag checks as used in Reticulated Python and similar systems. Several studies have found that these type checks have a negative impact on programs' performance. Chung, Li, Nardelli and Vitek, for example, found that "*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).*" and say "*transient semantics…is a worst case scenario…, there is a cast at almost every call*" [22]. Greenman and Felleisen find that the slowdown is predictable, as transient checking "*imposes a run-time checking overhead that is directly proportional to the number of [type annotations] in the program*" [28], and Greenman and Migeed found a "*clear trend that adding type annotations adds performance overhead. The increase is typically linear.*" [29].

In contrast, we demonstrate that transient type checks can be "almost free". In our demonstration, we insert gradual checks naïvely for each gradual type annotation. Whenever an annotated method is called or returns, or an annotated variable is accessed, we check types dynamically, and terminate the program with a type error if the check fails. Despite this simplistic approach, a just-in-time compiler can eliminate redundant checks – removing almost all of the checking overhead – resulting in a performance profile aligned with untyped code.

We evaluate our approach by adding transient type checks to Moth, an implementation of the Grace programming language built on top of Truffle and the Graal just-in-time compiler [67, 66]. Inspired by Richards et al. [45] and Bauman et al. [6], our approach conflates types with information about the dynamic object structure (maps [20] or object shapes [65]), which allows the just-in-time compiler to reduce redundancy between checking structure and checking types; consequently, most of the overhead that results from type checking is eliminated.

The contributions of this paper are:

- demonstrating that VM optimizations enable transient gradual type checks with low performance cost
- an implementation approach that requires only small changes to existing abstract-syntax-tree interpreters
- an evaluation based on classic benchmarks and benchmarks from the literature on gradual typing

2 Gradual Types in Grace

This section introduces Grace, and motivates supporting transient gradual typing in the language.

2.1 The Grace Programming Language

Grace is an object-oriented, imperative, educational programming language, with a focus on introductory programming courses, but is also intended for more advanced study and research [9, 19]. While Grace's syntax draws from the so-called "curly bracket" tradition of C, Java, and JavaScript, the structure of the language is in many ways closer to Smalltalk: all computation is performed via dynamically dispatched "method requests" where the object receiving the request decides which code to run; returns within lambdas are "non-local", returning to the method activation in which the block is instantiated [27]. In other ways, Grace is closer to JavaScript than Smalltalk: Grace objects can be created from object literals, rather than by instantiating classes [10, 35] and objects and classes can be deeply nested within each other [37].

Critically, Grace's declarations and methods' arguments and results can be annotated with types, and those types can be checked either statically or dynamically. This means the type system is intrinsically gradual: type annotations should not affect the semantics of a correct program [50], and the type system includes a distinguished "`Unknown`" type which matches any other type and is the implicit type for untyped program parts.

The static core of Grace's type system is well described elsewhere [34]; here we explain how these types can be understood dynamically, from the Grace programmer's point of view. Grace's types are structural [9], that is, an object implements a type whenever it implements all the methods required by that type, rather than requiring classes or objects to declare types explicitly. Methods match when they have the same name and arity: argument and return types are ignored. A type thus expresses the requests an object can respond to, for example whether a particular accessor is available, rather than a nominal location in an explicit inheritance hierarchy.

Grace then checks the types of values at run time:

- the values of arguments are checked after a method is requested, but before the body of the method is executed;
- the value returned by a method is checked after its body is executed; and
- the values of variables are checked whenever written or read by user code.[1]

In the spectrum of gradual typing, these semantics are closest to the transient typechecks of Reticulated Python [60, 29]. Reticulated Python inserts transient checks only when a value flows from untyped to typed code, while Grace inserts transient checks only at explicit type annotations (but in principle checks every annotation every time).

[1] Our rational for checking reads in addition to writes is described in Section 6.2.

2.2 Why Gradual Typing?

Our primary motivation for this work is to provide a flexible system to check consistency between an execution of a program and its type annotations. A key part of the design philosophy of Grace is that the language should not force students to annotate programs with types until they are ready, so that teachers can choose whether to introduce types early, late, or even not at all.

A secondary goal is to have a design that can be implemented with only a small set of changes to facilitate integration in existing systems.

Both of these goals are shared with much of the other work on gradual type systems, but our context leads to some different choices. First, while checking Grace's type annotations statically may be optional, checking them dynamically should not be: any value that flows into a variable, argument, or result annotated with a type must conform to that type annotation. Second, adding type annotations should not degrade a program's performance, or rather, programmers should not be encouraged to improve performance by removing type annotations. And third, we allow the programmer to execute a program even when not statically type-correct. Allowing such execution is useful to students, where they can see concrete examples of dynamic type errors. This is possible because Grace's static type checking is optional, which means that an implementation cannot depend on the correctness or mutual compatibility of a program's type annotations.

Existing gradual type implementations do not meet these goals, particularly regarding performance; hence the ongoing debate about whether gradual typing is alive, dead, or some state in between [56, 62, 42, 6, 45, 29, 30].

2.3 Using Grace's Gradual Types

We now illustrate how the gradual type checks work in practice in the context of a simple program to record information about vehicles. Suppose the programmer starts developing this vehicle application by defining an object intended to represent a car (Listing 1, Line 1) and writes a method that, given the car object, prints out its registration number (Line 5).

```
1 def car = object {
2     var registration is public := "JO3553"
3 }
4
5 method printRegistration(v) {
6     print "Registration: {v.registration}"
7 }
```

Listing 1 The start of a simple Grace program for tracking vehicle information.

Next, the programmer adds a check to ensure any object passed to the `printRegistration` method will respond to the `registration` request; they define the structural type `Vehicle` [58] naming just that method (Listing 2, Line 1), and annotate the `printRegistration` method's argument with that type (Listing 2, Line 5). The annotation ensures that a type error will be thrown if an object, passed to the `printRegistration` method, cannot respond to the `registration` message. Without the type check, the `print` method would cause a run-time error when interpolating the string. However, since type errors cause termination, the run-time error in the middle of the `print` implementation will now be avoided.

```
type Vehicle = interface {
    registration
}

method printRegistration(v: Vehicle) {
    print "Registration: {v.registration}"
}
```

Listing 2 Adding a type annotation to a method parameter.

In Listing 3, the programmer continues development and creates two car objects (Lines 9 and 18), that conform to an expanded `Vehicle` type (Line 1).

```
 1 type Vehicle = interface {
 2     registration
 3     registerTo(_)
 4 }
 5
 6 type Person = interface { name }
 7 type Department = interface { code }
 8
 9 var personalCar : Vehicle :=
10   object {
11     var registration is public := "DLS018"
12     method registerTo(p: Person) {
13       ...
14       print "{self} is now registered to {p.name}"
15     }
16   }
17
18 var governmentCar : Vehicle :=
19   object {
20     var registration is public := "FKD218"
21     method registerTo(d: Department) {
22       print "{self} is now registered to {d.code}"
23     }
24   }
25
26 governmentCar.registerTo(
27   object {
28     var name is public := "Richard"
29   }
30 )
```

Listing 3 A program in development with inconsistently typed `registerTo` methods.

Note that each version of the `registerTo` method declares a different type for its parameter (Lines 12 and 21). When the programmer executes this program, both `personalCar` and `governmentCar` pass the type check for `Vehicle`. Both objects respond to `registeration` and `registerTo`. Notably, the type of the argument for `registerTo` is ignored. At Line 26 the developer attempts to register the government car to an object representing a person: only when the method (Line 21) is *invoked* will the gradual type test on the argument fail (the object passed in is not a `Department` because it lacks a `code` method).

3 Graal, Truffle, Self-Optimization and Dynamic Adaptive Compilation

This section gives a brief introduction into just-in-time compilation, and the main techniques we rely on for our optimizations.

3.1 Self-Optimizing Interpreters

Self-optimizing abstract-syntax-tree (AST) interpreters [68] are the foundation for the work presented here. Together with partial evaluation [66], self-optimization enables efficient dynamic language implementations that reach the performance of custom state-of-the-art virtual machines (cf. Section 5.2 and [41]). The framework for building such interpreters is called Truffle.

The key idea is that an AST rewrites itself based on a program's run-time values to reflect the minimal set of operations needed to execute the program correctly.

As an example, consider the addition of two numbers in a dynamic language, possibly written simply as: `a + b`. Because there are no static types known, the run-time values for `a` and `b` could potentially be anything from an integer or a double, to a string or a collection, or any arbitrary objects that have a "`+`" method. In an self-optimizing interpreter, the expression may be represented by an `add` node, with two child nodes that each read a variable. The first time the `add` node executes, it may find that both values to be added are integers. It will then speculate that all future executions also see integers, and thus, rewrite itself to an `add-integer` node. This `add-integer` node will simply confirm that both values are integers, and then directly perform the integer addition. Compared to a general `add` node, we do not have to cover the cases for doubles, strings, and other kinds of objects, which results in much simpler code that can be more easily optimized. All other cases are supported by rewriting the `add` node to more general versions. This happens, for instance, when the values are not integers. However, in practice, programs tend be monomorphic and so such speculation is highly beneficial.

What starts out as something close to a traditional AST, in the end, incorporates additional knowledge about execution. As a consequence of this rewriting, such trees should be referred to more correctly as *execution trees* rather than ASTs.

3.2 Polymorphic Inline Caches for Optimizing Dynamic Behavior

Polymorphic inline caches (PICs) [32] are a variation on the theme of caching run-time values to improve performance. Originally, they focused on method invocation in dynamic languages to avoid costly method lookups by caching the looked-up method for a specific type. For dynamic languages, PICs can be generalized to not only consider the receiver type, but instead the object shape (cf. Section 3.3), which enables the optimizations we are aiming for.

In a language such as JavaScript, a PIC could be used for the following expression: `obj.toString()`. The dot can be thought of as the lexical representation of the method lookup. An implementation would keep a small cache for each such dot in the code. This means, for each lexical lookup location, we have a separate cache. PICs benefit from the relatively monomorphic behavior of programs. A specific lexical lookup is likely to see only one kind of object in the `obj` variable, so the cache will usually have the correct method for the object ready and can avoid a costly lookup.

3.3 Object Shapes: Metadata for Dynamic Objects

Object shapes [65], which are also know as maps [20] or hidden classes, are in the most general case a usage profile for groups of objects. In languages such as Self, JavaScript, and Grace, we do not have classes that define the set of fields for an object. The set of fields might even change over time. Furthermore, fields can theoretically store any possible value. However, in practice, the behavior of programs is again relatively monomorphic and objects created in a specific part of a program are likely to have always the same set of fields, which each are used to store only a small number of different kinds of values. For example, an object representing a counter would have a field `count`, which always stores integers, while an object representing a person may have always a field `name` that stores a string, but never an integer.

Object shapes represent this run-time information in a way that allows a just-in-time compiler to represent objects in memory similarly to C structs, and then to generate highly efficient code. Object shapes can be conflated with additional information, for instance to represent knowledge about types [6, 45]. PICs identify groups of objects with the same structure based on the shape. Consequently, objects with the same shape use the same entry in the PIC. Similar to classes, shapes tend to be monomorphic in practice for a specific lexical location.

3.4 Just-in-Time Compilation with Graal and Truffle

The Graal compiler is a just-in-time compiler for Java. For languages built on the Truffle framework, Graal applies partial evaluation, which enables efficient native code generation for Truffle interpreters [66]. As such, Graal is a metacompiler. This means that instead of compiling a specific program, in our case a Grace program, Graal compiles our Grace interpreter Moth for the execution of a specific Grace method.

For simplicity, partial evaluation can be thought of a highly aggressive inlining strategy. It starts with the root node of a specific Grace method and inlines all interpreter code reachable from it. This is possible, because it speculates that the execution tree is constant.

To enable further optimizations, Graal also inlines on the level of the Grace code, i.e., across Grace methods. This is important as it exposes more opportunities to apply optimization. Consequently, Graal is able to optimize Grace code similar to how a custom Grace just-in-time compiler would work, and it applies, e.g., constant folding, common subexpression elimination, and loop-invariant code motion.

Loop-invariant code motion and common subexpression elimination are especially important because Moth relies on self-optimizing nodes, PICs, and object shapes. These techniques introduce various optimistic checks, i.e., guards. To generate efficient native code, a compiler must move such checks out of loops and remove redundant checks altogether.

By combining all the techniques sketched in this section, Graal and Truffle are able to execute dynamic languages as efficiently as virtual machines built for a specific language – but with much less implementation effort.

4 Moth: Grace on Graal and Truffle

Implementing dynamic languages as state-of-the-art virtual machines can require enormous engineering efforts. Meta-compilation approaches [41] such as RPython [12, 14] and GraalVM [67, 66] reduce the necessary work dramatically, because they allow language implementers to leverage existing VMs and their support for just-in-time compilation and garbage collection.

Moth [47] adapts SOMns [38] to leverage this infrastructure for Grace. SOMns is a Newspeak implementation [18] on top of the Truffle framework and the Graal just-in-time compiler, which are part of the GraalVM project. One key optimization of SOMns for this work is the use of object shapes [65], also known as maps [20] or hidden classes. They represent the structure of an object and the types of its fields. In SOMns, shapes correspond to the class of an object and augment it with run-time type information. With Moth's implementation, SOMns was changed to parse Grace code, adapting a few of the self-optimizing abstract-syntax-tree nodes to conform to Grace's semantics. Despite these changes Moth preserves the peak performance of SOMns, which reaches that of Google's V8 JavaScript implementation (cf. Section 5.2 and Marr et al. [40]).

4.1 Adding Gradual Type Checking

One of the goals for our approach to gradual typing was to keep the necessary changes to an existing implementation small, while enabling optimization in highly efficient language runtimes. In an AST interpreter, we can implement this approach by attaching the checks to the relevant AST nodes: the expected types for the argument and return values can be included with the node for requesting a method, and the expected type for a variable can be attached to the nodes for reading from and writing to that variable. In practice, we encapsulate the logic of the check within a new class of AST nodes that are specially design to support gradual type checking. Moth's front end was adapted to parse and record type annotations and attach instances of this checking node as children of the existing method, variable read, and variable write nodes.

The check node uses the internal representation of a Grace type (cf. Listing 4, Line 13) to test whether an observed object conforms to that type. An object satisfies a type if all members required by the type are provided by that object (Line 5). *Note that here we use a pseudo code syntax similar to Python for all code examples that represent the implementation of Moth, even though Moth is implemented in Java. We chose this syntax to avoid any confusion with our Grace examples.*

```
 1 class Type:
 2   def init(members):
 3     self._members = members
 4
 5   def is_satisfied_by(other: Type):
 6     for m in self._members:
 7       if m not in other._members:
 8         return False
 9     return True
10
11   def check(obj: Object):
12     t = obj.get_type()
13     return self.is_satisfied_by(t)
```

Listing 4 Sketch of a `Type` in our system and its `check()` semantics.

```
 1 global record: Matrix
 2 
 3 class TypeCheckNode(Node):
 4 
 5   expected: Type
 6 
 7   @Spec(static_guard=`expected.check(obj)`)
 8   def check(obj: Number):
 9     pass
10 
11   @Spec(static_guard=`expected.check(obj)`)
12   def check(obj: String):
13     pass
14 
15   ...
16 
17   @Spec(guard=`obj.shape==cached_shape`, static_guard=`expected.check(obj)`)
18   def check(obj: Object, @Cached(obj.shape) cached_shape: Shape):
19     pass
20 
21   @Fallback
22   def check(obj: Any):
23     T = obj.get_type()
24 
25     if record[T, expected] is unknown:
26       record[T, expected] = T.is_subtype_of(expected)
27 
28     if not record[T, expected]:
29       raise TypeError(f"{obj} doesn't implement {expected}")
```

Listing 5 A sketch of the specializations in `TypeCheckNode` to minimize the run-time overhead of type checking. A specialization is a minimal set of operations for one specific situation, e.g., that the value to be checked is some type of number.

4.2 Optimization

There are two aspects to our implementation that are critical for a minimal-overhead solution:

- specialized executions of the type checking node, along with guards to protect these specialized versions, and
- a matrix to cache sub-typing relationships to eliminate redundant exhaustive subtype tests.

Optimized Type Check Node. The first performance-critical aspect to our implementation is the optimization of the type checking node. We rely on Truffle and its TruffleDSL [31]. This means we provide a number of special cases, which are selected during execution based on the observed concrete kinds of objects. A sketch of our type checking node using a pseudo-code version of the DSL is given in Listing 5. A simple optimization is for well known types such as numbers (Line 8) or strings (Line 12). The methods annotated with `@Spec` (shorthand for `@Specialization`) correspond to possible states in a state machine that is generated by the TruffleDSL. Thus, if a check node observes a number or a string, it will check on the first execution only that the expected type, i.e., the one defined by some type annotation, is satisfied by the object using a `static_guard`. If this is the case, the DSL will activate this state. For just-in-time compilation, only the activated states and their normal guards are considered. A `static_guard` is not included in the optimized code. If a check fails, or no specialization matches, a fallback (i.e., `check_generic` in Line 22) will be selected. This fallback will raise a type error when appropiate.

```
 1 class VariableReadNode(Node):
 2   slot: FrameSlot
 3   type_check: TypeCheckNode
 4
 5   @Spec
 6   def do_read(frame: VirtualFrame):
 7     value = frame.read(slot)
 8     if type_check:
 9       type_check.check(value)
10     return value
```

Listing 6 Sketch of a `VariableReadNode` using the `TypeCheckNode` to ensure Grace's transient semantics.

For generic objects we rely on the specialization of Line 18, which checks that the object satisfies the expected type. If that is the case, it reads the shape of the object (cf. Section 4) at specialization time and caches it for later comparisons. Thus, during normal execution, we only need to read the shape of the object and then compare it to the cached shape with a simple reference comparison. If the shapes are the same, we can assume the type check passed successfully. Note that shapes are not equivalent to types, however, shapes imply the set of members of an object, and thus, do imply whether an object fulfills one of our structural types.

The `TypeCheckNode` is used in Moth in all places that need to check types, which includes reading and writing variables as well as method requests and returns. Listing 6 shows a sketch of an AST node that implements reading from a local variable, which is stored in a frame object. A frame corresponds to a stack frame, sometimes also called an environment.

Line 8 first checks whether a type check needs to be performed. Since type annotations are optional, it may not be necessary to check for a type. Note that `type_check` is a constant for just-in-time compilation (cf. Section 3.4), which enables subsequent optimizations. Line 9 then calls the `check()` method on the `TypeCheckNode`, which may result in a type error. For a variable that only contains numbers, the `type_check` object would activate the number specialization in its state machine. For just-in-time compilation, this means only the code for checking numbers needs to be compiled, but none of the other specializations.

In many cases, the specialization for objects would be activated in a `TypeCheckNode`, which checks the shape of an object against a cache. This check is identical to the check performed by a polymorphic inline cache (PIC, cf. Section 3.2). Since PICs are used for all method calls, they are very common in most programs, and these additional checks can often be removed easily via common subexpression elimination.

Subtype Cache Matrix. The other performance-critical aspect to our implementation is the use of a matrix to cache sub-typing relationships. The matrix compares types against types, featuring all known types along the columns and the same types again along the rows. A cell in the table corresponds to a sub-typing relationship: does the type corresponding to the row implement the type corresponding to the column? All cells in the matrix begin as unknown and, as encountered in checks during execution, we populate the table. If a particular relationship has been computed before we can skip the check and instead recall the previously-computed value (Line 26 in Listing 5). Using this table we are able to eliminate the redundancy of evaluating the same type to type relationships across different checks in the program. To reduce redundancy further we also unify types in a similar way to Java's string interning; during the construction of a type we first check to see if the same set of members is expressed by a previously-created type and, if so, we avoid creating the new instance and provide the existing one instead.

Together the self-specializing type check node and the cache matrix ensure that our implementation eliminates redundancy, and consequently, we are able to minimize the run-time overhead of our system.

5 Evaluation

To evaluate our approach to gradual type checking, we first establish the baseline performance of Moth compared to Java and JavaScript and then assess the impact of the type checks themselves.

5.1 Method and Setup

To account for the complex warmup behavior of modern systems [4] as well as the non-determinism caused by e.g. garbage collection and cache effects, we run each benchmark for 1000 iterations in the same VM invocation.[2] To improve the confidence in the results further, we run all experiments with 30 invocations. Afterwards, we inspected the run-time plots over the iterations and manually determined a cutoff of 350 iterations for warmup, i.e., we discard iterations with signs of compilation. All reported averages use the geometric mean since they aggregate ratios.

All experiments were executed on a machine running Ubuntu Linux 16.04.4, with Kernel 3.13. The machine has two Intel Xeon E5-2620 v3 2.40GHz, with 6 cores each, for a total of 24 hyperthreads. We used ReBench 0.10.1 [39], Java 1.8.0_171, Graal 0.33 (`a13b888`), Node.js 10.4, and Higgs from 9 May 2018 (`aa95240`). Benchmarks were executed one by one to avoid interference between them. The analysis of the results was done with R 3.4.1, and plots are generated with ggplot 2.2.1 and tikzDevice 0.11. Our experimental setup is available online to enable reproductions.[3]

5.2 Are We Fast Yet?

To establish the performance of Moth, we compare it to Java and JavaScript. Moth is used in its untyped version, i.e., without type checks. For JavaScript we chose two implementations, Node.js with V8 as well as the Higgs VM. The Higgs VM is an interesting point of comparison, because Richards *et al.* [45] used it in their study. The goal of this comparison is to determine whether our approach could be applicable to industrial strength language implementations without adverse effects on their performance.

We compare across languages based on the Are We Fast Yet benchmarks [40], which are designed to enable a comparison of the effectiveness of compilers across different languages. To this end, they use only a common set of core language elements. While this reduces the performance-relevant differences between languages, the set of core language elements covers only common object-oriented language features with first-class functions. Consequently, these benchmarks are not necessarily a predictor for application performance, but can give a good indication for basic mechanisms such as type checking.

[2] For the Higgs VM, we only use 100 iterations, because of its lower performance. This is sufficient since Higgs's compilation approach induces less variation and leads to more stable measurements.

[3] `https://github.com/gracelang/moth-benchmarks/releases/tag/papers/ecoop19`

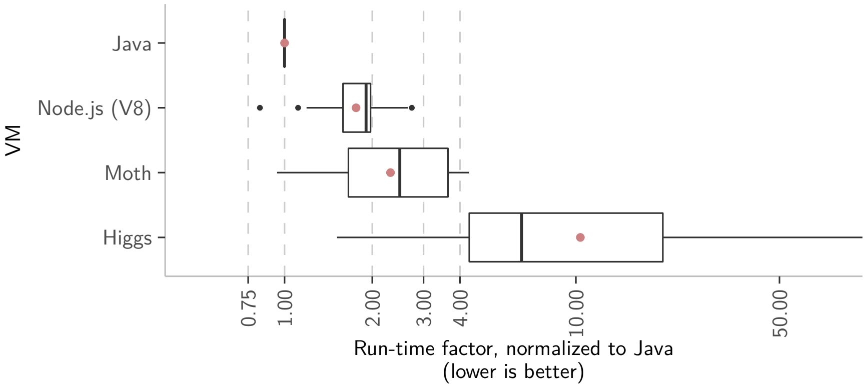


Figure 1 Comparison of Java 1.8, Node.js 10.4, Higgs VM, and Moth. The boxplot depicts the peak-performance results for the Are We Fast Yet benchmarks, each benchmark normalized individually based on the result for Java, which means all results for Java are 1.0, and its box appears as a line. The dots on the plot represent the geometric mean reported as averages. For these benchmarks, Moth is within the performance range of JavaScript, as implemented by Node.js, which makes Moth an acceptable platform for our experiments.

Figure 1 shows the results. We use Java as baseline since it is the fastest language implementation in this experiment. Note that we perform a unit conversion on the results separately for each benchmark, using the average of Java as 1 unit. While this conversion does not change the distribution of the data, it allows us to show it neatly on one plot.

We see that Node.js (V8) is about 1.8x (min. 0.8x, max. 2.7x) slower than Java. Moth is about 2.3x (min. 0.9x, max. 4.3x) slower than Java. As such, it is on average 31% (min. −16%, max. 2.3x) slower than Node.js. Compared to the Higgs VM, which is on these benchmarks 10.4x (min. 1.5x, max. 163x) slower than Java, Moth reaches the performance of Node.js more closely. With these results, we argue that Moth is a suitable platform to assess the impact of our approach to gradual type checking, because its performance is close enough to state-of-the-art VMs, and run-time overhead is not hidden by slow baseline performance.

5.3 Performance of Transient Gradual Type Checks

The performance overhead of our transient gradual type checking system is assessed based on the Are We Fast Yet benchmarks as well as benchmarks from the gradual-typing literature. The goal was to complement our benchmarks with additional ones that are used for similar experiments and can be ported to Grace. To this end, we surveyed a number of papers [56, 62, 42, 6, 45, 55, 29] and selected benchmarks that have been used by multiple papers. Some of these benchmarks overlapped with the Are We Fast Yet suite, or were available in different versions. While not always behaviorally equivalent, we chose the Are We Fast Yet versions since we already used them to establish the performance baseline. The selected benchmarks as well as the papers in which they were used are shown in Table 1.

The benchmarks were modified to have complete type information. To ensure correctness and completeness of these experiments, we added an additional check to Moth that reports absent type information to ensure each benchmark is fully typed. To assess the performance overhead of type checking, we compare the execution of Moth with all checks disabled, i.e., the baseline version from Section 5.2, against an execution that has all checks enabled. We did not measure programs that mix typed and untyped code because with our implementation technique a fully typed program is expected to have the largest overhead.

Table 1 Benchmarks selected from literature.

Fannkuch	[62, 29]	
Float	[62, 42, 29]	
Go	[62, 42, 29]	
NBody	[36, 62, 29]	used [40]
Queens	[62, 42, 29]	used [40]
PyStone	[62, 42, 29]	
Sieve	[56, 42, 6, 45, 30]	used [40]
Snake	[56, 42, 6, 45, 30]	
SpectralNorm	[62, 42, 29]	

Peak Performance

Figure 2 depicts the overall results comparing Moth, with all optimizations, against the untyped version. The run-time overhead, after discarding the warmup iterations, is on average 5% (min. −13%, max. 79%).

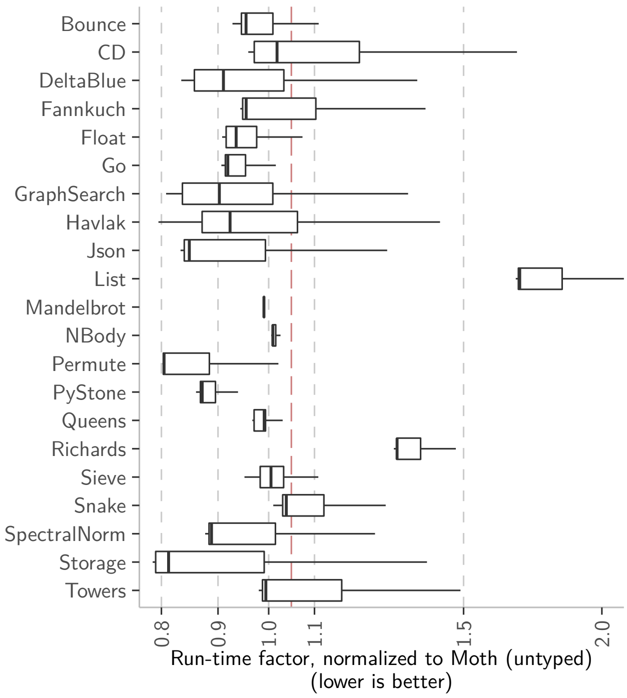


Figure 2 A boxplot comparing the performance of Moth with and without type checking. The plot depicts the run-time overhead on peak performance over the untyped performance. On average, transient type checking introduces an overhead of 5% (min. −13%, max. 79%). The average is indicated as a line with long dashes. Note that the axis is logarithmic to avoid distorting the proportions of relative speedups and slowdowns.

The benchmark with the highest overhead of 79% is List. The benchmark traverses a linked list and has to check the list elements individually. Unfortunately, the structure of this list introduces checks that do not coincide with shape checks on the relevant objects. We consider this benchmark a pathological case and discuss it in detail in Section 6.1.

Beside List, the highest overheads are on Richards (33%), CD (12%), Snake (14%), and Towers (12%). Richards has one major component, also a linked list traversal, similar to List. Snake and Towers primarily access arrays in a way that introduces checks that do not coincide with behavior in the unchecked version.

In some benchmarks, however, the run time decreased; notably Permute (−13%), GraphSearch (−3%), and Storage (−8%). Permute simply creates the permutations of an array. GraphSearch implements a page rank algorithm and thus is primarily graph traversal. Storage stresses the garbage collector by constructing a tree of arrays. For these benchmarks the introduced checks seem to coincide with shape-check operations already performed in the untyped version. The performance improvement is possibly caused by having checks earlier, which enables the compiler to more aggressively move them out of loops. Another reason could simply be that the extra checks shift the boundaries of compilation units. In such cases, checks might not be eliminated completely, but the shifted boundary between compilation units might mean that the generated native code interacts better with the instruction cache of the processor.

Warmup Performance

To more precisely measure warmup, all experiments were executed 30 times. The resulting Figure 3 shows the first 100 iterations for each benchmark. For each iteration n, we normalized the measurements to the mean of iteration n of the untyped Moth implementation. Thus, any increase indicates a slow down because of typing. The darker lines indicate the means, while the lighter area indicates a 95% confidence interval.

Looking only at the first few iterations, where we assume that most code is executed in the interpreter and might be affected by compilation activity, the overhead appears minimal. Only the Mandelbrot and CD benchmarks shows a noticeable slowdown.

Mandelbrot with its distinctly slow first iteration can be explained by its code structure. Since it is a computational kernel with many primitive operations, but no method calls, optimized code is only reached after the first full benchmark iteration. The problem could be alleviated with on-stack-replacement for loops, which is currently not done. Since other benchmarks use methods, they reach compiled code earlier and do not exhibit the same first-iteration slowdown.

PyStone however show various spikes. Since spikes appear in both directions (speedups and slowdowns), we assume that they indicate a shift, for instance, of garbage collection pauses, which may happen because of different heap configurations triggered by the additional data structures for type information.

5.4 Effectiveness of Optimizations

To characterize the concrete impact of our two optimizations – i.e., the optimized type checking node that replaces complex type tests with checks for object shapes and our matrix to cache sub-typing information, – we look at the number of type checks performed by the benchmarks as well as the impact on peak performance.

Impact on Performed Type Tests

Table 2 gives an overview of the number of type tests done by the benchmarks during execution. We distinguish two operations `check_generic` and `is_subtype_of`, which correspond to the operations in Line 22 and Line 5 of Listing 4. Thus, `check_generic` is the test called whenever a full type check has to be performed, and `is_subtype_of` is the part of the check

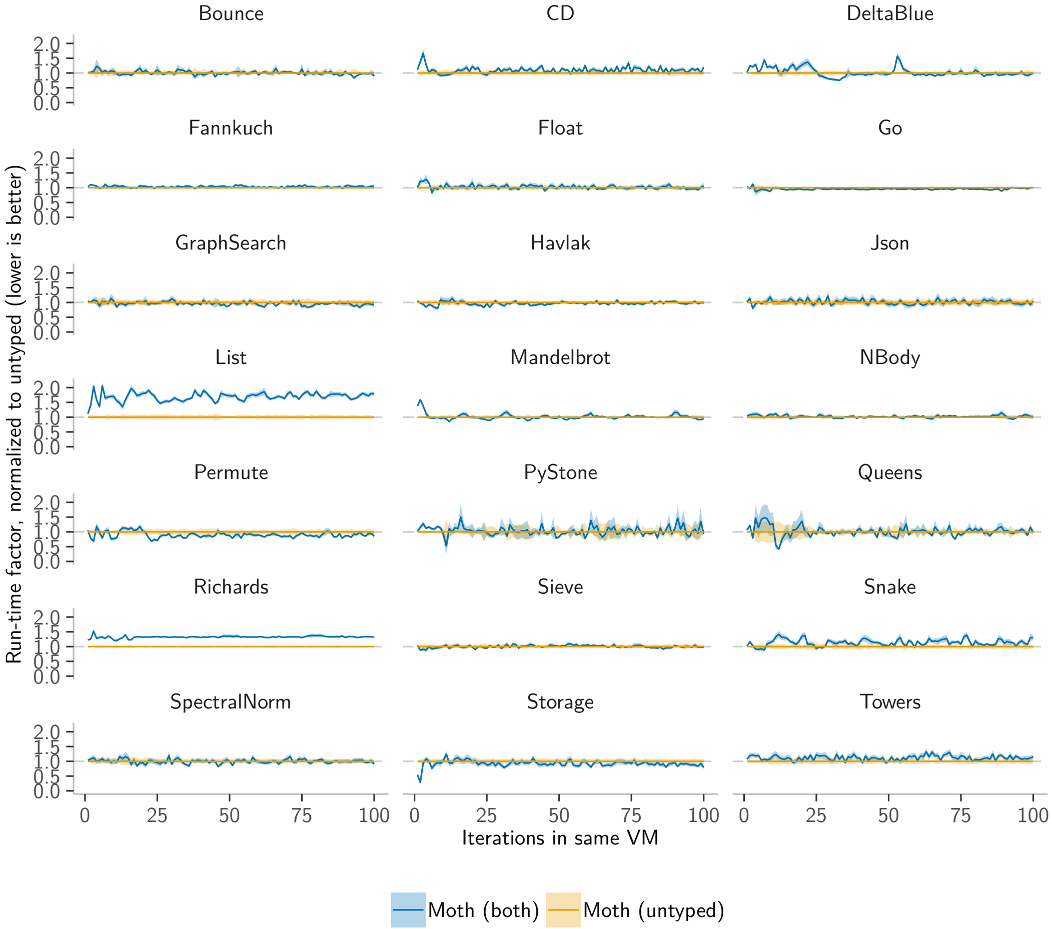


Figure 3 Plot of the run time for the first 100 iterations. The lines indicate the mean at iteration n normalized to the untyped result, the lighter area indicates a 95% confidence interval. The first iteration, i.e., mostly interpreted, seems to be affected significantly only for Mandelbrot, though CD shows slower behavior in early warmup, too.

that determines the relationship between two types. The second column of Table 2 indicates which optimization is applied, and the following columns show the mean, minimum, and maximum number of invocations of the tests over all benchmarks.

The baselines without optimizations are the rows with the results for neither of the optimizations being enabled. Depending on the benchmark, we see that the type tests are done tens of millions to hundreds of millions times for a single iteration of a benchmark.

Our optimizations reduce the number of type test invocations dramatically. As a result, the full check for the subtyping relationship is done only once for a specific type and super type. Similarly, the generic type check is replaced by a shape check and thus reduces the number of expensive type checks to the number of lexical locations that verify types combined with the number of shapes a specific lexical location sees at run time.

Impact on Performance

Figure 4 shows how our optimizations contribute to the peak performance. The figure depicts Moth's peak performance over all benchmarks, depending on the activated optimizations. As for Figure 1, we do a per-benchmark unit conversion using Moth (untyped), preserving the

Table 2 Type Test Statistics over all Benchmarks. This table shows how many of the type tests can be avoided based on our two optimizations. As indicated by the numbers, the number of type tests can vary significantly between benchmarks. Thus, the table shows the mean, minimum, and maximum number of type tests across all benchmarks for a given configuration. With the use of an optimized node that replaces type checks with simple object shape checks, `check_generic` is invoked only for the first time that a lexical location sees a specific object shape, which eliminates run-time type checks almost completely. Using our subtype matrix that caches type-check results, invocations of `is_subtype_of` are further reduced by an order of magnitude.

Type Test	Enabled Optimization	mean #invocations	min	max
check_generic	Neither	137,525,845	11,628,068	896,604,537
	Subtype Cache	137,525,845	11,628,068	896,604,537
	Optimized Node	292	68	1,012
	Both	292	68	1,012
is_subtype_of	Neither	134,125,215	11,628,067	896,604,534
	Subtype Cache	16	10	29
	Optimized Node	292	68	1,012
	Both	16	10	29

distribution properties of the results, but enabling us to show the results on a single plot.

As seen before in Figure 2, the untyped version is faster by 5%. Moth with both optimizations enabled as well as Moth with the optimized type-check node (cf. Listing 4) reach the same performance. This indicates that the subtype cache matrix is not strictly necessary for the peak performance. However, we can see that the subtype cache matrix improves performance by an order of magnitude over the Moth version without any type check optimizations. This shows that it is a relevant and useful optimization. Based on the numbers of Table 2, we see that this optimization is relevant for the very first execution of code. For code that has not executed before, having the global cache for the subtype relations gives the most benefit. After the first execution, the lexical caches in form of the type check nodes are primed with the same information, and the subtype cache matrix is only rarely needed. An example for code that benefits from the subtype cache matrix is unit test code, because most of the code is executed only once. While the performance of unit tests is not always critical, it can have a major impact on developer productivity.

Impact on Memory Usage

In our implementation, the subtype cache matrix is the largest additional data structure. We initialize it for up to 1000 types and use 1 byte per type combination. Java utilizes ca. 1MB of memory for the matrix. Additional memory is used to represent the type-check nodes at every lexical location. Since they behave like polymorphic inline caches (PIC) [32], their memory usage depends on the specific program execution. For the benchmarks used in this paper, the extra memory use can be up to 200KB.

In the context of Graal and Truffle, this additional memory usage is small, since the metacompilation approach uses a lot of memory [41]. In a dedicated virtual machine, memory use can be further optimized and be as efficient as for other kinds of PICs.

5.5 Transient Typechecks are (Almost) Free

As discussed in the introduction, in many existing gradually typed systems, one would expect a linear increase of the performance overhead with respect to an increasing number of type annotations.

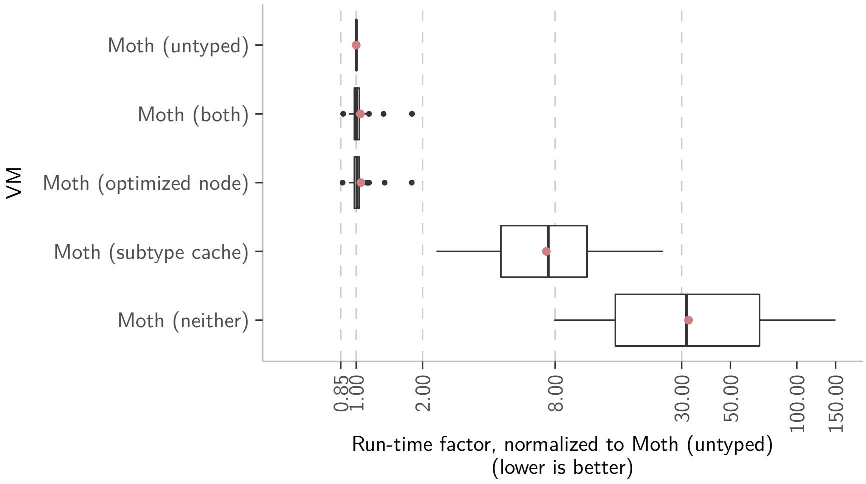


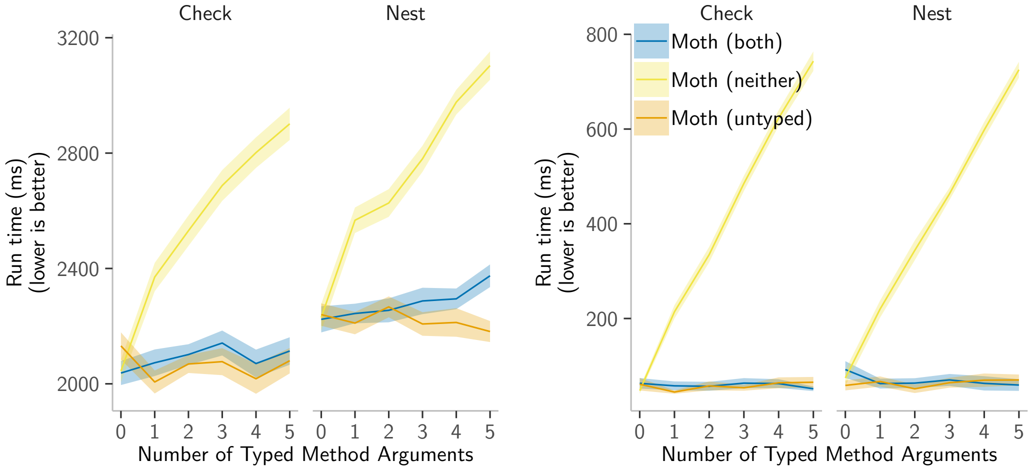
Figure 4 Performance Impact of the Optimizations on the Peak Performance over all benchmarks. The boxplot shows the performance of Moth normalized to the untyped version, i.e., without any type checks. This means all results for Moth (untyped) are 1.0 and its box appears as a line. The dots on the plot represent the geometric mean reported as averages. The performance of Moth with both optimizations and Moth with only the node for optimized type checks are identical. The subtype check cache improves performance over the unoptimized version, but does not contribute to the peak performance.

In this section, we show that this is not necessarily the case on our system. For this purpose we use two microbenchmarks, Check and Nest, which have at their core method calls with 5 parameters. The Check benchmark calls the same method 10 times in a row, i.e., it has 10 call sites. The Nest benchmark has 10 methods with identical signatures, which recurse from the first one to the last one. Thus, there are still 10 method calls, but they are nested in each other. In both benchmarks, each method increments a counter, which is checked at the end of the execution to verify that both do the same number of method activations, and only the shape of the activation stack differs.

Each benchmark exists in six variants, each variant in a separate file, going from having no type annotations over annotating only the first method parameter to annotating all 5 parameters. To demonstrate the impact of compilation, we present the results for the first iteration as well as the hundredth iteration. The first iteration is executed at least partially in the interpreter, while the hundredth iteration executes fully compiled.

Figure 5 shows that such a common scenario of methods being gradually annotated with types does not incur an overhead on peak performance in our system. The plot shows the mean of the run time for each benchmark configuration. Furthermore, it indicates a band with the 95% confidence interval. The yellow line, Moth (neither), corresponds to our Moth with type checking but without any optimizations. For this case, we see that the performance overhead grows linearly with the number of type annotations.

For Moth (both) and Moth (untyped), we see for the first iteration that the band of confidence intervals diverges, indicating that the additional type checks have an impact on startup performance. In contrast the confidence intervals overlap for the hundredth iteration, which shows that Moth does not suffer from a general linear overhead when adding type checks. Instead, most type checks do not have an impact on peak performance. However, as previously argued for the List benchmark, this is only the case for checks that can be subsumed by shape checks (shape checks are performed whether or not type checks are present).



(a) Iteration 1.

(b) Iteration 100.

Figure 5 Transient Typechecks are (Almost) Free. Two microbenchmarks, each with six variants, demonstrate the common scenario of adding type annotations over time, which in our system does not have an impact on peak performance. The benchmark variants differ only in the increasing number of method arguments that have type annotations. We show the result for the first benchmark iteration (a) and the one hundredth (b). Moth (neither), i.e., Moth without our two optimizations sees a linear increase in run time. For the first iteration, we see some difference between Moth (both) and Moth (untyped). By the hundredth iteration, however, the compiler has eliminated the overhead of the type checks and both Moth variants essentially have the same performance (independent of the number of method arguments with type annotations).

5.6 Changes to Moth

Outlined earlier in Section 4, a secondary goal of our design was to enable the implementation of our approach to be realized with few changes to the underlying interpreter. This helps to ensure that each Grace implementation can provide type checking in a uniform way.

By examining the history of changes maintained by our version control, we estimate that our implementation of Moth required 549 new lines and 59 changes to existing lines. The changes correspond to the implementation of new modules for the type class (179 lines) and the self-specializing type checking node (139 lines), modifications to the front end to extract typing information (115 new lines, 14 lines changes) and finally the new fields and amended constructors for AST nodes (116 new lines, 45 lines changes).

6 Discussion

6.1 The VM Could Not Already Know That

One of the key optimizations for our work and the work of others [6, 45] is the use of object shapes to encode information about types (in our case), or type casts and assumptions (in the case of gradually typed systems). The general idea is that a VM will already use object shapes for method dispatches, field accesses, and other operations on objects. Thus any further use to also imply type information can often be optimized away when the compiler sees that the same checks are done, and therefore can be combined by optimizations such as common subexpression elimination.

```
type ListElement = interface {
  next
}

var elem: ListElement := headOfList
while (...) do {
  elem := elem.next
}
```

Listing 7 Example for dynamic type checks not corresponding to existing checks.

This assumption breaks, however, when checks are introduced that do not correspond to those that exist already. As described in Section 4, our approach introduces checks for reading from and writing to variables. Listing 7 gives an example of a pathological case. It is a loop traversing a linked list. For this example our approach introduces a check, for the `ListElement` type, when (1) assigning to and reading from `elem` and (2) when activating the `next` method. The checks for reading from `elem` and activating the method can be combined with the dispatch's check on object shape. Unfortunately, the compiler cannot remove the check when writing to `elem`, because it has no information about what value will be returned from `next`, and so it needs to preserve the check to be able to trigger an error on the assignment. For our List benchmark, this check induces an overhead of 79%.

Compiler optimizations such as inlining are also insufficient for this particular case, because there are no guarantees about what `elem` does to implement `next`. The `next` method of a specific kind of `ListElement` may even have a type annotation for a return value. The best Graal can do in this example is to combine the check for the return value with the one writing to `elem`.

Since the example shows a somewhat generic data structure, there is the question of whether the issue applies to other data structures as well. Our benchmarks use a range of data structures including hash maps, sets, and vectors, none of which show the issue, because in more complex programs the chance of already having a check there is high, and cases were there has not been one before seem to be rare – although one can always consider additional optimizations to eliminate further checks. For generic data structures, storage strategies [13] could be used to encode type information about elements. This would allow the VM to check only once before a loop, and the loop could then rely on that check for guarantees about the elements of the data structure.

6.2 Optimizations

Read and Write Checks. As a simplification, we currently check variable access on both reads and writes. This approach simplifies the implementation, because we do not need to adapt all built-ins, i.e., all primitive operations provided by the interpreter. One optimization could be to avoid read checks. A type violation can normally only occur when writing to a variable, but not when reading. However, to maintain the semantics, this would require us to adapt many primitives. Examples for primitives are operations that activate blocks, which need to check their arguments or return values as well as any primitives that write to variables or fields. Given the number of primitives, this is error prone and incompleteness would result in missing type checks.

By checking reads and writes in a few well defined locations, we get errors as soon as user code accesses fields and variables. Moreover, only a small set of locations required changes to Moth's code, which reduces implementation overhead. Given the good results (cf. Sections 5.4 and 5.6), we decided to keep read checks, because it is a more uniform and maintainable approach for an academic project.

Dynamic Type Propagation. Another optimization could be to use Truffle's approach to self-specialization [68] and propagate type information to avoid redundant checks. At the moment, Truffle interpreters typically use self-specialization to specialize the AST to avoid boxing of primitive types. This is done by speculating that some subtree always returns the expected type. If this is not the case, the return value of the subtree is going to be propagated via an exception, which is caught and triggers respecialization. This idea could possibly be used to encode higher-level type information for return values, too. This could be used to remove redundant checks in the interpreter by simply discovering at run time that whole subexpressions conform to the type annotations.

Performance Impact of Types. As seen in Section 6.1, there are cases where adding types may reduce performance, even so, in the best case this does not happen (cf. Section 5.5).

While the expectation is that adding more types may result in higher potential for performance issues, in the context of dynamic and adaptive compilation as used for Moth, this is not necessarily the case. Since compilers rely on various heuristics, for instance for inlining, there may be situations where a fully typed program is faster than a program with fewer types. Since the checks need to be compiled themselves, they also influence such heuristics. This means it is possible that partially typed programs may show worse performance than fully typed ones.

6.3 Threats to Validity

This work is subject to many of the threats to validity common to evaluations of experimental language implementations. Our underlying implementation may contain undetected bugs that affect the semantics or performance of the gradual typing checks, affecting construct validity – we may not have implemented what we think we have. Given that our benchmarking harness runs on the same implementation, it is also subject to the same risks and thus affecting internal validity – we may not be measuring the implementation correctly. Moth is built on the Truffle and Graal toolchain, so we expect external validity there at least – we expect the results would transfer to other Graal VMs doing similar AST-based optimizations. We have less external validity regarding other kinds of VMs (such as VMs specialized to particular languages, or VMs built via meta-tracing rather than partial evaluation). Nevertheless, we expect our results should be transferable as we rely on common techniques.

Generalizability. Finally, because we are working in Grace, it is less obvious that our results generalize to other gradually typed-languages. We have worked to ensure that e.g. our benchmarks do not depend on any features of Grace that are not common in other gradually-typed object-oriented languages, but as Grace lacks a large corpus of programs the benchmarks are necessarily artificial, and it is not clear how the results would transfer to the kinds of programs actually written in practice. The advantage of Grace (and Moth) for this research is that their relative simplicity means we have been able to build an implementation that features competitive performance with significantly less effort than would be required for larger and more complex languages. On the other hand, more effort on optimisations could lead to even better performance.

Another aspect which limits generalizability is the specific semantics of Grace. Reticulated Python, Typed Racket, and Gradualtalk have semantics that need additional runtime support, and thus, we cannot draw any conclusions without further research.

For languages such as Newspeak, Strongtalk, or TypeScript, where types do not have run-time semantics, one could add termination based on type errors to these languages, or simply avoid termination and report the errors after program completion as a debugging aid. For either option, our approach should apply and we would expect similar results.

7 Related Work

Although syntaxes for type annotations in dynamic languages go back at least as far as Lisp [54], the first attempts at adding a comprehensive static type system to a dynamically typed language involved Smalltalk [33], with the first practical system being Bracha's Strongtalk [17]. Strongtalk (independently replicated for Ruby [26]) provided a powerful and flexible static type system, where crucially, the system was *optional* (also known as pluggable [16]). Programmers could run the static checker over their Smalltalk code (or not); either way the type annotations had no impact whatsoever of the semantics of the underlying Smalltalk program.

Siek and Taha [48] introduced the term "gradual typing" to describe the logical extension of this scheme: a dynamic language with type annotations that could, if necessary, be checked at runtime. Siek and Taha build on earlier complementary work extending fully statically typed languages with a "`DYNAMIC`" type – Abadi et al. 's 1991 TOPLAS paper [1] is an important early attempt and also surveys previous work.

Revived practical adoption of dynamic languages generated revived research interest, leading to the formulation of the *gradual guarantee* to characterize sound gradual type systems: informally "removing type annotations always produces a program that is still well typed" and also "evaluates to an equivalent value" [50], drawing on Boyland's critical insight that such a guarantee must by its nature exclude code that reflects on the presence or absence of type declarations [15]. Moth ensures that the values passing through type annotations cannot be incompatible with those annotations and that type annotations cannot change program values; notably, the type tests consider only method names and not any further type annotations. This means that removing type annotations cannot cause a program to fail or change its behaviour, satisfying the informal statement of the gradual guarantee. Moth does not meet the refined formal statement of the guarantee in Sieket al.'s [50]'s Theorem 5, however, because Theorem 5 requires all intermediate values conform to their inferred static types. Moth only checks at explicit type declarations, not inferred intermediate types.

Type errors in gradual, or other dynamically checked, type systems will be detected at the type declarations, but often those declarations will not be at fault – indeed in a correctly typed program in a sound gradually typed system, the declarations cannot be at fault because they will have passed the static type checker. Rather, the underlying fault must be somewhere within the barbarian dynamically typed code *trans vallum*. Blame tracking [63, 52, 2] localizes these faults by identifying the point in the program where the system makes an assumption about dynamically typed objects, so it can identify the root cause should the assumption fail. Different semantics for blame detect these faults slightly differently and incur differing implementation overheads [60, 51, 62].

The diversity of semantics and language designs incorporating gradual typing has been captured recently via surveys incorporating formal models of different design options. Chung et al. [22] present an object-oriented model covering optional semantics (erasure), transient semantics, concrete semantics (from Thorn [11]), and behavioural semantics (from Typed Racket), and give a series of programs to clarify the semantics of a particular language. Greenman et al. take a more functional approach, again modelling erasure, transient ("first order"), and behavioural ("higher order") semantics [28], and also present performance information based on Typed Racket. Wilson et al. take a rather different approach, employing questionnaires to investigate the semantics programmers expect of a gradual typing system [64].

As with languages more generally, there seem to be two main implementation strategies for languages mixing dynamic and static type checks: either adding static checks into a dynamic language implementation, or adding support for dynamic types to an implementation that depends on static types for efficiency. Typed Racket, for example, optimizes code with a combination of type inference and type declarations – the Racket IDE "optimizer coach" goes as far as to suggest to programmers type annotations that may improve their program's performance [53]. In these implementations, values flowing from dynamically to statically typed code must be checked at the boundary. Fully statically typed code needs no dynamic type checks, and so generally performs better than dynamically typed code. Adopting a gradual type system such as Typed Racket [59] allows programmers to explicitly declare types that can be checked statically, removing unnecessary overhead. Ortin et al.'s [43] approach takes this to a logical extreme using a rule base to guide program specialisation at compile time based on abstract interpretation.

On the other hand, systems such as Reticulated Python [60], SafeTypeScript [45], and our work here take the opposite approach. These systems do not use information from type declarations to optimize execution speed. Rather, the necessity to perform potentially repeated dynamic type checks tends to slow programs down; instead, here, code with no type annotations generally performs better than statically typed code or code with many type annotations. In the limit, these kinds of systems may only ever check types dynamically and may not involve a static type checker at all.

As gradual typing systems have come to wider attention, the question of their implementation overheads has become more prominent. Takikawa et al. [56] asked "is sound gradual typing dead?" based on a systematic performance measurement on Typed Racket. The key here is their evaluation method, where they constructed a number of different permutations of typed and untyped code, and evaluated performance along the spectrum [30]. Bauman et al. [6] replied to Takikawa et al.'s study, in which they used Pycket [5] (a tracing JIT for Racket) rather than the standard Racket VM, but maintained full gradually-typed Racket semantics. Bauman et al. are able to demonstrate most benchmarks with a slowdown of 2x on average over all configurations. Note that this is not directly comparable to our system, since typed modules do not need to do any checks at run time. Typed Racket only needs to perform checks at boundaries between typed and untyped modules, however, they use the same essential optimization technique that we apply, using object shapes to encode information about gradual types. Muehlboeck and Tate [42] also replied to Takikawa et al., using a similar benchmarking method applied to Nom, a language with features designed to make gradual types easier to optimize, demonstrating speedups as more type information is added to programs. Their approach enables such type-driven optimizations, but relies on a static analysis which can utilize the type information, and the underlying types are nominal, rather than structural.

Most recently, Kuhlenschmidt et al. [36] employ an ahead of time (i.e. traditional, static) compiler for a custom language called Grift and demonstrate good performance for code where more than half of the program is annotated with types, and reasonable performance for code without type annotations.

Perhaps the closest to our approach are Vitousek et al. [60] (incl. [62, 29]) and Richards et al. [45]. Vitousek et al. describe dynamically checking transient types for Reticulated Python (termed "tag-type" soundness by Greenman and Migeed [29]). As with our work, Vitousek et al.'s transient checks inspect only the "top-level" type of an object. Reticulated Python undertakes these transient type checks at different places to Moth. Moth only checks explicit type annotations, while Reticulated Python implicitly checks whenever values flow from dynamic to static types. We refrain from a direct performance comparison since Reticulated Python is an interpreter without just-in-time compilation and thus performance tradeoffs are different. In recent experimental work, however, Vitousek et al. [61] have evaluated Reticulated Python's transient semantics running on top of an unmodified PyPy JIT metacompiler. These results are broadly consistent with those presented here, finding similarly small slowdowns using just the tracing JIT, and reducing those slowdowns even further when some tests are elimited via static type inference.

Richards et al. [45] take a similar implementation approach to our work, demonstrating that key mechanisms such as object shapes used by a VM to optimize dynamic languages can be used to eliminate most of the overhead of dynamic type checks. Unlike our work, Richards implement "monotonic" gradual typing with blame, rather than the simpler transient checks, and do so on top of an adapted Higgs VM. The Higgs VM implements a baseline just-in-time compiler based on basic-block versioning [21]. In contrast, our implementation of dynamic checks is built on top of the Truffle framework for the Graal VM, and reaches performance approaching that of V8 (cf. Section 5.2). The performance difference is of relevance here since any small constant factors introduced into a VM with a lower baseline performance can remain hidden, while they stand out more prominently on a faster baseline.

Overall, it is unclear whether our results confirm the ones reported by Richards et al. [45], because our system is simpler. It does not introduce the polymorphism issues caused by accumulating cast information on object shapes, which could be important for performance. Considering that Richards et al. report ca. 4% overhead on the classic Richards benchmark, while we see 33%, further work seems necessary to understand the performance implications of their approach for a highly optimizing just-in-time compiler.

8 Conclusion

As gradually typed languages become more common, and both static and dynamically typed languages are extended with gradual features, efficient techniques for gradual type checking become more important. In this paper, we have demonstrated that optimizing virtual machines enable transient gradual type checks with relatively little overhead, and with only small modifications to an AST interpreter. We evaluated this approach with Moth, an implementation of the Grace language on top of Truffle and Graal.

In our implementation, types are structural and shallow: a type specifies only the names of members provided by objects, and not the types of their arguments and results. These types are checked on access to variables, when assigning to method parameters, and also on return values. The information on types is encoded as part of an object's shape, which means that shape checks already performed in an optimizing dynamic language implementation can also be used to check types. Being able to tie checks to the shapes in this way is critical for reducing the overhead of dynamic checking.

Using the Are We Fast Yet benchmarks as well as a collection of benchmarks from the gradual typing literature, we find that our approach to dynamic type checking introduces an overhead of 5% (min. −13%, max. 79%) on peak performance. In addition to the results from further microbenchmarks, we take this as a strong indication that transient gradual types do not need to imply a linear overhead compared to untyped programs. However, we also see that interpreter and startup performance are impacted by the additional type annotations.

Since Moth reaches the performance of a highly optimized JavaScript VM such as V8, we believe that these results are a good indication for the low peak-performance overhead of our approach.

In specific cases, the overhead is still significant and requires further research to be practical. Thus, future research should investigate how the number of gradual type checks can be reduced without causing the type feedback to become too imprecise to be useful. One approach might increase the necessary changes to a language implementation, but avoid checking every variable read. Another approach might further leverage Truffle's self-specialization to propagate type requirements and avoid unnecessary checks.

Finally, we hope to apply our approach to other parts of the spectrum of gradual typing, eventually reaching full structural types with blame that support the gradual guarantee. This should let us verify that Richards et al. [45]'s results generalize to highly optimizing virtual machines, or alternatively, show that other optimizations for precise blame need to be investigated.

References

1 Martín Abadi, Luca Cardelli, Benjamin C. Pierce, and Gordon D. Plotkin. Dynamic Typing in a Statically Typed Language. *ACM Trans. Program. Lang. Syst.*, 13(2):237–268, 1991. doi:10.1145/103135.103138.

2 Amal Ahmed, Robert Bruce Findler, Jeremy G. Siek, and Philip Wadler. Blame for all. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 201–214, 2011. doi:10.1145/1926385.1926409.

3 Esteban Allende, Oscar Callaú, Johan Fabry, Éric Tanter, and Marcus Denker. Gradual typing for Smalltalk. *Sci. Comput. Program.*, 96:52–69, 2014. doi:10.1016/j.scico.2013.06.006.

4 Edd Barrett, Carl Friedrich Bolz-Tereick, Rebecca Killick, Sarah Mount, and Laurence Tratt. Virtual Machine Warmup Blows Hot and Cold. *Proc. ACM Program. Lang.*, 1(OOPSLA):52:1–52:27, October 2017.

5 Spenser Bauman, Carl Friedrich Bolz, Robert Hirschfeld, Vasily Kirilichev, Tobias Pape, Jeremy G. Siek, and Sam Tobin-Hochstadt. Pycket: a tracing JIT for a functional language. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*, pages 22–34, 2015. doi:10.1145/2784731.2784740.

6 Spenser Bauman, Carl Friedrich Bolz-Tereick, Jeremy Siek, and Sam Tobin-Hochstadt. Sound Gradual Typing: Only Mostly Dead. *Proc. ACM Program. Lang.*, 1(OOPSLA):54:1–54:24, October 2017.

7 Michael Bayne, Richard Cook, and Michael D. Ernst. Always-available static and dynamic feedback. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*, pages 521–530, 2011.

8 Gavin M. Bierman, Martín Abadi, and Mads Torgersen. Understanding TypeScript. In *ECOOP 2014 - Object-Oriented Programming - 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014. Proceedings*, pages 257–281, 2014. doi:10.1007/978-3-662-44202-9_11.

9 Andrew P. Black, Kim B. Bruce, Michael Homer, and James Noble. Grace: the absence of (inessential) difficulty. In *Onward! '12: Proceedings 12th SIGPLAN Symp. on New Ideas in Programming and Reflections on Software*, pages 85–98, New York, NY, 2012. ACM.

10 Andrew P. Black, Norman C. Hutchinson, Eric Jul, and Henry M. Levy. The development of the Emerald programming language. In *Proceedings of the Third ACM SIGPLAN History*

of Programming Languages Conference (HOPL-III), San Diego, California, USA, 9-10 June 2007, pages 1–51, 2007. doi:10.1145/1238844.1238855.

11 Bard Bloom, John Field, Nathaniel Nystrom, Johan Östlund, Gregor Richards, Rok Strniša, Jan Vitek, and Tobias Wrigstad. Thorn: Robust, Concurrent, Extensible Scripting on the JVM. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 117–136, 2009.

12 Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. Tracing the Meta-level: PyPy's Tracing JIT Compiler. In *Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, ICOOOLPS '09, pages 18–25. ACM, 2009.

13 Carl Friedrich Bolz, Lukas Diekmann, and Laurence Tratt. Storage Strategies for Collections in Dynamically Typed Languages. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA'13, pages 167–182. ACM, 2013.

14 Carl Friedrich Bolz and Laurence Tratt. The Impact of Meta-Tracing on VM Design and Implementation. *Science of Computer Programming*, 98:408–424, February 2013.

15 John Tang Boyland. The Problem of Structural Type Tests in a Gradual-Typed Language. In *FOOL*, 2014.

16 Gilad Bracha. Pluggable Type Systems. OOPSLA Workshop on Revival of Dynamic Languages, October 2004.

17 Gilad Bracha and David Griswold. Stongtalk: Typechecking Smalltalk in a Production Environment. In *OOPSLA*, 1993.

18 Gilad Bracha, Peter von der Ahé, Vassili Bykov, Yaron Kashai, William Maddox, and Eliot Miranda. Modules as Objects in Newspeak. In *European Conference on Object-Oriented Programming (ECOOP)*, volume 6183 of *Lecture Notes in Computer Science*, pages 405–428. Springer, 2010.

19 Kim Bruce, Andrew Black, Michael Homer, James Noble, Amy Ruskin, and Richard Yannow. Seeking Grace: a new object-oriented language for novices. In *Proceedings 44th SIGCSE Technical Symposium on Computer Science Education*, pages 129–134. ACM, 2013.

20 Craig Chambers, David Ungar, and Elgin Lee. An Efficient Implementation of SELF a Dynamically-Typed Object-Oriented Language Based on Prototypes. In *Proceedings on Object-Oriented Programming Systems, Languages and Applications*, OOPSLA'89, pages 49–70. ACM, October 1989.

21 Maxime Chevalier-Boisvert and Marc Feeley. Interprocedural Type Specialization of JavaScript Programs Without Type Analysis. In *30th European Conference on Object-Oriented Programming (ECOOP 2016)*, volume 56 of *LIPIcs*, pages 7:1–7:24. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016. doi:10.4230/LIPIcs.ECOOP.2016.7.

22 Benjamin Chung, Paley Li, Francesco Zappa Nardelli, and Jan Vitek. KafKa: gradual typing for objects. In *32nd European Conference on Object-Oriented Programming, ECOOP 2018, July 16-21, 2018, Amsterdam, The Netherlands*, pages 12:1–12:24, 2018. doi:10.4230/LIPIcs.ECOOP.2018.12.

23 Daniel Clifford, Hannes Payer, Michael Stanton, and Ben L. Titzer. Memento Mori: Dynamic Allocation-site-based Optimizations. In *Proceedings of the 2015 International Symposium on Memory Management*, ISMM '15, pages 105–117. ACM, 2015.

24 Benoit Daloze, Stefan Marr, Daniele Bonetta, and Hanspeter Mössenböck. Efficient and Thread-Safe Objects for Dynamically-Typed Languages. In *Proceedings of the 2016 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA'16, pages 642–659. ACM, 2016.

25 Ulan Degenbaev, Jochen Eisinger, Manfred Ernst, Ross McIlroy, and Hannes Payer. Idle Time Garbage Collection Scheduling. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI'16, pages 570–583. ACM, 2016.

26 M. Furr, J.-H. An, J. Foster, and M.J. Hicks. Static type inference for Ruby. In *Symposium on Applied Computation*, pages 1859–1866, 2009.

27 Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

28 Ben Greenman and Matthias Felleisen. A spectrum of type soundness and performance. *PACMPL*, 2(ICFP):71:1–71:32, 2018. `doi:10.1145/3236766`.

29 Ben Greenman and Zeina Migeed. On the Cost of Type-Tag Soundness. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, PEPM'18, pages 30–39. ACM, 2018.

30 Ben Greenman, Asumu Takikawa, Max S. New, Daniel Feltey, Robert Bruce Findler, Jan Vitek, and Matthias Felleisen. How to evaluate the performance of gradual type systems. *Journal of Functional Programming*, 29:45, 2019. `doi:10.1017/S0956796818000217`.

31 Christian Humer, Christian Wimmer, Christian Wirth, Andreas Wöß, and Thomas Würthinger. A Domain-Specific Language for Building Self-Optimizing AST Interpreters. In *Proceedings of the 13th International Conference on Generative Programming: Concepts and Experiences*, GPCE '14, pages 123–132. ACM, 2014. `doi:10.1145/2658761.2658776`.

32 Urs Hölzle, Craig Chambers, and David Ungar. Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches. In *ECOOP '91: European Conference on Object-Oriented Programming*, volume 512 of *LNCS*, pages 21–38. Springer, 1991. `doi:10.1007/BFb0057013`.

33 Ralph E. Johnson. Type-Checking Smalltalk. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'86), Portland, Oregon, USA, Proceedings.*, pages 315–321, 1986. `doi:10.1145/28697.28728`.

34 Timothy Jones. *Classless Object Semantics.* PhD thesis, Victoria University of Wellington, 2017.

35 Timothy Jones, Michael Homer, James Noble, and Kim Bruce. Object Inheritance Without Classes. In *30th European Conference on Object-Oriented Programming (ECOOP 2016)*, volume 56, pages 13:1–13:26, 2016.

36 Andre Kuhlenschmidt, Deyaaeldeen Almahallawi, and Jeremy G. Siek. Efficient Gradual Typing. *CoRR*, abs/1802.06375, 2018. `arXiv:1802.06375`.

37 Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. *Object-Oriented Programming in the BETA Programming Language.* Addison-Wesley, 1993.

38 Stefan Marr. SOMns: a newspeak for concurrency research. `https://github.com/smarr/SOMns`, 2018.

39 Stefan Marr. ReBench: Execute and Document Benchmarks Reproducibly, June 2019. Version 1.0rc2. `doi:10.5281/zenodo.3242039`.

40 Stefan Marr, Benoit Daloze, and Hanspeter Mössenböck. Cross-Language Compiler Benchmarking—Are We Fast Yet? In *Proceedings of the 12th Symposium on Dynamic Languages*, DLS'16, pages 120–131. ACM, November 2016.

41 Stefan Marr and Stéphane Ducasse. Tracing vs. Partial Evaluation: Comparing Meta-Compilation Approaches for Self-Optimizing Interpreters. In *Proceedings of the 2015 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '15, pages 821–839. ACM, October 2015.

42 Fabian Muehlboeck and Ross Tate. Sound Gradual Typing is Nominally Alive and Well. *Proc. ACM Program. Lang.*, 1(OOPSLA):56:1–56:30, October 2017.

43 Francisco Ortin, Miguel Garcia, and Seán McSweeney. Rule-based program specialization to optimize gradually typed code. *Knowledge-Based Systems*, 2019. `doi:10.1016/j.knosys.2019.05.013`.

44 Aaron Pang, Craig Anslow, and James Noble. What Programming Languages Do Developers Use? A Theory of Static vs Dynamic Language Choice. In *2018 IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2018, Lisbon, Portugal, October 1-4, 2018*, pages 239–247, 2018. `doi:10.1109/VLHCC.2018.8506534`.

45 Gregor Richards, Ellen Arteca, and Alexi Turcotte. The VM Already Knew That: Leveraging Compile-time Knowledge to Optimize Gradual Typing. *Proc. ACM Program. Lang.*, 1(OOPSLA):55:1–55:27, October 2017.

46 Gregor Richards, Francesco Zappa Nardelli, and Jan Vitek. Concrete Types for TypeScript. In *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic*, pages 76–100, 2015. doi:10.4230/LIPIcs.ECOOP.2015.76.

47 Richard Roberts, Stefan Marr, Michael Homer, and James Noble. Toward Virtual Machine Adaption Rather than Reimplementation. In *MoreVMs'17: 1st International Workshop on Workshop on Modern Language Runtimes, Ecosystems, and VMs at <Programming> 2017*, 2017. Presentation.

48 Jeremy G. Siek and Walid Taha. Gradual typing for functional languages. In *Seventh Workshop on Scheme and Functional Programming*, volume Technical Report TR-2006-06, pages 81–92. University of Chicago, September 2006.

49 Jeremy G. Siek and Walid Taha. Gradual Typing for Objects. In *ECOOP 2007 - Object-Oriented Programming, 21st European Conference, Berlin, Germany, July 30 - August 3, 2007, Proceedings*, pages 2–27, 2007. doi:10.1007/978-3-540-73589-2_2.

50 Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. Refined Criteria for Gradual Typing. In Thomas Ball, Rastislav Bodik, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett, editors, *1st Summit on Advances in Programming Languages (SNAPL 2015)*, volume 32 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 274–293. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2015.

51 Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, Sam Tobin-Hochstadt, and Ronald Garcia. Monotonic References for Efficient Gradual Typing. In *European Symposium on Programming (ESOP)*, pages 432–456, 2015. doi:10.1007/978-3-662-46669-8_18.

52 Jeremy G. Siek and Philip Wadler. Threesomes, with and without blame. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, pages 365–376, 2010. doi:10.1145/1706299.1706342.

53 Vincent St-Amour, Sam Tobin-Hochstadt, and Matthias Felleisen. Optimization coaching: optimizers learn to communicate with programmers. In *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*, pages 163–178, 2012. doi:10.1145/2384616.2384629.

54 G.L. Steele. *Common Lisp the Language*. Digital Press, 1990.

55 Nataliia Stulova, José F. Morales, and Manuel V. Hermenegildo. Reducing the Overhead of Assertion Run-time Checks via Static Analysis. In *Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming*, PPDP'16, pages 90–103. ACM, 2016.

56 Asumu Takikawa, Daniel Feltey, Ben Greenman, Max S. New, Jan Vitek, and Matthias Felleisen. Is Sound Gradual Typing Dead? In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL'16, pages 456–468. ACM, 2016.

57 Asumu Takikawa, T. Stephen Strickland, Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. Gradual typing for first-class classes. In *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*, pages 793–810, 2012. doi:10.1145/2384616.2384674.

58 The Clean. Vehicle. Flying Nun Records, FN147, 1990.

59 Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of Typed Scheme. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, pages 395–406, 2008. doi:10.1145/1328438.1328486.

60 Michael M. Vitousek, Andrew M. Kent, Jeremy G. Siek, and Jim Baker. Design and evaluation of gradual typing for Python. In *DLS'14, Proceedings of the 10th ACM Symposium on Dynamic Languages, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, pages 45–56, 2014. doi:10.1145/2661088.2661101.

61 Michael M. Vitousek, Jeremy G. Siek, and Avik Chaudhuri. Optimizing and Evaluating Transient Gradual Typing. *CoRR*, abs/1902.07808, 2019. arXiv:1902.07808.

62 Michael M. Vitousek, Cameron Swords, and Jeremy G. Siek. Big Types in Little Runtime: Open-world Soundness and Collaborative Blame for Gradual Type Systems. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL'17, pages 762–774. ACM, 2017.

63 Philip Wadler and Robert Bruce Findler. Well-Typed Programs Can't Be Blamed. In *European Symposium on Programming Languages and Systems (ESOP)*, pages 1–16, 2009.

64 Preston Tunnell Wilson, Ben Greenman, Justin Pombrio, and Shriram Krishnamurthi. The Behavior of Gradual Types: A User Study. In *Dynamic Language Symposium (DLS)*, 2018.

65 Andreas Wöß, Christian Wirth, Daniele Bonetta, Chris Seaton, Christian Humer, and Hanspeter Mössenböck. An Object Storage Model for the Truffle Language Implementation Framework. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, PPPJ'14, pages 133–144. ACM, 2014.

66 Thomas Würthinger, Christian Wimmer, Christian Humer, Andreas Wöß, 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'17, pages 662–676. ACM, 2017.

67 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.

68 Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. Self-Optimizing AST Interpreters. In *Proceedings of the 8th Dynamic Languages Symposium*, DLS'12, pages 73–82, October 2012. doi:10.1145/2384577.2384587.

A Typing Discipline for Hardware Interfaces

Jan de Muijnck-Hughes
University of Glasgow, UK
Jan.deMuijnck-Hughes@glasgow.ac.uk

Wim Vanderbauwhede
University of Glasgow, UK
Wim.Vanderbauwhede@glasgow.ac.uk

Abstract

Modern Systems-on-a-Chip (SoC) are constructed by composition of IP (Intellectual Property) Cores with the communication between these IP Cores being governed by well described interaction protocols. However, there is a disconnect between the machine readable specification of these protocols and the verification of their implementation in known hardware description languages. Although tools can be written to address such separation of concerns, the tooling is often hand written and used to check hardware designs *a posteriori*. We have developed a dependent type-system and proof-of-concept modelling language to reason about the physical structure of hardware interfaces using user provided descriptions. Our type-system provides correct-by-construction guarantees that the interfaces on an IP Core will be well-typed if they adhere to a specified standard.

2012 ACM Subject Classification Theory of computation → Type theory; Hardware → System on a chip; Software and its engineering → System description languages

Keywords and phrases System-on-a-Chip, AXI, Dependent Types, Substructural Typing

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2019.6

Supplement Material ECOOP 2019 Artifact Evaluation approved artifact available at
`https://dx.doi.org/10.4230/DARTS.5.2.14`

Funding This work is part of *Border Patrol: Improving Smart Device Security through Type-Aware Systems Design* (EP/N028201/1) and has been sponsored by an EPSRC funding call on *Trust, Identity, Privacy and Security in the Digital Economy.*

Acknowledgements The authors would like to thank the anonymous reviewers for commenting on the paper, and also various members of *Scottish Programming Language* Community (SPLS) for their helpful comments on early versions of the work.

1 Introduction

Hardware Description Languages (HDLs) such as Verilog, SystemVerilog and VHDL are designed to realise both the structure and behaviour of hardware systems. Hardware is modelled as interconnected *components* (modules) that are connected through *ports*; ports being individual wires or a collection of wires. Ports carry data, and the flow of data on a port is directional. HDLs abstract over groupings of ports (*port groups*) as an *interface*, and present values at higher levels of abstraction such as integers and strings. A component can have multiple interfaces that each send multiple values, and can each be characterised differently. An initiating interface (*initiator*) initiates communication, and the targeted interface (*target*) is the recipient of the communication.

Modern hardware design is not just about digital circuits, it is also about describing systems of systems. For instance, System-on-a-Chip (SoC) views hardware modules (IP Cores) as boxes connected using well-known and bespoke interfaces. The structure, and behaviour, of these interfaces are described in natural language documents [3, 41, 4]. Such standards documents will present an *abstract interface description* which is a global view of

33rd European Conference on Object-Oriented Programming (ECOOP 2019).
Editor: Alastair F. Donaldson; Article No. 6; pp. 6:1–6:27
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

an interface agnostic to its endpoint usage, and will provide salient structural information (using natural language) about each port in the interface required for realisation in a HDL. Details provided include a port's size, sensitivity, necessity, flow, and dependencies between the details specified. Further, these documents describe behavioural characteristics of the interfaces as a whole. For example, how ports are grouped to describe different channels.

While circuit-level designs are required to implement behaviour at a low-level, the designer must also ensure that the components in a SoC design are correctly connected according to the provided specification. Standardised machine readable formats such as `IP-XACT` capture many of the structural information found within the standards document's natural language descriptions [23]. However, interface specifications written using `IP-XACT` cannot be parameterised, nor can structural dependencies be specified between ports and over interfaces. Further, not all the information contained within a natural language document can be specified using `IP-XACT`. For instance, `IP-XACT` does not support the definition of *strobe*, a signal carried on a separate port that is linked to an individual bit in multi-wire data bus. The number of strobe is dependent on the size of the bus. Conversely, the machine readable specification can present information more clearly than the specification document itself. For example, port necessity for the `APB` specification is more clearly described in the `IP-XACT` specification than in the standards document.

Generally speaking, there is a disconnect between the description of an interface's structure in a standards document, its representation in a standardised machine readable format, and its enforcement in a HDL. When instantiating these interfaces in a HDL there are no mechanisms to ensure that the characterised interfaces respect the specifications. As a result, mismatches between the specifications and their implementations are common.

1.1 Contributions

The aim of our work is to improve the security and safety of SoC design by utilising state-of-the-art concepts from programming language theory to provide greater correct-by-construction guarantees over the structural and behavioural aspects of SoC designs.

Dependent type-systems present a rich and expressive setting that allows for precise program properties to be stated and verified directly in the language's type-system [28]. Such type-systems also support modelling of resource usage in the style of substructural typing [40, 6]. By building upon existing work from hardware design we can use these concepts to construct a type-based formal description of abstract interfaces, and formally validate that concrete component interfaces adhere to these descriptions at design-time using type checking.

Specifically, we make the following contributions:

1. We present a type-driven modelling framework (Cordial) for reasoning about interfaces on components within a SoC design.
2. We show the use of Cordial for describing an exemplar protocol Mungo, and discuss how Cordial can be used to model real-world protocol specifications: `APB`, `LocalLink` and `AXI` [4, 3, 41].
3. We describe the formalisation of our framework in the dependently typed programming language Idris [9] that also constitutes a proof-of-concept implementation.

Figure 1 summarises the core constructs that comprise Cordial and their relations. Modelling information is taken from the `IP-XACT` standard [23] and existing work [30] to construct a model (θ_{AID}) to represent *abstract interface descriptions.* Our model construction language (λ_{AID}) is a simple extension to the Simply Typed Lambda Calculus (STLC) and

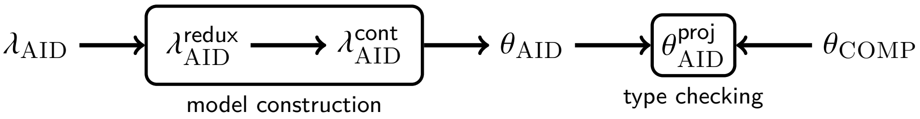

λ_{AID}
$\lambda^{\mathsf{redux}}_{\text{AID}}$
$\lambda^{\mathsf{cont}}_{\text{AID}}$
model construction
θ_{AID}
$\theta^{\mathsf{proj}}_{\text{AID}}$
type checking
θ_{COMP}

Figure 1 Relationships between various languages, models, and intermediate representations.

models parameterised specifications as computable functions, and allows dependencies to be made between signals. The type-system of λ_{AID} follows a substructural design [40, 5] allowing correctness guarantees towards labelling of signals to be lifted into the type-system. Model construction is from reduction of λ_{AID} instances to a reduced form ($\lambda^{\mathsf{redux}}_{\text{AID}}$) which is then evaluated to construct θ_{AID} instances using continuation passing. Concrete interfaces are modelled using θ_{COMP} to present components in a SoC with multiple interfaces.

Inspired by notions of global and local types from *Session Types* [22] abstract interface specifications are treated as a global description that is characterised to a local description – $\theta^{\mathsf{proj}}_{\text{AID}}$. By embedding the projected model ($\theta^{\mathsf{proj}}_{\text{AID}}$) into the type of the interface description (θ_{COMP}) the model's type-system ensures that a local type is satisfied by its global type. Further, the concept of *thinnings* [1, § 3] captures a specification's optional ports, and allows optional ports to be knowingly skipped.

Application of Cordial would see it embedded within existing SoC tooling and to enrich existing HDLs with static design-time mechanisms that would make mismatches between interface specification and implementation impossible and thus reduce errors, increase design productivity and enhance safety and security of the SoC designs. The transformations of specification instances, and model projections would be automatic and hidden from users. Protocol designers would have a tool (based on λ_{AID}) to design interface specifications. During the SoC design phase SoC designers use these specifications to annotate their components (θ_{COMP}) and ensure their port selections are correct.

1.2 Outline

Section 2 presents a running example that further motivates our work. Section 3 introduces our model for abstract interface descriptions (θ_{AID}) and the specification language (λ_{AID}) used to construct θ_{AID} model instances. Section 4 details our model (θ_{COMP}) for describing concrete components and how projected θ_{AID} instances are used to type-check interfaces. Section 5 briefly describes the formalisation of Cordial in Idris, and Section 6 considers use of the framework to model real-world interaction protocol specifications. Section 7 discusses the efficacy of the framework, and considers related work. The paper concludes with a discussion over future work in Section 8.

Notation. For simplicity the syntax for standard algebraic types are abstracted over. Similar abstractions are used for dependent types. Single-field variant types are presented with a constructor name as the label and the body being a n-ary tuple. Where possible simple typing rules are embedded within the presentation of abstract syntax and types. Model types are denoted using blackboard style letters. Types from construction languages are denoted using uppercase Greek letters. Constructs subscripted with: d are from θ_{AID}; and p are from $\theta^{\mathsf{proj}}_{\text{AID}}$.

2 The Mungo Protocol

Presentation of Cordial will be aided through consideration of an exemplar protocol (Mungo) that captures salient physical properties common to many interaction protocols.

Table 1 Signal descriptions for Mungo.

Name	Width	Direction	Necessity	Source	Sensitivity
SYS_CLK	1	Always System	Optional	System	High
CTRL_R	1	To Initiator	Required	IP	High
CTRL_W	1	To Initiator	Required	IP	High
DATA	32/16	Bi-Directional	Required	IP	High
ADDR	8/4	To Target	Required	IP	High
ERR_MODE	2	To Initiator	Target Optional	IP	High
ERR_INFO	user defined	To Initiator	Target Optional	IP	High

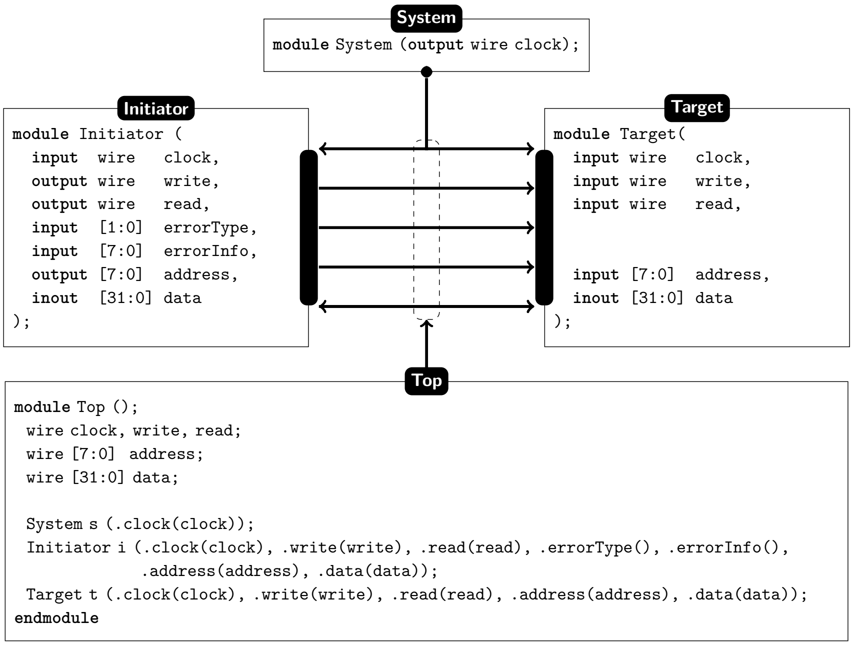
Table 1 presents the signal descriptions (abstract interface description) for Mungo. Behaviourally the protocol represents the reading and writing of data from the initiating IP Core to the target[1]. Mungo provides unicast style communication, it does not support broadcast communication through a shared bus. A system clock (SYS_CLK) can send signals to both the target and initiator. The clock is optional as the clock source for the specified component might not go through this interface. Reading and writing are dictated by the initiator using control wires CTRL_R and CTRL_W. A data bus is bidirectional and data can have a width of 32 or 16 bits. The address bus is eight or four bits in width. Error reporting is optional where: ERR_MODE indicates the type of error; and ERR_INFO is the message itself. The width of error messages are left to the implementer. All wires have high sensitivity.

```
interface Mungo #(AWIDTH = 8, DWIDTH = 32, EWIDTH) (input bit clk);

  logic [AWIDTH-1:0] addr;
  logic [DWIDTH-1:0] data;
  logic [2:0]        errType;
  logic [EWIDTH-1:0] errInfo;
  logic ctrlr, ctrlw;

  modport initiator(input clk,
                    input errType, errInfo,
                    output addr, ctrlw, ctrlr,
                    inout data);

  modport target(input clk,
                 output errType, errInfo,
                 input addr, ctrlw, ctrlr,
                 inout data);
endinterface
```

Figure 2 Realising Mungo using SystemVerilog.

[1] Only the physical structure of the interface is considered. How the framework can be extended to capture behaviour properties is discussed later.

Figure 2 shows how Mungo can be realised in SystemVerilog. The interface is parameterised, however, SystemVerilog only allows such interfaces to have a single default parameter. Mungo has multiple default parameters. Two characterised interfaces for both an initiator and target are presented as modports. Error related signals and clock information are optional. Interfaces can take many other valid structural forms. SystemVerilog supports unrestricted use of *dangling ports*, in which a receiving port is unconnected. In these cases the value received is taken as the default value as dictated by the port's type. A designer can also deviate from the specification and make required ports optional. When connecting two modules together that support Mungo the wrong interface might be left out. Further, not all HDLs support the concept of dangling ports.



```
module System (output wire clock);
```

```
module Initiator (
  input  wire   clock,
  output wire   write,
  output wire   read,
  input  [1:0]  errorType,
  input  [7:0]  errorInfo,
  output [7:0]  address,
  inout  [31:0] data
);
```

```
module Target(
  input wire   clock,
  input wire   write,
  input wire   read,


  input [7:0]  address,
  inout [31:0] data
);
```

```
module Top ();
 wire clock, write, read;
 wire [7:0] address;
 wire [31:0] data;

 System s (.clock(clock));
 Initiator i (.clock(clock), .write(write), .read(read), .errorType(), .errorInfo(),
             .address(address), .data(data));
 Target t (.clock(clock), .write(write), .read(read), .address(address), .data(data));
endmodule
```

Figure 3 SystemVerilog module definitions adhering to Mungo, and signal flow indicators.

Figure 3 presents a second use of SystemVerilog to declare two components that support Mungo, and connect them together. Within Figure 3 we present a visualisation of the module interconnections. Within this example, the initiating component has two dangling ports for optional error reporting, and we have deviated from the specification in using different labels. SystemVerilog provides name and positional oriented connection of modules. Ultimately, the programmer is responsible for ensuring that the ports are connected correctly, and can wire or name ports freely. We need to ensure that the interfaces on a module are valid against their respective specifications.

3 Abstract Interface Descriptions

This section presents a model (θ_{AID}) for reasoning about abstract interface descriptions, together with a language (λ_{AID}) for model construction. How λ_{AID} instances are transformed into θ_{AID} instances is also described.

Taking inspiration from `IP-XACT` [23], abstract interfaces are modelled as a named-tuple of port descriptions and other metadata. This is a common approach as seen by existing work [30, 19]. For each port a variety of emergent properties are also tracked. Dependent types control invariants over model structure and property values. θ_{AID} model instances are not parameterised, the construction language (λ_{AID}) facilitates creation of parameterised specifications and ensures models created use unique labels using substructural typing.

3.1 Properties

Ghica et al. [19] modelled ports according to their size and signal direction. However, there are other important properties as shown by McKechnie [30]. Ports are uniquely identified using *labels*. Similar types of ports share similar behaviour. A port can: communicate data; provide addressing information; provide clock ticks; trigger a reset; signal an interrupt; indicate control; provide port-level behavioural information; or is used in a general sense. Differentiating between these behaviours is important when connecting two (or multiple) ports together. Not all ports in an interface are required, and how a port responds to changes in signal (sensitivity) should also be captured.

For interfaces, salient properties concern the style of communication. Does the interface expect to interact with a set number of other interfaces, or interact directly with another interface?

3.2 Model Components

$$
\begin{array}{rcll}
 & & & \textbf{Metadata} \\
i,n : \mathbb{N}^* & ::= & \text{Natural numbers greater than zero.} & \\
r : L & ::= & \text{User provided labels} & \\
s : \mathbb{S} & ::= & \mathsf{High} \mid \mathsf{Low} \mid \mathsf{Rising} \mid \mathsf{Falling} \mid \mathsf{Insensitive} & \text{Wire Sensitivity} \\
h : \mathbb{H} & ::= & \mathsf{System} \mid \mathsf{IP} & \text{Port Origin} \\
c_{\text{cstyle}} : \mathbb{C}_{\text{style}} & ::= & \mathsf{Broadcast} \mid \mathsf{Unicast} & \text{Comm. Style} \\
 & & & \textbf{Ports} \\
k_p : \mathbb{K}_P & ::= & \mathsf{WIRE} \mid \mathsf{ARRAY} & \text{Kind} \\
\mathbb{A} & : & \mathbb{K}_P \longrightarrow \mathsf{Type} & \text{Type} \\
t : \mathbb{A}\,(\mathsf{ARRAY}) & ::= & \mathsf{Data} \mid \mathsf{Address} & \text{Values} \\
t : \mathbb{A}\,(\mathsf{WIRE}) & ::= & \mathsf{Clock} \mid \mathsf{Reset} \mid \mathsf{Interrupt} \mid \mathsf{Control} & \\
t : \mathbb{A}\,(k_p) & ::= & \mathsf{General} \mid \mathsf{Info} & \\
 & & & \textbf{Labels} \\
k_l : \mathbb{K}_L & ::= & \mathsf{NMD} \mid \mathsf{IDX} & \text{Kind} \\
\mathbb{L} & : & \mathbb{K}_L \longrightarrow \mathsf{Type} \longrightarrow \mathsf{Type} & \text{Type} \\
l : \mathbb{L}(\mathsf{NMD}, L) & ::= & \mathsf{Named}(r) & \text{Values} \\
l : \mathbb{L}(\mathsf{IDX}, L) & ::= & \mathsf{Indexed}(r, i) & \\
 & & & \\
e_c & ::= & n \mid r \mid s \mid h \mid c_{\text{style}} \mid k_p \mid t \mid l & \\
\mathcal{T}_c : \mathsf{Type} & ::= & L \mid \mathbb{N}^* \mid \mathbb{S} \mid \mathbb{H} \mid \mathbb{C}_{\text{style}} \mid \mathbb{K}_L \mid \mathbb{L}(k_l, L) \mid \mathbb{K}_P \mid \mathbb{A}\,(k_p) &
\end{array}
$$

Figure 4 Common terms and their types.

Figure 4 presents the shared terms and types used throughout the models and languages presented. Numbers originate from the set of natural numbers greater than zero. Port labels are specification dependent and assumed to be typed enumerations. Signals are either sensitive or insensitive. Sensitive wires are level sensitive (high, or low) or edge sensitive – rising or falling. Signals either originate from a system interface, or from another component – IP Core. An interface's communication style is either broadcast or unicast.

Of interest is how a port's type and label are modelled. A "kind" provides type-level disambiguation between different kinds of labels and ports. θ_{AID} & λ_{AID} support several types of port, and different port types have different shapes. Data and address ports will always be an array of ports. Clocking information, resets, interrupts, and control ports will always use a single wire. General and information ports can have either shape. When describing widths, the shape of the port will dictate possible values.

Ports must be labelled, however, they can also share a common name with a fixed set of other ports – cf. strobes in `APB` and `AXI`. A label is either *named* and is used once, or *indexed* and used i times. To prevent ambiguities between different label families, the type for labels is indexed with the type associated with the underlying name used.

3.3 A Model for Abstract Interfaces

Figure 5 presents the core modelling constructs, and typing rules, for θ_{AID} model instances. Within θ_{AID}, signal flow is directional. Signals flow from: initiator to target ($\rightsquigarrow$); target to initiator ($\leftsquigarrow$); bidirectional ($\leftrightsquigarrow$); always received ($\overline{\leftsquigarrow}$); or always produced – ($\overline{\rightsquigarrow}$). Ports can be completely optional ($?$), target optional ($?_t$), initiator optional ($?_i$), or are required – ($!$). Wire ports have width ($\mathbb{1}_d$), and array ports have width ($\mathsf{w}_d(n)$) where n is greater than one. Ports can be specified with an arbitrary width – (∞_d). The type for port widths is paramterised by a port kind. This enforces the relation that ports will have the correct width for their kind i.e. a wire can only have length one.

A port description is a named tuple comprising of the port's label (l), kind (k_p), type (t), flow (f), necessity (o), width (w), sensitivity (s), and origin – (h). The type for ports is a type synonym for the following dependent function:

$$\mathsf{port}_p : \mathbb{L}(L, k_l) \rightarrow (k_p : \mathbb{K}_P) \rightarrow \mathbb{A}(k_p) \rightarrow \mathbb{F} \rightarrow \mathbb{O}_d \rightarrow \mathbb{W}_d(k_p) \rightarrow \mathbb{S} \rightarrow \mathbb{H} \rightarrow \mathbb{P}_d(L)$$

Dependently typed terms allow for an invariant to hold during term construction. The port kind associated with a port type and width, must respect the specified port kind. Thus, if the port has kind WIRE then its width and type must be suitable for a wire. Further, the port type itself ($\mathbb{P}_d(L)$) is indexed by the type associated with label.

Ports are grouped in a cons-style collection ($ps : \mathbb{PG}_d(L)$) whose type is also parameterised by the type associated with labels. All ports in a group must have the same type of label. An abstract interface is a named tuple containing the interface's communication style, max number of initiators and targets, and a collection of ports.

3.3.1 Example

Figure 6 presents a θ_{AID} instance for Mungo – Table 1. An enumerated type provides labelling information. θ_{AID} instances are, however, not parameterised. Mungo is an interface that can be instantiated with several address and data bus widths. The example instance for Mungo in Figure 6 provides holes ($\square$) in place of precise widths. Exact values for widths must be presented. Further, there are no restrictions on label use, one can easily duplicate the use of a name. The next section presents a language to present parameterised specifications and ensure label uniqueness.

$$
\begin{array}{rll}
f : \mathbb{F} & ::= \rightsquigarrow \mid \leftsquigarrow \mid \leftrightsquigarrow \mid \overline{\leftsquigarrow} \mid \overline{\rightsquigarrow} & \text{Signal Flow} \\
o_d : \mathbb{O}_d & ::= \, ? \mid ?_i \mid ?_t \mid ! & \text{Necessity} \\
w : \mathbb{W}_d\,(k_p) & ::= \mathbb{1}_d \mid \mathsf{w}_d\,(n) \mid \infty_d & \text{Widths} \\
p_d : \mathbb{P}_d\,(L) & ::= \mathsf{port}_d(l, k_p, t, f, o_d, w_d, s, h) & \text{Port} \\
ps_d : \mathbb{PG}_d\,(L) & ::= \emptyset_d \mid p_d ::_d ps_d & \text{Portgroup} \\
i_d : \mathbb{I}_d\,(L) & ::= \mathsf{iface}_d(\text{cstyle}, n, n, ps_d) & \text{Interface} \\
e_{\text{aidl}} & ::= e_c \mid f \mid o_d \mid w_d \mid p_d \mid ps_d \mid i_d & \text{Expressions} \\
\mathcal{T}_{\text{aidl}} & ::= T \in \mathcal{T}_c \mid \mathbb{F} \mid \mathbb{O}_d \mid \mathbb{W}_d\,(k_p) \mid \mathbb{P}_d\,(L) \mid \mathbb{PG}_d\,(L) \mid \mathbb{I}_d\,(L) & \text{Types}
\end{array}
$$

(a) Terms and types.

$$
\text{DWU}\ \frac{k_p : \mathbb{K}_P}{\infty_d : \mathbb{W}_d\,(k_p)}
\qquad
\text{DWO}\ \frac{}{\mathbb{1}_d : \mathbb{W}_d\,(\mathsf{WIRE})}
\qquad
\text{DWM}\ \frac{i : \mathbb{N}^*, [i \geq 2]}{\mathsf{w}_d\,(i) : \mathbb{W}_d\,(\mathsf{ARRAY})}
$$

$$
\text{PD}\ \frac{l : \mathbb{L}(L, k_l) \qquad ty : \mathbb{A}\,(k_p) \qquad f : \mathbb{F} \qquad \begin{array}{c} k_p : \mathbb{K}_P \\ w_d : \mathbb{W}_d\,(k_p) \end{array} \qquad s : \mathbb{S} \qquad o_d : \mathbb{O}_d \qquad h : \mathbb{H}}{\mathsf{port}_d(l, k_p, ty, f, o_d, w_d, s, h) : \mathbb{P}_d\,(L)}
$$

$$
\text{PGD-E}\ \frac{}{\emptyset_d : \mathbb{PG}_d\,(L)}
\qquad
\text{PGD-C}\ \frac{p : \mathbb{P}_d\,(L) \qquad ps : \mathbb{PG}_d\,(L)}{p ::_d ps : \mathbb{PG}_d\,(L)}
$$

$$
\text{ID}\ \frac{c : \mathbb{C}_{\text{style}} \qquad \text{maxI} : \mathbb{N}^* \qquad \text{maxT} : \mathbb{N}^* \qquad ps : \mathbb{PG}_d\,(L)}{\mathsf{iface}_d(c, \text{maxI}, \text{maxT}, ps) : \mathbb{I}_d\,(L)}
$$

(b) Typing Rules.

Figure 5 Definition of θ_{AID}.

$$
\begin{array}{l}
\qquad\qquad L ::= C \mid R \mid W \mid D \mid A \mid E \mid I \\
\mathsf{iface}_d(\mathsf{Unicast}, 1, 1, \mathsf{port}_d(\mathsf{Named}(C), \mathsf{WIRE}, \mathsf{Clock}, \overline{\leftsquigarrow}, \ ?\ , \mathbb{1}_d, \mathsf{High}, \mathsf{System}) \\
\qquad\qquad ::_d \mathsf{port}_d(\mathsf{Named}(R), \mathsf{WIRE}, \mathsf{Control}, \rightsquigarrow, \ !\ , \mathbb{1}_d, \mathsf{High}, \mathsf{IP}) \\
\qquad\qquad ::_d \mathsf{port}_d(\mathsf{Named}(W), \mathsf{WIRE}, \mathsf{Control}, \rightsquigarrow, \ !\ , \mathbb{1}_d, \mathsf{High}, \mathsf{IP}) \\
\qquad\qquad ::_d \mathsf{port}_d(\mathsf{Named}(D), \mathsf{ARRAY}, \mathsf{Data}, \leftrightsquigarrow, \ !\ , \mathsf{w}_d\,(\square), \mathsf{High}, \mathsf{IP}) \\
\qquad\qquad ::_d \mathsf{port}_d(\mathsf{Named}(A), \mathsf{ARRAY}, \mathsf{Address}, \rightsquigarrow, \ !\ , \mathsf{w}_d\,(\square), \mathsf{High}, \mathsf{IP}) \\
\qquad\qquad ::_d \mathsf{port}_d(\mathsf{Named}(E), \mathsf{ARRAY}, \mathsf{Info}, \leftsquigarrow, \ ?_t\ , \mathsf{w}_d\,(2), \mathsf{High}, \mathsf{IP}) \\
\qquad\qquad ::_d \mathsf{port}_d(\mathsf{Named}(I), \mathsf{ARRAY}, \mathsf{Info}, \leftsquigarrow, \ ?_t\ , \infty_d, \mathsf{High}, \mathsf{IP}) \\
\qquad\qquad ::_d \emptyset_d)
\end{array}
$$

Figure 6 Mungo as a partial θ_{AID} instance.

3.4 Specifying Interface Descriptions

Figure 5 presents a model instance that is dependently typed, however, the model design itself has several limitations. First, labels are not required to be unique. Second, model instances cannot be parameterised. We address these issues through creation of a description language λ_{AID}. An extension of the STLC, λ_{AID} describes the construction of model instances. Specifications are a sequencing of port descriptions, and other metadata. Function, and application, in λ_{AID} provide parameterisation of specifications, and descriptions of structural dependencies. Evaluation of λ_{AID}, using continuation passing, constructs instances of a θ_{AID} model. A substructural type-system provides further correct-by-construction guarantees that labels are unique. Construction semantics detail model instance construction from λ_{AID} programs.

3.4.1 Counting Label Usage

Substructural type-system's extend existing type-systems with extra information [40]. Labels in θ_{AID} instances are required to be unique. The type-system for λ_{AID} is designed to ensure that label usage is linear: A label can only be used once.

Inspired by the work of McBride [29] we utilise a "rig" to capture label usage. For our bespoke use case a rig of the same style is not required. McBride's rig is for computation (addition and multiplication), and our rig is for usage accounting only. We define our rig as:

▶ **Definition 1** (Rig o' 2). *Let* $\mathbb{R} = \{\mathsf{used}, \mathsf{free}\}$ *be a set, with an operation* $\mathsf{use}(u)$ *to change a* $u \in \mathbb{R}$ *as follows:*

$$\mathsf{use}(u) ::= \begin{cases} \mathsf{free} & \mapsto \mathsf{used} \\ \mathsf{used} & \mapsto \mathsf{used} \end{cases}$$

3.4.2 Terms

$$\begin{array}{lr}
e ::= i \mid n \mid r \mid f \mid k_p \mid s \mid h \mid c & \text{Constants} \\
\quad \mid (\mathsf{add}\ e\ e) \mid (\mathsf{sub}\ e\ e) \mid (\mathsf{mul}\ e\ e) \mid (\mathsf{div}\ e\ e) & \text{Maths} \\
\quad \mid \star_1 \mid \omega & \text{Unit Values \& Variables} \\
\quad \mid e; e \mid \lfloor e \rfloor \mid \underline{\mathsf{let}}\ \omega\ \underline{\mathsf{be}}\ e\ \underline{\mathsf{in}}\ e & \text{Statements} \\
\quad \mid (\lambda(i) \cdot e) \mid e \mathbin{\$} n \mid (\lambda(i; [i \in ps]) \ \cdot e) \mid e \mathbin{\$}_{[i \in ps]} i & \text{Function \& Application} \\
\quad \mid \mathsf{label}(n) & \text{Label Creation} \\
\quad \mid \mathsf{portDesc}(\omega, k_p, ty, f, o, w, s, h) \mid \mathsf{replicate}(i, e) & \text{Port Specification} \\
\quad \mid \underline{\mathsf{stop}} & \text{Stopping} \\
\quad \mid \mathsf{setCommunication}(c) \mid \mathsf{setMaxInitiators}(n) & \text{Metadata} \\
\quad \mid \mathsf{setMaxTargets}(n) &
\end{array}$$

Figure 7 Terms for λ_{AID}.

Figure 7 presents the terms for λ_{AID}. Common structures from Figure 4 are included except for the terms for labels. Terms can be sequenced, and bound to variables using let-bindings. Pure values are indicated with $\lfloor e \rfloor$. Combined the terms "Let", "Seq", "Unit" and “Pure” form a monadic computation context in which the labels and their usage are the computation in context. Although, sequencing is presented separately from “Let”-bindings, sequencing can also be described as a “Let”-binding where ω is bound to $\star_1$.

The term stop denotes the end of a specification such that all labels are used. Terms are presented to represent functions, and function application. Predicated versions of functions and application exist to restrict parameters of type $\mathbb{N}^*$ to predefined sets of whole numbers. Whole labels are created using a single term. Port declarations are similar to port construction in Figure 5a, except that rather than a direct label, port descriptions must take a label variable. There are terms for setting communication style, and max number of initiators and targets. Within λ_{AID}, labels are not indexed. Ports with an indexable label are indicated using replicate.

A simple arithmetic language with binary operators to operate on whole numbers is embedded within λ_{AID}. Supported operations are addition, subtraction, multiplication, and division. With this, user provided widths can be used to construct arithmetic dependencies on the number of ports in a specification. This is described using replicate. Allowing for data dependent port specifications (i.e. strobes) to be supported.

3.4.3 Type-System

$$\begin{array}{rlr}
T \in \mathcal{T} ::= & L \mid T_c \in \mathcal{T}_c \setminus (\mathbb{L}(k_l, L)) \mid \mathbb{F} \mid \mathbb{O}_d \mid \mathbb{W}_d\,(k_p) & \text{Types for Constants} \\
& \mid 1 \mid \Lambda\,(L,u) \mid \Psi\,(L) & \text{Types for Unit/Labels/Port} \\
& \mid T \to T \mid (i:\mathbb{N}^*) \xrightarrow{[i \in ps]} T & \text{Types for Functions} \\
\Gamma ::= & \varnothing \mid \Gamma + (e:T) \mid \Gamma \pm (e:T) & \text{Context}
\end{array}$$

Figure 8 Types & typing context for λ_{AID}.

Figure 8 presents the types for λ_{AID}. Here L is a placeholder to represent a user defined set of labels. Several types are taken from existing constructs (Figures 4 and 5a) without the type for labels, ports, port groups and interfaces. Three new types are introduced. First is the unit type (1) to represent terms that do not represent computations. Second is the type for label variables ($\Lambda\,(L,u)$), indexed by the type of the underlying label value and parameterised with usage information from the “Rig o’ 2”. Function types follow the standard definition, and predicated functions are restricted to acting on whole numbers.

Within λ_{AID}well-typed contexts (Γ) comprise of name type pairings. Contexts can be extended using (+), and named terms updated using (±).

Section 3.4.3 presents the typing rules for λ_{AID}. For brevity the typing rules for maths expressions are not provided. Like the syntax definition for λ_{AID} itself, the typing rules follow that of the STLC, but with extensions for describing abstract interfaces. Rules Var,Lam, App, Let, Pure, and Seq follow standard conventions with one noticeable difference that follows from the work of Atkey [5]: Usage information associated with labels presents stateful information. The monad form by “Let”, “Seq”, “Unit”, and “Pure” is a Hoare Monad that allows state information for label usage to be threaded through the entire computation [8]. The notation $\Gamma_{old} \vdash e \dashv \Gamma_{new}$ represents updating the context from Γ_{old} to Γ_{new}. The context will change only if the rules are well-typed. Where the notation is not used implies the context does not change.

$$\textsc{Var}\ \frac{\omega : T \in \Gamma}{\Gamma \vdash \omega : T} \qquad \textsc{Let}\ \frac{\Gamma_1 \vdash a : T_1 \dashv \Gamma_2 \qquad \Gamma_2 + (\omega : T_1) \vdash b : T_2 \dashv \Gamma_3}{\Gamma_1 \vdash \underline{\mathsf{let}}\ \omega\ \underline{\mathsf{be}}\ a\ \underline{\mathsf{in}}\ b : T_2 \dashv \Gamma_3} \qquad \textsc{Pure}\ \frac{x : T \in \Gamma}{\Gamma \vdash \lfloor x \rfloor : T}$$

$$\textsc{Lam}\ \frac{\Gamma_1 + (i : T_1) \vdash e : T_2 \dashv \Gamma_2}{\Gamma_1 \vdash (\lambda(i) \cdot e) : T_1 \rightarrow T_2 \dashv \Gamma_2} \qquad \textsc{App}\ \frac{\Gamma_1 \vdash f : T_1 \rightarrow T_2 \dashv \Gamma_2 \qquad \Gamma_1 \vdash i : T_1}{\Gamma_1 \vdash f\ \$\ i : T_2 \dashv \Gamma_2}$$

$$\text{PLAM}\ \frac{\Gamma_1 + (i : \mathbb{N}^*) \vdash e : T_2 \dashv \Gamma_2 \qquad ps = \{i_1,\ \dots,\ i_n\} \qquad [i \in ps]}{\Gamma_1 \vdash (\lambda(i; [i \in ps])\ \cdot e) : \mathbb{N}^* \xrightarrow{[i \in ps]} T_2 \dashv \Gamma_2}$$

$$\text{PApp}\ \frac{\Gamma_1 \vdash f : (i : \mathbb{N}^*) \xrightarrow{[i \in ps]} T \dashv \Gamma_2 \qquad \Gamma_1 \vdash i' : \mathbb{N}^* \qquad ps = \{i_1,\ \dots,\ i_n\} \qquad [i' \in ps]}{\Gamma_1 \vdash f\ \$_{[i \in ps]} i' : T \dashv \Gamma_2}$$

$$\textsc{Seq}\ \frac{\Gamma_1 \vdash e_1 : T_1 \dashv \Gamma_2 \qquad \Gamma_2 \vdash e_2 : T_2 \dashv \Gamma_3}{\Gamma_1 \vdash e_1 ; e_2 : T_2 \dashv \Gamma_3} \qquad \textsc{Unit}\ \frac{}{\Gamma \vdash \star_1 : 1}$$

$$\text{LBL}\ \frac{\Gamma \vdash L : \mathsf{Type} \qquad \Gamma \vdash n : L}{\Gamma \vdash \mathsf{label}(n) : \Lambda\,(L, \mathsf{free})} \qquad \textsc{Stop}\ \frac{\forall \omega : \Lambda\,(L, u) \in \Gamma\,[u \equiv \mathsf{used}]}{\Gamma \vdash \underline{\mathsf{stop}} : 1 \dashv \varnothing}$$

$$\textsc{Port}\ \frac{\begin{array}{c} \Gamma \vdash l : \Lambda\,(L, \mathsf{free}) \qquad \Gamma \vdash k_p : \mathbb{K}_P \\ \Gamma \vdash ty : \mathbb{A}\,(k_p) \qquad \Gamma \vdash f : \mathbb{F} \qquad \Gamma \vdash o : \mathbb{O}_d \qquad \Gamma \vdash w : \mathbb{W}_d\,(k_p) \qquad \Gamma \vdash s : \mathbb{S} \qquad \Gamma \vdash h : \mathbb{H} \end{array}}{\Gamma \vdash \mathsf{portDesc}(l, k_p, ty, f, o, w, s, h) : \Psi\,(L) \dashv \Gamma \pm (l : \Lambda\,(L, \mathsf{used}))}$$

$$\text{REP}\ \frac{\Gamma_1 \vdash i : \mathbb{N}^* \qquad [i > 2] \qquad \Gamma_1 \vdash e : \Psi\,(L) \dashv \Gamma_2}{\Gamma_1 \vdash \mathsf{replicate}(i, e) : 1 \dashv \Gamma_2} \qquad \textsc{Max-I}\ \frac{\Gamma \vdash n : \mathbb{N}^*}{\Gamma \vdash \mathsf{setMaxInitiators}(n) : 1}$$

$$\textsc{Max-T}\ \frac{\Gamma \vdash n : \mathbb{N}^*}{\Gamma \vdash \mathsf{setMaxTargets}(n) : 1} \qquad \text{COM}\ \frac{\Gamma \vdash c : \mathbb{C}_{\text{style}}}{\Gamma \vdash \mathsf{setCommunication}(c) : 1}$$

Figure 9 Typing rules for λ_{AID}.

Predicated functions, and their application, mirror their plain counterparts but have a side-condition that requires the predicate to hold true for type-checking to be successful.

The typing rules for labels and ports use the "Rig o' 2" to instantiate and augment the usage count for types for labels – $\Lambda\,(L, u)$. These are the type-level computations that enforce correct label usage. Rule LBL presents the typing rule for label creation, initialising the type with usage free. Rule Stop describes the end conditions for λ_{AID} programs, and results in erasure of the context. λ_{AID} programs will successfully type-check only if all label variables have been used. Rule Port specifies a new port, and consumes labels. A label ω with type $\Lambda\,(L, u)$, and usage u can only be used if the usage is free. If the label is available to use the type for ω in resulting computations will be $\Lambda\,(L, \mathsf{used})$, and thus be unavailable.

Replication of a port description (Rule REP) details that a port will be replicated if the number of replications is greater than two. The remaining rules detail the simple typing rules for the remaining terms.

3.4.4 Example

$$
\begin{array}{l}
\varnothing \vdash \textsc{Mungo} : (x : \mathbb{N}^{*}) \xrightarrow{[x \in \{32,16\}]} (y : \mathbb{N}^{*}) \xrightarrow{[y \in \{8,4\}]} 1 \dashv \varnothing \\
\textsc{Mungo} \coloneqq (\lambda(x ; [x \in \{32,16\}]) \cdot (\lambda(y ; [y \in \{8,4\}]) \cdot \\
\quad \mathsf{setCommunication}(\mathsf{Unicast});\ \mathsf{setMaxInitiators}(1);\ \mathsf{setMaxTargets}(1) \\
\ \ ;\ \underline{\mathsf{let}}\ c\ \underline{\mathsf{be}}\ \mathsf{label}(C)\ \underline{\mathsf{in}} \\
\quad \underline{\mathsf{let}}\ r\ \underline{\mathsf{be}}\ \mathsf{label}(R)\ \underline{\mathsf{in}} \\
\quad \underline{\mathsf{let}}\ w\ \underline{\mathsf{be}}\ \mathsf{label}(W)\ \underline{\mathsf{in}} \\
\quad \underline{\mathsf{let}}\ d\ \underline{\mathsf{be}}\ \mathsf{label}(D)\ \underline{\mathsf{in}} \\
\quad \underline{\mathsf{let}}\ a\ \underline{\mathsf{be}}\ \mathsf{label}(A)\ \underline{\mathsf{in}} \\
\quad \underline{\mathsf{let}}\ e\ \underline{\mathsf{be}}\ \mathsf{label}(E)\ \underline{\mathsf{in}} \\
\quad \underline{\mathsf{let}}\ i\ \underline{\mathsf{be}}\ \mathsf{label}(I)\ \underline{\mathsf{in}} \\
\qquad \mathsf{portDesc}(c, \mathsf{WIRE}, \mathsf{Clock}, \overline{\leftsquigarrow}, ?, \mathbb{1}_{d}, \mathsf{High}, \mathsf{System}) \\
\quad\ ;\ \mathsf{portDesc}(r, \mathsf{WIRE}, \mathsf{Control}, \rightsquigarrow, !, \mathbb{1}_{d}, \mathsf{High}, \mathsf{IP}) \\
\quad\ ;\ \mathsf{portDesc}(w, \mathsf{WIRE}, \mathsf{Control}, \rightsquigarrow, !, \mathbb{1}_{d}, \mathsf{High}, \mathsf{IP}) \\
\quad\ ;\ \mathsf{portDesc}(d, \mathsf{ARRAY}, \mathsf{Data}, \leftrightsquigarrow, !, \mathsf{w}_{d}\,(x), \mathsf{High}, \mathsf{IP}) \\
\quad\ ;\ \mathsf{portDesc}(a, \mathsf{ARRAY}, \mathsf{Address}, \rightsquigarrow, !, \mathsf{w}_{d}\,(y), \mathsf{High}, \mathsf{IP}) \\
\quad\ ;\ \mathsf{portDesc}(e, \mathsf{ARRAY}, \mathsf{Info}, \leftsquigarrow, ?_{t}, \mathsf{w}_{d}\,(2), \mathsf{High}, \mathsf{IP}) \\
\quad\ ;\ \mathsf{portDesc}(e, \mathsf{ARRAY}, \mathsf{Info}, \leftsquigarrow, ?_{t}, \infty_{d}, \mathsf{High}, \mathsf{IP}) \\
\quad\ ;\ \underline{\mathsf{stop}}))
\end{array}
$$

Figure 10 Mungo specified using λ_{AID}.

Figure 10 demonstrates how Mungo is specified using λ_{AID}. The concrete set of labels from Figure 6 are reused. The specification for Mungo is parameterised through function specification. This is reflected in the specification's type signature. The use of the specification to create θ_{AID} instances is also restricted to values of $x \in \{32,16\}$ and $y \in \{8,4\}$. A type error will occur if values for x and y were chosen that were not in the provided sets. The type signature also shows the initial and end contexts for the function; start with nothing, end with nothing. Different specification instances are generated through application of the function to different values.

3.5 Building Models from Specifications

In this section the set of transformations to construct θ_{AID} instances from λ_{AID} programs are described. We first reduce λ_{AID} programs to core terms, and use continuations to represent model construction as evaluation of a reduced λ_{AID} program.

Figure 11 presents the reduction rules for reducing λ_{AID} programs to core terms following standard conventions. Reduction of $\mathbb{N}^{*}$ values mirrors reduction of natural numbers but with smallest value being one not zero. The reduced version of a λ_{AID} program is called $\lambda_{\mathrm{AID}}^{\mathsf{redux}}$, and the reduction of a λ_{AID} program l to its reduced form (l') by the operation $l \Downarrow^{redux} l'$. Much like the STLC reduction of λ_{AID} programs will also be strongly normalising.

$$\textsc{Let}\ \frac{e_1 \Downarrow^{\mathsf{redux}} \lfloor e_1' \rfloor}{\underline{\mathsf{let}}\ \omega\ \underline{\mathsf{be}}\ e_1\ \underline{\mathsf{in}}\ e_2 \Downarrow^{\mathsf{redux}} e_2[\omega/e_1']} \qquad \textsc{Lam}\ \frac{f \Downarrow^{\mathsf{redux}} (\lambda(x)\cdot b) \qquad a \Downarrow^{\mathsf{redux}} \lfloor a' \rfloor}{f \mathbin{\$} a \Downarrow^{\mathsf{redux}} b[x/a']}$$

$$\textsc{PLam}\ \frac{f \Downarrow^{\mathsf{redux}} (\lambda(x;[x\in ps])\ \cdot b) \qquad a \Downarrow^{\mathsf{redux}} a' \qquad [a \in ps]}{f \mathbin{\$_{[x\in ps]}} a \Downarrow^{\mathsf{redux}} b[x/a']} \qquad \textsc{Add}\ \frac{e_1 \Downarrow^{\mathsf{redux}} i_1 \qquad e_2 \Downarrow^{\mathsf{redux}} i_2}{(\mathsf{add}\, e_1\, e_2) \Downarrow^{\mathsf{redux}} i_1 + i_2}$$

$$\textsc{Sub}\ \frac{e_1 \Downarrow^{\mathsf{redux}} i_1 \qquad e_2 \Downarrow^{\mathsf{redux}} i_2}{(\mathsf{sub}\, e_1\, e_2) \Downarrow^{\mathsf{redux}} i_1 - i_2} \quad \textsc{Mul}\ \frac{e_1 \Downarrow^{\mathsf{redux}} i_1 \qquad e_2 \Downarrow^{\mathsf{redux}} i_2}{(\mathsf{mul}\, e_1\, e_2) \Downarrow^{\mathsf{redux}} i_1 \times i_2} \quad \textsc{Div}\ \frac{e_1 \Downarrow^{\mathsf{redux}} i_1 \qquad e_2 \Downarrow^{\mathsf{redux}} i_2}{(\mathsf{div}\, e_1\, e_2) \Downarrow^{\mathsf{redux}} i_1 \div i_2}$$

Figure 11 Reduction rules for λ_{AID}.

To construct θ_{AID} model instances, the reduced form, $\lambda^{\mathsf{redux}}_{\mathrm{AID}}$, is first transformed into a continuation ($\lambda^{\mathsf{cont}}_{\mathrm{AID}}$) in the style of Hatcliff and Danvy [21]. This transformation is denoted using $[\![\ l\ ]\!]_{\mathsf{cont}}$, where l is an instance of $\lambda^{\mathsf{redux}}_{\mathrm{AID}}$. Using this approach we can make model construction more easily checkable for termination, and model construction comes from evaluation of $\lambda^{\mathsf{cont}}_{\mathrm{AID}}$ instances. For brevity we do not provide the definitions of $\lambda^{\mathsf{cont}}_{\mathrm{AID}}$ and $[\![\ l\ ]\!]_{\mathsf{cont}}$, and remark that they follow standard contructions [21].

Evaluation of $\lambda^{\mathsf{cont}}_{\mathrm{AID}}$ instances, which we denote using $\Downarrow^{\mathsf{cont}}$, transforms: label variables into labels; port declarations into port descriptions; and repeated port declarations into port groups. Each evaluation of the accessor functions for setting description values replaces the previously see value, and a default set of values are supplied initially. When evaluated, the continuation returns a tuple containing the final interface metadata and collated port groups.

The complete steps to construct a θ_{AID} from λ_{AID} are defined as:

▶ **Definition 2** (Construction of θ_{AID} from λ_{AID})**.** *Let m be a θ_{AID} model instance, and l be a λ_{AID} program. The construction of m is defined as the reduction of l to an instance l' of $\lambda^{\mathsf{redux}}_{\mathrm{AID}}$. This instance l', is then evaluated to an instance of $\lambda^{\mathsf{cont}}_{\mathrm{AID}}$ that is then reduced using $\Downarrow^{\mathsf{cont}}$ to produce m:*

$$[\![\ l\ ]\!]^{\mathrm{AID}}_{\lambda\mapsto\theta} = [\![\ l \Downarrow^{\mathsf{redux}} l'\ ]\!]_{\mathsf{cont}} \Downarrow^{\mathsf{cont}} m$$

4 Specifying IP Core Interfaces

Section 3 described the specification and construction of θ_{AID} instances, these are *abstract interface descriptions*. This section looks at the specification of components in a SoC design and how guarantees are made that the physical interfaces satisfy given θ_{AID} instances.

A model for reasoning about components (θ_{COMP}) is introduced. Each component comprises of a set of physical interface models whose interfaces are satisfied by a θ_{AID} instance. Interface satisfaction is a two-step process: first a θ_{AID} instance is projected ($\upharpoonleft$) using the specified endpoint e, creating the projected model $\theta^{\mathsf{proj}}_{\mathrm{AID}}$. Second, the projected model is used in the type of an interface to provide a type-level invariant that the presented interface satisfies the projected model. Dependently typed model terms ensures that if an interface is well-typed then the interface satisfies the provided specification.

$$
\begin{aligned}
t : \mathbb{E} &::= \mathsf{Initiator} \mid \mathsf{Target} && \text{Endpoint}\\
d_p : \mathbb{D}\,(t, f) &::= + \mid - \mid \pm && \text{Direction}\\
o_p : \mathbb{O}\,(t, o_d) &::= \;?\; \mid ! && \text{Necessity}\\
p_p : \mathbb{P}_p\,(L, t, p_d) &::= \mathsf{port}_p(l, k_p, t, d_p, o_p, w_d, s, h) && \text{Port}\\
ps_p : \mathbb{PG}_p\,(L, t, ps_d) &::= \emptyset_p \mid p_p ::_p ps_p && \text{Portgroup}\\
i_p : \mathbb{I}_p\,(L, t, i_d) &::= \mathsf{iface}_p(c_{\text{style}}, n, n, ps_p) && \text{Interface}\\
e_p &::= e_d \mid t \mid d_p \mid o_p \mid p_p \mid ps_p \mid i_p && \text{Expressions}\\
\mathcal{T}_p &::= T \in \mathcal{T}_d \mid \mathbb{E} \mid \mathbb{D}\,(t, f) \mid \mathbb{O}\,(t, o_d) && \text{Types}\\
&\quad\mid \mathbb{P}_p\,(L, t, p_d) \mid \mathbb{PG}_p\,(L, t, p_d) \mid \mathbb{I}_p\,(L, t, i_d)
\end{aligned}
$$

Figure 12 Terms and types for $\theta^{\mathsf{proj}}_{\text{AID}}$.

4.1 Projecting Abstract Interfaces

Figure 12 presents the terms, and salient types for a projected interface model instance. A projected interface represents the local view of an abstract interface. The structure of a projected interface mirrors that of an abstract interface, and values only differ w.r.t. a port's signal flow (direction), and necessity. The type's for ports, port groups, and interfaces are parameterised by the type of labels associated with ports, the endpoint that the term was projected to, and the originating abstract interface. The types are indexed with the originating term to allow for structural invariants, for example a port's shape, to be specified. – cf. Section 3.3.

A projected port is either unidirectional in receiving (+) or sending signals (−); or is bidirectional – (±). A port is required (!) or optional (?). The types for directions and necessity are both dependent, each containing the endpoint being projected and the original projected value. These values are not free to choose.

$(\upharpoonleft_d)$	$?$	$?_i$	$?_t$	$!$
Target	$?$	$!$	$?$	$!$
Initiator	$?$	$?$	$!$	$!$

(a) Necessity.

$(\upharpoonleft_n)$	$\rightsquigarrow$	$\leftsquigarrow$	$\leftrightsquigarrow$	$\overline{\leftsquigarrow}$	$\overline{\rightsquigarrow}$
Target	$-$	$+$	$\pm$	$+$	$-$
Initiator	$+$	$-$	$\pm$	$+$	$-$

(b) Direction.

Figure 13 Typing rules for flow and necessity projection.

Figure 13 presents the typing rules that constrain the values in the projected model to predetermined pairings. Dependent types ensure the correctness of projection transformation. The projection for port flow and necessity are defined as functions ($(\upharpoonleft_d)$ and $(\upharpoonleft_n)$) that, given a flow or optional description will compute the required direction – or necessity. Their type signatures are:

- $(\upharpoonleft_d) : (d_o : \mathbb{O}_d) \rightarrow (e : \mathbb{E}) \rightarrow \mathbb{O}\,(e, o_d)$
- $(\upharpoonleft_n) : (f : \mathbb{F}) \rightarrow (e : \mathbb{E}) \rightarrow \mathbb{D}\,(e, f)$

These projection functions will only type check if the given inputs match the allowed pairing of values for the returned type.

$$\text{PP}\ \dfrac{\begin{array}{c} l : \mathbb{L}(L, k_l) \\ k_p : \mathbb{K}_P \qquad ty : \mathbb{A}\,(k_p) \qquad d : \mathbb{D}\,(e, f) \qquad o : \mathbb{O}\,(e, o_d) \qquad w_d : \mathbb{W}_d\,(k_p) \qquad s : \mathbb{S} \qquad h : \mathbb{H} \end{array}}{\mathsf{port}_p(l, k_p, ty, d, o, w_d, s, h) : \mathbb{P}_p\,(L, e, \mathsf{port}_d(l, k_p, ty, f, o_d, w_d, s, h))}$$

$$\text{PGP-E}\ \dfrac{}{\emptyset_p : \mathbb{PG}_p\,(L, e, \emptyset_d)} \qquad \text{PGP-C}\ \dfrac{p_p : \mathbb{P}_p\,(L, e, p_d) \qquad ps_p : \mathbb{PG}_p\,(L, e, ps_d)}{p_p ::_p ps_p : \mathbb{PG}_p\,(L, e, p_d ::_d ps_d)}$$

$$\text{IP}\ \dfrac{c : \mathbb{C}_{\text{style}} \qquad \text{maxI} : \mathbb{N}^* \qquad \text{maxT} : \mathbb{N}^* \qquad ps_p : \mathbb{PG}_p\,(L, e, ps_d)}{\mathsf{iface}_p(c, \text{maxI}, \text{maxT}, ps_p) : \mathbb{I}_p\,(L, e, \mathsf{iface}_d(c, \text{maxI}, \text{maxT}, ps_d))}$$

Figure 14 Typing rules for $\theta^{\mathsf{proj}}_{\text{AID}}$.

Figure 14 presents the remaining typing rules for projected interfaces, ports, and port groups. Like the structural definition, the typing rules mirror those for their abstract counterparts, and are indexed by the type associated with labels. However, the types are further indexed by their abstract counterparts, and also by the endpoint the term is being projected under. The θ_{AID} instance provides as a type-level meta-model from which information is sourced. This approach allows for several invariants on the structure of the projected interface to be established.

1. Non-projected values *must* match.
2. Values parameterising the type of projected values are sourced from the abstract description and *must* match.
3. The endpoint that terms are being projected under *must* match.
4. The structure of the projected interface *must* match the structure of the abstract interface.

$$\boxed{\theta_{\text{AID}} \mapsto \theta^{\mathsf{proj}}_{\text{AID}}}$$

$$\begin{aligned}
\mathsf{port}_d(l, k_p, t, d_p, o_p, w_d, s, h) \upharpoonright_p e &::= \mathsf{port}_p(l, k_p, t, (d_p \upharpoonright_d e), (o_p \upharpoonright_n e), w_d, s, h) \\
\emptyset_d \upharpoonright_g e &::= \emptyset_p \\
p_d ::_d ps_d \upharpoonright_g e &::= (p_d \upharpoonright_p e) ::_p (ps_d \upharpoonright_g e) \\
\mathsf{iface}_d(\text{cstyle}, a, b, ps_p) \upharpoonright_i e &::= \mathsf{iface}_p(\text{cstyle}, a, b, (ps_p \upharpoonright_g e))
\end{aligned}$$

Figure 15 Projection semantics for $\theta^{\mathsf{proj}}_{\text{AID}}$.

Figure 15 presents the projection semantics for projecting θ_{AID} instances to $\theta^{\mathsf{proj}}_{\text{AID}}$ instances. The type signatures for each projection function are omitted for brevity, but they follow those for $(\upharpoonright_d)$ and $(\upharpoonright_n)$. By design, projections and their invariants are well-typed. Malformed projections will fail to type-check, for example if the widths or calculated directions are wrong.

4.1.1 Example

Figure 16 presents example projections for the θ_{AID} instance for Mungo, applied to parameters 32 and 8 using predicated function application. Figure 16a shows a projection for an initiator interface, and Figure 16b shows a projection for a target interface. The set of

$$\boxed{[\![\, (\textsc{Mungo}\ \$_{\{32,16\}}32\ \$_{\{8,4\}}8)\,]\!]^{\mathrm{AID}}_{\lambda\mapsto\theta} \upharpoonright_i \mathsf{Initiator}}$$

$$\begin{aligned}
&\mathsf{iface}_p(\mathsf{Unicast}, 1, 1, \mathsf{port}_p(\mathsf{Named}(C), \mathsf{WIRE}, \mathsf{Clock}, +, ?, \mathbb{1}_d, \mathsf{High}, \mathsf{System})\\
&\qquad\qquad ::_p \mathsf{port}_p(\mathsf{Named}(R), \mathsf{WIRE}, \mathsf{Control}, -, !, \mathbb{1}_d, \mathsf{High}, \mathsf{IP})\\
&\qquad\qquad ::_p \mathsf{port}_p(\mathsf{Named}(W), \mathsf{WIRE}, \mathsf{Control}, -, !, \mathbb{1}_d, \mathsf{High}, \mathsf{IP})\\
&\qquad\qquad ::_p \mathsf{port}_p(\mathsf{Named}(D), \mathsf{ARRAY}, \mathsf{Data}, \pm, !, \mathsf{w}_d\,(32), \mathsf{High}, \mathsf{IP})\\
&\qquad\qquad ::_p \mathsf{port}_p(\mathsf{Named}(A), \mathsf{ARRAY}, \mathsf{Address}, -, !, \mathsf{w}_d\,(8), \mathsf{High}, \mathsf{IP})\\
&\qquad\qquad ::_p \mathsf{port}_p(\mathsf{Named}(E), \mathsf{ARRAY}, \mathsf{Info}, +, !, \mathsf{w}_d\,(2), \mathsf{High}, \mathsf{IP})\\
&\qquad\qquad ::_p \mathsf{port}_p(\mathsf{Named}(I), \mathsf{ARRAY}, \mathsf{Info}, +, !, \infty_d, \mathsf{High}, \mathsf{IP})\\
&\qquad\qquad ::_p \emptyset_p)
\end{aligned}$$

(a) Mungo projected as an initiator interface.

$$\boxed{[\![\, (\textsc{Mungo}\ \$_{\{32,16\}}32\ \$_{\{8,4\}}8)\,]\!]^{\mathrm{AID}}_{\lambda\mapsto\theta} \upharpoonright_i \mathsf{Target}}$$

$$\begin{aligned}
&\mathsf{iface}_p(\mathsf{Unicast}, 1, 1, \mathsf{port}_p(\mathsf{Named}(C), \mathsf{WIRE}, \mathsf{Clock}, +, ?, \mathbb{1}_d, \mathsf{High}, \mathsf{System})\\
&\qquad\qquad ::_p \mathsf{port}_p(\mathsf{Named}(R), \mathsf{WIRE}, \mathsf{Control}, +, !, \mathbb{1}_d, \mathsf{High}, \mathsf{IP})\\
&\qquad\qquad ::_p \mathsf{port}_p(\mathsf{Named}(W), \mathsf{WIRE}, \mathsf{Control}, +, !, \mathbb{1}_d, \mathsf{High}, \mathsf{IP})\\
&\qquad\qquad ::_p \mathsf{port}_p(\mathsf{Named}(D), \mathsf{ARRAY}, \mathsf{Data}, +, !, \mathsf{w}_d\,(32), \mathsf{High}, \mathsf{IP})\\
&\qquad\qquad ::_p \mathsf{port}_p(\mathsf{Named}(A), \mathsf{ARRAY}, \mathsf{Address}, +, !, \mathsf{w}_d\,(8), \mathsf{High}, \mathsf{IP})\\
&\qquad\qquad ::_p \mathsf{port}_p(\mathsf{Named}(E), \mathsf{ARRAY}, \mathsf{Info}, -, ?, \mathsf{w}_d\,(2), \mathsf{High}, \mathsf{IP})\\
&\qquad\qquad ::_p \mathsf{port}_p(\mathsf{Named}(I), \mathsf{ARRAY}, \mathsf{Info}, -, ?, \infty_d, \mathsf{High}, \mathsf{IP})\\
&\qquad\qquad ::_p \emptyset_p)
\end{aligned}$$

(b) Mungo projected as a target interface.

Figure 16 Mungo projected as $\theta^{\mathsf{proj}}_{\mathrm{AID}}$ instances.

directions for each port are mirror images of each other, aside from the constant direction for the system clock and the bi-directional data port. Further, the definition of a port's necessity have been calculated to respect if the port is optional or not. The ports for returning error information are optional if the interface's endpoint is a target interface and required if the endpoint is an initiator.

4.2 Specifying Physical Interfaces

Abstract interfaces, and their projections, represent *descriptions* of a component's interface, we must also model the component itself. Figure 17 presents the terms and types for θ_{COMP} model instances. Figure 18 presents the typing rules.

The structure of concrete interfaces mirrors that of the abstract interface and projection. However, a concrete interface does not have optional ports, within our model dangling ports are not allowed. To model skippable ports the concept of thinnings is used [1]. A thinning allows for structures to be weakened using some decision procedure [13, 2]. We can use this concept to *weaken* the specified ports in an interface's portgroup w.r.t. a given specification. Our decision procedure is simple: a port can be skipped if the projected port is optional. The

$$
\begin{array}{rcll}
w : \mathbb{W}\,(k_p, w_d) & ::= & \mathsf{u}\,(i) \mid \mathbb{1} \mid \mathsf{w}\,(i) & \text{Widths} \\
p : \mathbb{P}\,(L, e, p_p) & ::= & (l, k_p, t, d, w, s, h) & \text{Ports} \\
ps : \mathbb{P}\mathbb{G}\,(L, e, ps_p) & ::= & \emptyset \mid p :: ps \mid \sqcup ps \mid p ::\approx ps & \text{Port Groups} \\
i : \mathbb{I}\,(L, e, i_p) & ::= & (ps) & \text{Interfaces} \\
is : \mathbb{I}\mathbb{G}\,(is_d, es) & ::= & \emptyset_i \mid i ::_i is & \text{Interface Group} \\
c : \mathbb{C}\,(xs) & ::= & \mathsf{comp}(is) & \text{Components} \\
e_{\text{CSL}} & ::= & e_{\text{aidl}} \mid w \mid p \mid ps \mid i \mid is \mid c & \text{Expressions} \\
\mathcal{T}_{\text{csl}} & ::= & \mathcal{T}_{\text{aidl}} \mid \mathbb{W}\,(k_p, w_d) \mid \mathbb{P}\,(L, e, p_p) & \text{Types} \\
 & & \mid \mathbb{P}\mathbb{G}\,(L, e, ps_p) \mid \mathbb{I}\,(L, e, i_p) \mid \mathbb{I}\mathbb{G}\,(is_d, es) \mid \mathbb{C}\,(xs) &
\end{array}
$$

Figure 17 Terms for concrete interfaces.

$$
\text{W-Z}\ \frac{i : \mathbb{N}^* \qquad \mathsf{shape} : \mathbb{K}_P}{\mathsf{u}\,(i) : \mathbb{W}\,(\mathsf{shape}, \infty_d)}
\qquad
\text{W-O}\ \frac{}{\mathbb{1} : \mathbb{W}\,(\mathsf{WIRE}, \mathbb{1}_d)}
\qquad
\text{W-A}\ \frac{i : \mathbb{N}^*}{\mathsf{w}\,(i) : \mathbb{W}\,(\mathsf{ARRAY}, \mathsf{w}_d\,(i))}
$$

$$
\textsc{Port}\ \frac{\begin{array}{c} l : \mathbb{L}(L, k_l) \\ k_p : \mathbb{K}_P \qquad ty : \mathbb{A}\,(k_p) \qquad w : \mathbb{W}\,(k_p, w_d) \qquad s : \mathbb{S} \qquad h : \mathbb{H} \qquad d : \mathbb{D}\,(e, f) \end{array}}{(l, k_p, t, d, w, s, h) : \mathbb{P}\,(L, e, \mathsf{port}_p(l, k_p, ty, d, o, w_d, s, h))}
$$

$$
\text{PGP-E}\ \frac{}{\emptyset : \mathbb{P}\mathbb{G}_p\,(L, e, \emptyset_d)}
\qquad
\text{PG-C}\ \frac{p : \mathbb{P}\,(L, e, p_p) \qquad ps : \mathbb{P}\mathbb{G}\,(L, e, ps_p)}{p :: ps : \mathbb{P}\mathbb{G}\,(L, e, p_p :: ps_p)}
$$

$$
\text{PG-S}\ \frac{ps : \mathbb{P}\mathbb{G}\,(L, e, ps_p)}{\sqcup ps : \mathbb{P}\mathbb{G}\,(L, e, \mathsf{port}_p(l, k_p, t, d_p, ?\,, w_d, s, h) :: ps_p)}
$$

$$
\text{PG-SC}\ \frac{p : \mathbb{P}\,(L, e, p_p) \qquad ps : \mathbb{P}\mathbb{G}\,(L, e, ps_p)}{p ::\approx ps : \mathbb{P}\mathbb{G}\,(L, e, p_p ::_p \mathsf{port}_p(l, k_p, t, d_p, ?\,, w_d, s, h) ::_p ps_p)}
$$

$$
\textsc{Interface}\ \frac{ps : \mathbb{P}\mathbb{G}\,(L, e, ps_p)}{(ps) : \mathbb{I}\,(L, e, \mathsf{iface}_p(c, \mathrm{maxI}, \mathrm{maxT}, ps_p))}
\qquad
\text{IG-E}\ \frac{}{\emptyset_i : \mathbb{I}\mathbb{G}\,(\emptyset_d, \emptyset_e)}
$$

$$
\text{IG-C}\ \frac{i : \mathbb{I}\,(L, e, i_d \upharpoonleft_i e) \qquad is : \mathbb{I}\mathbb{G}\,(is_d, es)}{i ::_i is : \mathbb{I}\mathbb{G}\,(i_d ::_i is_d, e ::_e es)}
$$

$$
\textsc{Component}\ \frac{xs = \{(i_{d,0}, e_0), \ldots, (i_{d,j}, e_j)\} \qquad is : \mathbb{I}\mathbb{G}\,(\cup_{k=0}^{j} i_{d,k}, \cup_{k=0}^{j} e_k)}{\mathsf{comp}(is) : \mathbb{C}\,(xs)}
$$

Figure 18 Typing rules for concrete interfaces.

thinning decision procedure does not occur at the value level. Thus a concrete port group is either: empty – $\emptyset$; extended by a port ($::$); skipped by an optional port ($\sqcup$); or an optional port is skipped when extending the group with a port – ($::\approx$). The operator ($::\approx$) can be defined as the combination of the ($::$) and ($\sqcup$) operators. That is $p :: (\sqcup ps) \equiv p ::\approx ps$. The typing rules (Figure 18) show how thinning works for interface specifications. The $\theta_{\text{AID}}^{\mathsf{proj}}$ is a type-level invariant, the specification of the necessity of the projected ports is what allows the thinning to occur, or not.

A component is modelled as a collection of interfaces. The type for a component is indexed by a collection (xs) of θ_{AID}-endpoint pairings. The type for a collection of interfaces is indexed by the separated elements of each pair in xs. As a collection of interfaces is constructed, the θ_{AID} instances are projected (at the type-level) by the endpoint type to construct the $\theta_{\text{AID}}^{\mathsf{proj}}$ instance indexing the collected interface. Use of projection at the type-level ensures that the type for a concrete interface is sourced from the specified projection.

In θ_{AID} model instances, wires have width one, arrays have fixed width greater than one, or are unrestricted. When projecting an abstract port, the width does not need to be projected into a local value. However, the port width in a θ_{COMP} instance needs to respect the width in the θ_{AID}. Widths are modelled using a dependent data type that captures, and reasons with, the width of an abstract port.

An instantiated port with an abstract port and an unrestricted width has no restrictions on kind or width. A port with an abstract port of width one *must* also have a width of one. Similarly, a port with an abstract port of fixed width *must* also have the same fixed width.

4.2.1 Example

Figure 19 presents two example interfaces that model MUNGO. Figure 19b shows the interface with a port to receive the clock, and Figure 19c without a clock port. Within Figure 19c the skip term ($\sqcup$) allows for the clock to the be skipped. If other required ports were to be skipped this will result in a type error. For both interfaces we have chosen the user defined error message width to be of width 32 and 16. This will not cause a type error as the MUNGO allows the signal to have a user-defined width. Further, should other value level information (e.g. port width and label) be incorrectly specified the example will also fail to type-check.

4.3 Type-Checking Interfaces

The type of interfaces in θ_{COMP} are parameterised by projected interfaces from $\theta_{\text{AID}}^{\mathsf{proj}}$. A satisfaction relation is defined to link programs written in λ_{AID} to interfaces from θ_{COMP}.

▶ **Definition 3** (Interface Satisfaction). *Given a λ_{AID} specification ($\varnothing \vdash v : 1 \dashv \varnothing$), and an interface $\vartheta : \mathbb{I}(L, e, \vartheta_p)$ then the interface ϑ satisfies v, which is defined as $\vartheta \models v$, if $[\![\, v \,]\!] \upharpoonright_i e \mapsto \vartheta_p$.*

Proof. By construction. The evaluation of v ($[\![\, v \,]\!]$) produces a model ($\vartheta_d : \mathbb{I}_d\,(L')$), for some ($L' : \mathsf{Type}_L$). The type of ϑ is $\mathbb{I}(L, e, \vartheta_p)$. The projected interface ϑ_p has type $\mathbb{I}_p\,(L, e, \vartheta'_d)$. If $\vartheta_d \equiv \vartheta'$ and $L \equiv L'$ then $\vartheta_d \upharpoonright_i e \equiv \vartheta'_d \upharpoonright_i e$. If $\vartheta_d \not\equiv \vartheta'_d$ or $L \not\equiv L'$ then ϑ would fail to type check. ◀

5 Implementation

We have realised CORDIAL using Idris, a general purpose dependently typed programming language [9]. This provides both a practical proof-of-concept implementation, and mechanised formalisation that the framework's type-system holds. The models representing interfaces, port groups, and ports translate directly to standard dependently (and non-dependently)

$$\mathbb{I}\,(L, \mathsf{Initiator}, [\![\ (\textsc{Mungo}\ \$_{\{32,16\}}32\ \$_{\{8,4\}}8)\ ]\!]^{\text{AID}}_{\lambda\mapsto\theta} \upharpoonright_i \mathsf{Initiator})$$

(a) Type for both Interfaces.

$$\begin{aligned}
&(\mathsf{Named}(C), \mathsf{WIRE}, \mathsf{Clock}, +, \mathbb{1}, \mathsf{High}, \mathsf{System})\\
::_i&(\mathsf{Named}(R), \mathsf{WIRE}, \mathsf{Control}, -, \mathbb{1}, \mathsf{High}, \mathsf{IP})\\
::_i&(\mathsf{Named}(W), \mathsf{WIRE}, \mathsf{Control}, -, \mathbb{1}, \mathsf{High}, \mathsf{IP})\\
::_i&(\mathsf{Named}(D), \mathsf{ARRAY}, \mathsf{Data}, \pm, \mathsf{w}\,(32), \mathsf{High}, \mathsf{IP})\\
::_i&(\mathsf{Named}(A), \mathsf{ARRAY}, \mathsf{Address}, -, \mathsf{w}\,(4), \mathsf{High}, \mathsf{IP})\\
::_i&(\mathsf{Named}(E), \mathsf{ARRAY}, \mathsf{Info}, +, \mathsf{w}\,(2), \mathsf{High}, \mathsf{IP})\\
::_i&(\mathsf{Named}(I), \mathsf{ARRAY}, \mathsf{Info}, +, \mathsf{u}\,(32), \mathsf{High}, \mathsf{IP})\\
::_i&\emptyset_i
\end{aligned}$$

(b) With a clock port.

$$\begin{aligned}
\sqcup&(\mathsf{Named}(R), \mathsf{WIRE}, \mathsf{Control}, -, \mathbb{1}, \mathsf{High}, \mathsf{IP})\\
::_i&(\mathsf{Named}(W), \mathsf{WIRE}, \mathsf{Control}, -, \mathbb{1}, \mathsf{High}, \mathsf{IP})\\
::_i&(\mathsf{Named}(D), \mathsf{ARRAY}, \mathsf{Data}, \pm, \mathsf{w}\,(32), \mathsf{High}, \mathsf{IP})\\
::_i&(\mathsf{Named}(A), \mathsf{ARRAY}, \mathsf{Address}, -, \mathsf{w}\,(4), \mathsf{High}, \mathsf{IP})\\
::_i&(\mathsf{Named}(E), \mathsf{ARRAY}, \mathsf{Info}, +, \mathsf{w}\,(2), \mathsf{High}, \mathsf{IP})\\
::_i&(\mathsf{Named}(I), \mathsf{ARRAY}, \mathsf{Info}, +, \mathsf{u}\,(16), \mathsf{High}, \mathsf{IP})\\
::_i&\ \emptyset_i
\end{aligned}$$

(c) Without a clock port.

Figure 19 Sample θ_{COMP} instances that show port skipping.

typed algebraic data structures. λ_{AID}, and its substructural type-system, has been implemented as an Embedded Domain Specific Language (EDSL) using standard techniques [12, Chp. 14]. Predicated functions were realised using Idris' support for *auto implicits* that attempt to search for constructors that can satisfy the type of the implicit variable. The "model construction" process (i.e. $[\![\ l\ ]\!]^{\text{AID}}_{\lambda\mapsto\theta}$), again uses standard techniques for working with EDSLs in dependently typed languages [10, 11, 7].

6 Case Studies

Using our implementation we have constructed models for several interaction protocols of varying sizes. We describe the structural properties of the protocols and report on their modeling in Cordial as λ_{AID} programs.

6.1 ARMs Advanced Peripheral Bus

The first protocol considered was ARMs *Advanced Peripheral Bus* (`APB`) [3]. The protocol is a legacy protocol comprising of at least eleven signals, and can be connected to many target IP Cores using an intermediary IP Core called an interconnect. This requires that we construct two specifications one for target connections to the interconnect, and the other for initiating connections. At least seven signals are required, and two were target optional. At least seven signals have a known width. The data and address width can be up to 32 bits in width. There are three interesting features of APB worth noting. First, the specification is parameterised depending on the number of IP Cores that an initiator is connecting too. When connecting to x targets, the signal PSelx will be replicated x times in an initiating interface, which will only be seen once in a target interface. Second, the protocol has optional signals for the clock and a signal to enable the clock. The dependency between the two clock signals is not clear from the specification: Can one be skipped when the other is required? Third, the specification requires a set of strobes that connects to every 8^{th} bit on the data bus. This restricts data widths to multiples of eight.

$$
\begin{array}{l}
(\lambda(\mathrm{nrSlaves}) \cdot (\lambda(\mathrm{dwidth}; \{8,16,32\}) \ \cdot (\lambda(\mathrm{awidth}; \{8,16,32\}) \cdot \\
\quad \underline{\mathsf{let}}\ x\ \underline{\mathsf{be}}\ (\mathsf{div}\ \mathrm{dwidth}\ 8)\ \underline{\mathsf{in}} \\
\quad \underline{\mathsf{let}}\ \alpha\ \underline{\mathsf{be}}\ \mathsf{label}(\mathrm{T})\ \underline{\mathsf{in}}\ \underline{\mathsf{let}}\ \omega\ \underline{\mathsf{be}}\ \mathsf{label}(\mathrm{S})\ \underline{\mathsf{in}} \\
\quad \ldots \\
\quad ;\ \ \mathsf{replicate}(\mathrm{nrSlaves}, \mathsf{portDesc}(\alpha, \mathsf{WIRE}, \mathsf{Info}, \rightsquigarrow, !\,, \mathbb{1}_d, \mathsf{High}, \mathsf{IP})) \\
\quad ;\ \ \mathsf{portDesc}(\omega, \mathsf{ARRAY}, \mathsf{Info}, \leftsquigarrow, ?\,, \mathsf{w}_d\,(x), \mathsf{High}, \mathsf{IP}) \\
\quad ;\ \ \ldots)))
\end{array}
$$

Figure 20 Partial presentation of the “Master Interface” for the APB specification.

We chose to implement the specifications using two predicated functions, and one normal function. The two predicated functions was used to ensure that all bus widths are multiples of eight and less than 32 i.e. 8, 16, and 32. The normal function was used to represent the number of targets. Specification of labels and ports followed the known information taken from the specification. The term replicate was used to ensure that the correct number of PSelx signals were generated. For strobes, the maths operations were used to calculate the number of strobes required, based of the size of the data bus. Figure 20 shows part of the APB specification detailing the predicated functions, use of replicate, and strobe specifcation.

6.2 Xilinx’s LocalLink

The second protocol chosen was a legacy protocol from Xilinx called LocalLink [41]. The LocalLink protocol requires seven signals and thirteen optional signals. As with the APB protocol many of the signals were encoded without complications. However, and unlike APB, LocalLink does not specify a set of bus widths and requires that the presented bus width is a multiple of eight. Predicated functions require that the list of possible values are known *a priori*. Thus, we modelled the specification as a predicated function on bus widths of 8, 16, and 32, together with a normal function that takes the number of channels.

The LocalLink specification contains various size dependencies based on the data bus. The size of the remainder bus is dependent on how they are implemented within the IP Core. Specifically, if the remainder is encoded then the bus size will be $\lceil \log_2 \frac{d}{8} - 1 \rceil$. If the remainder is masked then the size is $\frac{d}{8} - 1$. Our framework is not expressive enough to encode these differences mathematically, we support simple arithmetic operations and these operations are not simple. Moreover, our framework does not support related specifications that overlap to be collapsed into a single definition. The LocalLink specification is *too* value dependent.

Another interesting aspect of the LocalLink specification is that of *channels*. These are a set of optional signals whose flow is dependent on the application context. The size of *channel* related signals are too calculated using a floating point operation. These three signals are directional, and whether they send data from target to initiator is dependent on the application, and how the other two signals are specified. Although we can represent these channels as being bidirectional our language is not expressive enough to capture these application specific, and inter signal, dependencies.

6.3 ARMs Advanced eXtensible Interface

ARMs *Advanced eXtensible Interface* (`AXI`) is a widely used family of interaction protocols [4] for transferring addressable data between IP Cores. There are several previous versions of `AXI` each building upon previous versions. Each version differs by number of signals and changes to specific signal properties. The `AXI` specification defines three protocols that offer three different interaction styles: Full, Lite, and Stream. The protocol can be directly connected to another interface, or it can be used to connect to multiple other IP Cores using an interconnect. Version four of the protocol requires 47 signals comprising of: two global signals for the clock and a reset; thirteen signals each for specifying writing and reading of an address; seven signals for reading and writing data; and five signals for writing responses from the target to the initiator. Of the 47 signals, 36 are required and eleven are either optional, target optional, or initiator optional. Several signals have user defined widths. The `AXI` protocol is parameterised such that the address and data busses can be: 8, 16, 32, 64, 128, 256, 512, or 1024 bits wide. Further, the protocol specification supports custom sets of signals that are completely user definable. In fact the `AXI` standard explicitly warns against their use due to potential interoperability issues if different modules present user-defined signals that behave differently.

We report on describing version four of the `AXI` protocol for direct interfaces only. For versions of the protocol that connect to an interconnect, the techniques presented in Section 6.1, can be leveraged. The specification was modelled in λ_{AID} using two predicated functions to control the width of the address and data busses. Each of the signals were translated into the required port descriptions. Like the `APB` protocol, `AXI` has the concept of strobes. The same technique as used in Section 6.1 was used.

We chose not to include user-defined signals in this study. Cordial requires that signal details are known *a priori*. Ideally, designers would create parameterised specifications (functions) that take as parameters the user-defined signals. However, functions in λ_{AID} take pure values as parameters, port declarations modify the typing environment to update label usage information and are thus not pure. This is a restriction from the substructural type-system for λ_{AID} – see Section 7.3.

The protocol specification divides the signals among several channels. Such a grouping is only required for the specification. At the module level these groupings do not exist. Although, Cordial does not support this grouping incorporating this into the framework would be a potential benefit for reasoning about subgroupings of an interface's portgroup.

7 Discussion & Related Work

This section discusses the efficacy of the framework, and related work.

7.1 Discussion

Section 6 described three case studies that modelled three interaction protocols using λ_{AID}. For each of the protocols we were able to encode most of the ports correctly Cordial is suitable for capturing each port's values. However, both `LocalLink` and `APB` illustrated the limitations of λ_{AID} in capturing all of the specification's dependencies.

While Cordial can capture limited value dependencies, for example strobes and number of targets within `APB`, the framework prohibits the construction of concise descriptions based on other value dependencies. Specifically, for ports whose direction is dependent on a mode of operation (`LocalLink`), and how to version protocols – `AXI` versions 1–4. Although we can write multiple different versions, a dependently typed construction should be able to capture these properties concisely, and without resorting to *copying and pasting* protocol specifications. We need to explore what other dependencies there are within a protocol specification, how prevalent these dependencies are, and how we can capture and reason about the relevant dependent properties.

The construction of λ_{AID} exposes the resource tracking of the type systems directly as resources are associated with variables. This does leads to a more verbose language such that for n signals there will be n variables, and n additional ports declared. This is not optimal. An alternative approach would be to embedd the resource tracking directly in our monad's type to remove the need for variables–cf. Swiestra's Hoare & Atkey's parameterised monads [37, 5]. However, our current formulation is more extensible allowing arbitrary new states to be added by indexing the type of new variables.

The type-level resource tracking in λ_{AID} also prohibits the creation of higher-order descriptions. The typing rule Lam requires a pure value, and sequencing using Let and Seq require that the *knowledge* contained in the environment is passed from the previous construct to the next. This is a limitation of the Hoare Monad used to sequencing expressions.

7.2 Modelling Hardware Interfaces

Many attempts at reasoning about hardware have centred on formalising hardware systems as a collection of digital circuts and capturing the behaviour of signals through the specified circuits. Ghica et al. exploited category theory to investigate connection of components [18, 17, 19]. EDSLs have been developed for Haskell such as Lava [20] and Cλash [35] that take other mathematical approaches to reasoning about hardware behaviour. ΠWare utilises dependent types to reason about hardware [15, 16]. Vijayaraghavan et al. [38] presents a complete formalisation of the behaviour of SoC designs, however, their approach does not look at the validation of interfaces against a specification, and concentrates on modelling the behaviour of components as a distributed system. Our framework complements existing work by providing guarantees about the physical structure of a component's ports.

Tooling such as Vivado IP Integrator [39] and Kactus2 [24] can automatically construct, and connect, components in a SoC architecture correctly. Such tooling is based on `IP-XACT` and vendor extensions. Examination of the Vivado toolchain reveals handwritten TCL scripts bespoke for the `AXI` family of protocols. Our work presents a specification agnostic framework for type-checking hardware interfaces against a richer specification than seen with `IP-XACT` solutions. We position our work as possible foundation for machine derivable code to develop richer integration and construction checks to that seen with IP Integrator and Kactus2.

Click is an untyped SoC design language for describing the routing of data [25]. McKechnie [30] developed a type-system for typing the interconnections found within Click specifications. Our work provides a natural extension to McKechnie's work and provides a means to type components in a Click design against external specifications.

7.3 Substructural Typing

The substructural type-system for λ_{AID} is based upon Hoare logic [5, 8]. Unfortunately, Hoare logics do not support the *frame rule*, a means to divide and share invariants in a composable manner. This results in λ_{AID} not being able to support higher-order descriptions.

Separation Logic is an advancement that does support the frame rule [34], and has been used to construct substructural type-systems for EDSLs [26], However, it is not clear how straightforward it would be to realise such a type-system for an EDSL within a dependently typed language.

There are other formal models upon which one can realise substructural type-systems for EDSLs, namely TypeStates [31], and Refinement Types from Hoare Types [8, 32]. All allow for reasoning about type-level resource usage protocols, however, how straightforward these models can be realised within a dependently typed language is not clear.

λ_{AID} was realised as an EDSL, perhaps realising it as a standalone Domain Specific Language (DSL) written in Idris might allow for Idris' rich type-system to better realise the substructural typing for λ_{AID}. Future work will be to investigate how to realise, and implement, the substructural typing for λ_{AID}.

7.4 Implementing Cordial

Cordial has been implemented within Idris. Any other dependently typed language that supports full-spectrum dependent types, such as Agda [33], would also be suitable host language.

Although Cordial uses dependent type theory and substructural typing, non-dependently type languages can also realise the framework. The ideas are transferable, but the implementation would not be as clean nor concise. Racket is a general purpose language that supports EDSL creation through fine-grained control over the language's type-system [14]. $F^\star$ is a general purpose language with value-dependent types [36]. Whereas Idris provides full-spectrum dependent types, $F^\star$ provides value-dependencies using refinement types. This provides a novel, alternate, environment in which to construct "value-dependently-typed" programs. How the approach behind Cordial is transferable to these languages is worth investigating.

8 Conclusion

We presented a framework (Cordial) to provide correct-by-construction guarantees over interface specifications in SoC designs. We have demonstrated use of the framework to model real world protocols, and noted limitations in the models expressiveness and future work to enrich said expressiveness. There are other areas for future work:

Checking Existing Systems

Our approach lends itself well to the *generation* of designs from model instances. We can easily extend our Idris implementation to generate stubbs for various HDLs. However, how do we evaluate existing interfaces? To do so, not only do we need to be able to extract interface model descriptions from existing HDL code, but also associate these model descriptions with abstract interface model descriptions. That is, we need to be able to infer from a component specification the concrete interfaces, and for those interfaces their abstract descriptions and which characterisation corresponds to the found interface. The problem of model inference is difficult as component interfaces are not always cleanly defined. Multiple interfaces for a component can be presented as a flat port group, sending ports can send to multiple recipients, and the ordering of ports does not necessarily reflect the ordering in the specification. Further, the names given to ports may not match the labels described in the specification. Developers, and code generation tools, have complete freedom in structuring their components interfaces. Further work will be to explore how to infer models from such "messy" SoC descriptions.

Enriching existing HDL

We have formalised Cordial in an existing general purpose language. A more interesting area for future work, and to increase adoption of the ideas mentioned, would be to develop extensions for various HDL such as SystemVerilog with the presented framework. A similar approach would be to extend existing design environments such as Vivado from Xilinx to incorporate our tooling and ideas.

Checking Behaviour

Our solution reasons about the *structural correctness* of SoC architectures. These provided guarantees are a design time check. Standards documents also describe a protocol's *behavioural correctness*. Our models do not capture a component's behaviour, a correctly connected component may show incorrect behaviour (as described in the specification) at run time. We saw this when modelling the `AXI` family of protocols. Cordial borrows notions of global and local projections from *Session Types*. We could also look to use Session Types to reason about hardware behaviour. While there have been attempts at extending Session Types to fit communication models similar to those found in hardware [27], none have been directly applied to checking hardware. Future work will be to explore how we can extend our model descriptions to capture the behaviour of a component's interface.

Modelling complete SoC architectures

SoC designs are about connecting components. A natural extension to our work would be to provide an orchestration language that uses θ_{COMP} to model components and their connections. Existing work has investigated verifying IP Core connections using static typing to ensure substructural properties of a SoC design hold – Section 7.2. Integrating the work of McKechnie [30] into the expressive type-system of our framework can serve as the basis for a more complete solution to SoC design. We leave this aspect of future work as an open problem.

References

1 Guillaume Allais, Robert Atkey, James Chapman, Conor McBride, and James McKinna. A type and scope safe universe of syntaxes with binding: their semantics and proofs. *PACMPL*, 2(ICFP):90:1–90:30, 2018. doi:10.1145/3236785.

2 Thorsten Altenkirch, Martin Hofmann, and Thomas Streicher. Categorical Reconstruction of a Reduction Free Normalization Proof. In David H. Pitt, David E. Rydeheard, and Peter T. Johnstone, editors, *Category Theory and Computer Science, 6th International Conference, CTCS '95, Cambridge, UK, August 7-11, 1995, Proceedings*, volume 953 of *Lecture Notes in Computer Science*, pages 182–199. Springer, 1995. doi:10.1007/3-540-60164-3_27.

3 ARM Limited. *AMBA APB Protocol*, ARM IHI 0024C edition, 2010.

4 ARM Limited. *AXI and ACE Protocol Specification*, ARM IHI 0022F.b edition, 2017.

5 Robert Atkey. Parameterised notions of computation. *J. Funct. Program.*, 19(3-4):335–376, 2009. doi:10.1017/S095679680900728X.

6 Robert Atkey. Syntax and Semantics of Quantitative Type Theory. In Anuj Dawar and Erich Grädel, editors, *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, pages 56–65. ACM, 2018. doi:10.1145/3209108.3209189.

7 Lennart Augustsson and Magnus Carlsson. An Exercise in Dependent Types: A Well-Typed Interpreter. In *In Workshop on Dependent Types in Programming, Gothenburg*, 1999.

8 Johannes Borgström, Juan Chen, and Nikhil Swamy. Verifying stateful programs with substructural state and hoare types. In Ranjit Jhala and Wouter Swierstra, editors, *Proceedings of the 5th ACM Workshop Programming Languages meets Program Verification, PLPV 2011, Austin, TX, USA, January 29, 2011*, pages 15–26. ACM, 2011. doi:10.1145/1929529.1929532.

9 Edwin Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *J. Funct. Program.*, 23(5):552–593, 2013. doi:10.1017/S095679681300018X.

10 Edwin Brady. Programming and reasoning with algebraic effects and dependent types. In Greg Morrisett and Tarmo Uustalu, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, pages 133–144. ACM, 2013. doi:10.1145/2500365.2500581.

11 Edwin Brady. Resource-Dependent Algebraic Effects. In Jurriaan Hage and Jay McCarthy, editors, *Trends in Functional Programming - 15th International Symposium, TFP 2014, Soesterberg, The Netherlands, May 26-28, 2014. Revised Selected Papers*, volume 8843 of *Lecture Notes in Computer Science*, pages 18–33. Springer, 2014. doi:10.1007/978-3-319-14675-1_2.

12 Edwin Brady. *Type-Driven Development with Idris.* Manning, 1st edition, 2016.

13 James Maitland Chapman. *Type checking and normalisation.* PhD thesis, School of Computer Science, University of Nottingham, July 2009. URL: http://eprints.nottingham.ac.uk/10824/.

14 Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay A. McCarthy, and Sam Tobin-Hochstadt. A programmable programming language. *Commun. ACM*, 61(3):62–71, 2018. doi:10.1145/3127323.

15 J Pizani Flor. π-Ware: An Embedded Hardware Description Language using Dependent Types. Masters, Department of Information and Computing Sciences, 2014. URL: https://dspace.library.uu.nl/bitstream/handle/1874/298576/.

16 João Paulo Pizani Flor, Wouter Swierstra, and Yorick Sijsling. Pi-Ware: Hardware Description and Verification in Agda. In Tarmo Uustalu, editor, *21st International Conference on Types for Proofs and Programs, TYPES 2015, May 18-21, 2015, Tallinn, Estonia*, volume 69 of *LIPIcs*, pages 9:1–9:27. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015. doi:10.4230/LIPIcs.TYPES.2015.9.

17 Dan R. Ghica. The Geometry of Synthesis - How to Make Hardware Out of Software. In Jeremy Gibbons and Pablo Nogueira, editors, *Mathematics of Program Construction - 11th International Conference, MPC 2012, Madrid, Spain, June 25-27, 2012. Proceedings*, volume 7342 of *Lecture Notes in Computer Science*, pages 23–24. Springer, 2012. doi:10.1007/978-3-642-31113-0_3.

18 Dan R. Ghica, Achim Jung, and Aliaume Lopez. Diagrammatic Semantics for Digital Circuits. In Valentin Goranko and Mads Dam, editors, *26th EACSL Annual Conference on Computer Science Logic, CSL 2017, August 20-24, 2017, Stockholm, Sweden*, volume 82 of *LIPIcs*, pages 24:1–24:16. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017. doi:10.4230/LIPIcs.CSL.2017.24.

19 Dan R. Ghica, Alex I. Smith, and Satnam Singh. Geometry of synthesis iv: compiling affine recursion into static hardware. In Manuel M. T. Chakravarty, Zhenjiang Hu, and Olivier Danvy, editors, *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*, pages 221–233. ACM, 2011. doi:10.1145/2034773.2034805.

20 Andy Gill, Tristan Bull, Garrin Kimmell, Erik Perrins, Ed Komp, and Brett Werling. Introducing Kansas Lava. In Marco T. Morazán and Sven-Bodo Scholz, editors, *Implementation and Application of Functional Languages - 21st International Symposium, IFL 2009, South Orange, NJ, USA, September 23-25, 2009, Revised Selected Papers*, volume 6041 of *Lecture Notes in Computer Science*, pages 18–35. Springer, 2009. doi:10.1007/978-3-642-16478-1_2.

21 John Hatcliff and Olivier Danvy. A Generic Account of Continuation-Passing Styles. In Hans-Juergen Boehm, Bernard Lang, and Daniel M. Yellin, editors, *Conference Record of POPL'94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland,*

Oregon, USA, January 17-21, 1994, pages 458–471. ACM Press, 1994. doi:10.1145/174675.178053.

22 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In George C. Necula and Philip Wadler, editors, *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, pages 273–284. ACM, 2008. doi:10.1145/1328438.1328472.

23 IEEE. *IEEE Standard for IP-XACT, Standard Structure for Packaging, Integrating, and Reusing IP within Tool Flows*, ieee std 1685-2014 edition, September 2014. doi:10.1109/IEEESTD.2014.6898803.

24 Antti Kamppi, Lauri Matilainen, Joni-Matti Määttä, Erno Salminen, Timo D. Hämäläinen, and Marko Hännikäinen. Kactus2: Environment for Embedded Product Development Using IP-XACT and MCAPI. In *14th Euromicro Conference on Digital System Design, Architectures, Methods and Tools, DSD 2011, August 31 - September 2, 2011, Oulu, Finland*, pages 262–265. IEEE Computer Society, 2011. doi:10.1109/DSD.2011.36.

25 Eddie Kohler, Robert Tappan Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The click modular router. *ACM Trans. Comput. Syst.*, 18(3):263–297, 2000. doi:10.1145/354871.354874.

26 Neelakantan R. Krishnaswami, Aaron Turon, Derek Dreyer, and Deepak Garg. Superficially substructural types. In Peter Thiemann and Robby Bruce Findler, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP'12, Copenhagen, Denmark, September 9-15, 2012*, pages 41–54. ACM, 2012. doi:10.1145/2364527.2364536.

27 Julien Lange, Nicholas Ng, Bernardo Toninho, and Nobuko Yoshida. A static verification framework for message passing in Go using behavioural types. In Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman, editors, *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 1137–1148. ACM, 2018. doi:10.1145/3180155.3180157.

28 Per Martin-Löf and Giovanni Sambin. *Intuitionistic Type Theory.* Bibliopolis, 1984.

29 Conor McBride. I Got Plenty o' Nuttin'. In Sam Lindley, Conor McBride, Philip W. Trinder, and Donald Sannella, editors, *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, volume 9600 of *Lecture Notes in Computer Science*, pages 207–233. Springer, 2016. doi:10.1007/978-3-319-30936-1_12.

30 Paul Edward McKechnie. *Validation and verification of the interconnection of hardware intellectual property blocks for FPGA-based packet processing systems.* PhD thesis, University of Glasgow, 2010. URL: http://theses.gla.ac.uk/1879/.

31 Filipe Militão, Jonathan Aldrich, and Luís Caires. Substructural typestates. In Nils Anders Danielsson and Bart Jacobs, editors, *Proceedings of the 2014 ACM SIGPLAN Workshop on Programming Languages meets Program Verification, PLPV 2014, January 21, 2014, San Diego, California, USA, Co-located with POPL '14*, pages 15–26. ACM, 2014. doi:10.1145/2541568.2541574.

32 Aleksandar Nanevski, J. Gregory Morrisett, and Lars Birkedal. Hoare type theory, polymorphism and separation. *J. Funct. Program.*, 18(5-6):865–911, 2008. doi:10.1017/S0956796808006953.

33 Ulf Norell. Dependently typed programming in Agda. In Andrew Kennedy and Amal Ahmed, editors, *Proceedings of TLDI'09: 2009 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Savannah, GA, USA, January 24, 2009*, pages 1–2. ACM, 2009. doi:10.1145/1481861.1481862.

34 John C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*, pages 55–74. IEEE Computer Society, 2002. doi:10.1109/LICS.2002.1029817.

35 Gerard J. M. Smit, Jan Kuper, and Christiaan P. R. Baaij. A mathematical approach towards hardware design. In Peter M. Athanas, Jürgen Becker, Jürgen Teich, and Ingrid

Verbauwhede, editors, *Dynamically Reconfigurable Architectures, 11.07. - 16.07.2010*, volume 10281 of *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany, 2010. URL: `http://drops.dagstuhl.de/opus/volltexte/2010/2840/`.

36 Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. Secure distributed programming with value-dependent types. *J. Funct. Program.*, 23(4):402–451, 2013. `doi:10.1017/S0956796813000142`.

37 Wouter Swierstra. A Hoare Logic for the State Monad. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, volume 5674 of *Lecture Notes in Computer Science*, pages 440–451. Springer, 2009. `doi:10.1007/978-3-642-03359-9_30`.

38 Muralidaran Vijayaraghavan, Adam Chlipala, Arvind, and Nirav Dave. Modular Deductive Verification of Multiprocessor Hardware Designs. In Daniel Kroening and Corina S. Pasareanu, editors, *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II*, volume 9207 of *Lecture Notes in Computer Science*, pages 109–127. Springer, 2015. `doi:10.1007/978-3-319-21668-3_7`.

39 Product page for Vivado IP Integrator by Xilinx. Online, 2019. URL: `https://www.xilinx.com/products/design-tools/vivado/integration.html`.

40 David Walker. *Advanced Topic in Types and Programming Languages*, chapter Substructural Type Systems, pages 3–43. The MIT Press, 2004.

41 Xilinx. *LocalLink Interface Specification*, SP006 (v2.0) edition, 2005.

On Satisfiability of Nominal Subtyping with Variance

Aleksandr Misonizhnik

JetBrains Research, Saint Petersburg State University, Russia
misonijnik@gmail.com

Dmitry Mordvinov

JetBrains Research, Saint Petersburg State University, Russia
dmitry.mordvinov@jetbrains.com

Abstract

Nominal type systems with variance, the core of the subtyping relation in object-oriented programming languages like Java, C# and Scala, have been extensively studied by Kennedy and Pierce: they have shown the undecidability of the subtyping between ground types and proposed the decidable fragments of such type systems. However, modular verification of object-oriented code may require reasoning about the relations of open types. In this paper, we formalize and investigate the satisfiability problem for nominal subtyping with variance. We define the problem in the context of first-order logic. We show that although the non-expansive ground nominal subtyping with variance is decidable, its satisfiability problem is undecidable. Our proof uses a remarkably small fragment of the type system. In fact, we demonstrate that even for the non-expansive class tables with only nullary and unary covariant and invariant type constructors, the satisfiability of quantifier-free conjunctions of positive subtyping atoms is undecidable. We discuss this result in detail, as well as show one decidable fragment and a scheme for obtaining other decidable fragments.

2012 ACM Subject Classification Theory of computation → Type theory; Theory of computation → Logic and verification; Software and its engineering → Object oriented languages; Software and its engineering → Automated static analysis; Software and its engineering → Polymorphism; Software and its engineering → Inheritance

Keywords and phrases nominal type systems, structural subtyping, first-order logic, decidability, software verification

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2019.7

Acknowledgements We thank Sophia Drossopoulou, Dmitry Boulytchev and anonymous reviewers for their insight and comments that significantly improved the manuscript.

1 Introduction

Although object-oriented languages like Java, C# or Scala are ubiquitous in modern programming, the investigation of their type systems is still in progress. For example, Java type checking has only recently been shown to be undecidable [7]. One important feature of such languages is that types can appear at runtime and influence program execution (in contrast to the ML programming language family, Haskell, etc., in which type information is erased during compilation). For example, consider the following snippet:

```
IDictionary<TKey, TValue> MakeCache<TKey, TValue>()
{
    if (typeof(TKey) == typeof(int))
        return new SortedDictionary<TKey, TValue>();
    if (typeof(TKey) == typeof(string))
        return new Dictionary<TKey, TValue>();
    throw new InvalidOperationException();
}
```

33rd European Conference on Object-Oriented Programming (ECOOP 2019).
Editor: Alastair F. Donaldson; Article No. 7; pp. 7:1–7:20
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

The runtime behaviour of `MakeCache` depends on the value of the formal type parameter `TKey`: the method returns a fresh instance of `SortedDictionary` indexed by integer keys, Dictionary for string keys, or throws an exception otherwise. But what if we would like to *statically* check that `MakeCache` is used correctly, i.e. it does not throw an exception? Unfortunately, the type parameter constraint system of .NET does not allow to specify this restriction, as it does not implement disjunctive constraints.

Our study is motivated by the problem of verification of .NET programs. Modern deductive software verifiers are capable of checking non-trivial properties of programs by proving that under certain *preconditions*, executing a function guarantees certain *postconditions*. For instance, for the example above, the correctness property could be specified as a logical precondition for the method. In Spec#-style [1], this could be expressed as

```
IDictionary<TKey, TValue> MakeCache<TKey, TValue>()
    requires typeof(TKey) == typeof(int) || typeof(TKey) == typeof(string)
{
    ...
}
```

In more complex cases, arbitrary boolean combinations (including negation) of subtyping constraints or even quantified specification can be useful. Unfortunately, neither compilers nor modern verifiers (including Spec#) are capable of statically checking such properties because their *assertion language* cannot express properties of types.

Consider another snippet:

```
1  interface ICloneable<out T> { T Clone(); }
2  class Base { }
3  sealed class Derived : Base, ICloneable<Derived>
4  {
5      public Derived Clone() { return new Derived(); }
6  }
7  void F<T>(Base arg1, Derived arg2)
8  {
9      if (arg2 is ICloneable<T> && arg1 is T)
10     {
11         var clone = ((Derived) arg1).Clone();
12         ...
13     }
14 }
```

Using our fictitious extension of Spec# specification language, the specification of the function F would include the following clause:

$$\begin{gathered} \mathit{requires}\ \texttt{Derived} <: \texttt{ICloneable<T>} \land \mathit{typeof}(arg1) <: \texttt{Base}\land \\ \mathit{typeof}(arg1) <: \texttt{T} \land \mathit{typeof}(arg1) <: \texttt{Derived} \end{gathered}$$

One might think that the last conjunct, ie $\mathit{typeof}(arg1) <: \mathit{Derived}$, can be omitted, and that the type cast expression (line 11) never fails: as `Derived` is sealed, the actual type of `arg2` can only be `Derived`. In contrast, the type of `arg1` can be `Base` or a subtype of `Base` type; as `Derived` implements only `ICloneable<Derived>`, `T` may only be `Derived`, therefore, if line 11 is reachable, then `arg1` is `Derived`. However, this reasoning is wrong: as `ICloneable` is a covariant constructor and `Base` is a supertype of `Derived`, `ICloneable<Base>` is a supertype of `Derived`, and line 11 can be reached with `T` = `Base`. Is it possible to determine the satisfiability of such violations automatically?

It would be legitimate to omit the last conjunct, ie $typeof(arg1) <: \texttt{Derived}$, and also omit the cast on line 11, if we knew that

$$\forall \texttt{T}, \texttt{T'}. \texttt{Derived} <: \texttt{ICloneable<T>} \wedge \texttt{T'} <: \texttt{Base} \wedge \texttt{T'} <: \texttt{T} \Rightarrow \texttt{T'} <: \texttt{Derived}$$

Or, equivalently, if we could prove the unsatisfiability of the following assertion:

$$\phi \stackrel{\text{def}}{=} \exists \texttt{T}, \texttt{T'}. \texttt{Derived} <: \texttt{ICloneable<T>} \wedge \texttt{T'} <: \texttt{Base} \wedge \texttt{T'} <: \texttt{T} \wedge \texttt{T'} \not<: \texttt{Derived}$$

In the particular case, ϕ is satisfiable. Namely, take $\texttt{T} = \texttt{Base} = \texttt{T}'$, and the optimizations proposed above would be unsound.

Our examples have demonstrated the relevance of subtype satisfiability in program specification as well as for code optimization. The next question is the design of decision procedures for such questions. However, it turns out that this question in undecidable!

In section 2, we formalize nominal subtyping with variance following definitions and propositions from [8]. We focus our attention on *non-expansive inheritance* fragment [19], which was shown to be decidable [8] and has been adopted in the .NET Framework [4].

In section 3, we formalize and investigate the *first-order satisfiability problem* for nominal subtyping with variance; to the best of our knowledge, this is the first attempt at its detailed examination. Unlike the subtyping problem, which answers the question “Is one type a subtype of another type?”, the satisfiability problem answers the question “Are there types that meet the required constraints?”. In fact, satisfiability involves the subtyping problem, in the case where the constraints do not contain type variables. This means that the decidability of the subtyping problem does not imply the decidability of the satisfiability problem. We present a number of quantifier-free first-order formulas, demonstrating that the satisfiability problem is tricky even for non-expansive inheritance.

In section 4, we reinforce this by proving the undecidability of this problem, which is the primary contribution of the paper. Our proof uses a remarkably small fragment of the type system; in fact, we show that the subtyping satisfiability problem is undecidable for *non-expansive* class tables (1) without contravariant constructors, (2) with only nullary and unary constructors, (3) for quantifier-free conjunctions of subtyping atoms without negation.

Afterwards, in section 5, we demonstrate one practical decidable fragment which we call *semiground*, and prove its decidability. Using the intuitions from the proof, we provide a scheme to obtain other decidable fragments.

Our results may give rise to the construction of an effective decision procedure for the quantifier-free case. We formulate the problem in terms of satisfiability modulo theory, aiming at the implementation of its decision procedure in SMT-solvers [2, 3]. The support of SMT-reasoning for nominal type theory is useful for software verification tools, static analysers and generation techniques of automated tests, and exploits and patches for object-oriented languages. The first-order theory of nominal types may be used for type specifications in the assertion languages of deductive verifiers, while its decision procedure may be used for solving path conditions of different program branches. It may also be useful in compilers: for example, in improving dead code elimination techniques by proving the unsatisfiability of the path condition for a certain code fragment.

2 The type system[1]

In this section, we formalize nominal subtyping with variance. Types can either be type variables, denoted by lowercase letters, or constructed types C<$\overline{T}$>, where C is an n-ary type constructor and $\overline{T}$ is a vector of arguments of length n. We omit angle brackets if a type constructor is unary: for example, we write $ABCx$ instead of A<B<C<x>>>. C^rT denotes the type $C \dots CT$, where C occurs r times. We refer to such types as *chains* of r type constructors C ending with T. *Ground* types are types that contain no variables. *Open* types are types that are not ground.

The subtyping relation of nominal type systems with variance is defined via an explicit specification of the names of supertypes and *variances* of type parameters. Such specifications are usually expressed as class tables.

▶ **Definition 2.1.** *A* class table *is a finite set of entries of the form*

$$C\texttt{<}\overline{vx}\texttt{>} \ \texttt{<::}\ T_1, \dots, T_n$$

Each entry contains a unique declaration of a type constructor and a finite list of constructed types, which are nominal supertypes for all types constructed by this type constructor. The left-hand side of an entry contains the name for constructor C and its formal type parameters x_i with variances *v_i. v_i may be either $\circ$ (invariant) or $+$ (covariant) or $-$ (contravariant). The right-hand side contains a finite list of types T_i obtained from constructors declared in other entries, constructor C, and parameters x_i. To simplify the notation, we omit $\circ$ in class table declarations.*

System.Object	<::	
System.ValueType	<::	System.Object
IEnumerable	<::	System.Object
IEnumerable<$+x$>	<::	IEnumerable
ICollection	<::	IEnumerable
ICollection<x>	<::	IEnumerable<x>
Pair<x, y>	<::	System.ValueType
IDictionary	<::	ICollection
IDictionary<x, y>	<::	ICollection<Pair<x, y>>
Dictionary<x, y>	<::	IDictionary<x, y> IDictionary

```
interface IEnumerable {}
interface IEnumerable<out x>:
    IEnumerable{}
interface ICollection:
    IEnumerable {}
interface ICollection<x>:
    IEnumerable<x> {}
struct Pair<x, y> {}
interface IDictionary:
    ICollection {}
interface IDictionary<x, y>:
    ICollection<Pair<x, y>> {}
class Dictionary<x, y>:
    IDictionary<x, y>,
    IDictionary {}
```

Listing 1 The declaration of the class Dictionary and its class table.

▶ **Example 2.2.** Listing 1 demonstrates a simplified fragment of the class table for the Dictionary<x, y> standard container in .NET.

[1] In this section, we follow definitions and propositions from [8].

The i-th formal type parameter of a type constructor C and its variance are denoted by $C\#i$ and $var(C\#i)$ correspondingly: $C\#i \overset{\text{def}}{=} x_i$ and $var(C\#i) \overset{\text{def}}{=} v_i$. For example, $\texttt{IEnumerable}\#1 = x$ and $var(\texttt{IEnumerable}\#1) = +$.

▶ **Definition 2.3.** *A* substitution *is a total mapping from type variables to types, which is an identity everywhere except for a finite set of variables which are mapped into constructed types. The* domain *of a substitution subst is a set of variables mapped to types, and the* range *is the image of the domain. We write substitutions as*

$$[x_1 \mapsto T_1;\ \ldots,\ x_n \mapsto T_n]\ \textit{and}\ [\overline{x} \mapsto \overline{T}],$$

where $x_1, \ldots, x_n$ are type variables from the domain of substitution and $T_1, \ldots, T_n$ are their images. We write the application of substitution $[\overline{x} \mapsto \overline{T}]$ to type T as $[\overline{x} \mapsto \overline{U}]T$. If the domain of a substitution consists of one type variable x, we omit the brackets:

$$x \mapsto T$$

We use $<::$ not only as a class table separator, but also to denote the binary relation of nominal subtyping. If a class table has an entry $C\texttt{<}\overline{x}\texttt{>} <:: T_i$, then $C\texttt{<}\overline{U}\texttt{>} <:: [\overline{x} \mapsto \overline{U}]T_i$. We write the transitive closure of $<::$ as $<::^+$.

We require class tables to define only acyclic $<::^+$ relations and to be well-formed with respect to the variance of formal type parameters, i.e. variant type parameters should appear only in positions of the same polarity. Furthermore, we require that the supertypes do not overlap: if $C\texttt{<}\overline{x}\texttt{>} <:: T$ and $C\texttt{<}\overline{x}\texttt{>} <:: U$, then for all $\overline{V}$ if $[\overline{x} \mapsto \overline{V}]T = [\overline{x} \mapsto \overline{V}]U$, then $T = U$.

Finally, we can define the subtyping relation.

▶ **Definition 2.4.** *The ground subtyping relation $<:$ is defined by the set of the following rules:*

$$\frac{T <: U}{T <:_+ U} \qquad \frac{}{T <:_\circ T} \qquad \frac{U <: T}{T <:_- U}$$

$$\textit{(Var)}\ \frac{\textit{for each}\, i \qquad T_i <:_{var(C\#i)} U_i}{C\texttt{<}\overline{T}\texttt{>} <: C\texttt{<}\overline{U}\texttt{>}}$$

$$\textit{(Super)}\ \frac{C\texttt{<}\overline{x}\texttt{>} <:: V \qquad [\overline{x} \mapsto \overline{T}]V <: D\texttt{<}\overline{U}\texttt{>}}{C\texttt{<}\overline{T}\texttt{>} <: D\texttt{<}\overline{U}\texttt{>}}\ C \neq D$$

Due to the multiple instantiation inheritance, the Super rule can be applied non-deterministically.

▶ **Example 2.5.** The following sequence of rules should be applied to deduce that $\texttt{Dictionary<}T\texttt{, }U\texttt{>}$ is a subtype of `IEnumerable<System.Object>`:

$$\begin{array}{lll}
 & \texttt{Dictionary<}T\texttt{, }U\texttt{>} <: \texttt{IEnumerable<System.Object>} & \\
\longrightarrow & \texttt{IDictionary<}T\texttt{, }U\texttt{>} <: \texttt{IEnumerable<System.Object>} & \textit{by}\ \text{Super} \\
\longrightarrow & \texttt{ICollection<Pair<}T\texttt{, }U\texttt{>>} <: \texttt{IEnumerable<System.Object>} & \textit{by}\ \text{Super} \\
\longrightarrow & \texttt{IEnumerable<Pair<}T\texttt{, }U\texttt{>>} <: \texttt{IEnumerable<System.Object>} & \textit{by}\ \text{Super} \\
\longrightarrow & \texttt{Pair<}T\texttt{, }U\texttt{>} <: \texttt{System.Object} & \textit{by}\ \text{Var} \\
\longrightarrow & \texttt{System.ValueType} <: \texttt{System.Object} & \textit{by}\ \text{Super} \\
\longrightarrow & \texttt{System.Object} <: \texttt{System.Object} & \textit{by}\ \text{Super} \\
\longrightarrow & & \textit{by}\ \text{Var}
\end{array}$$

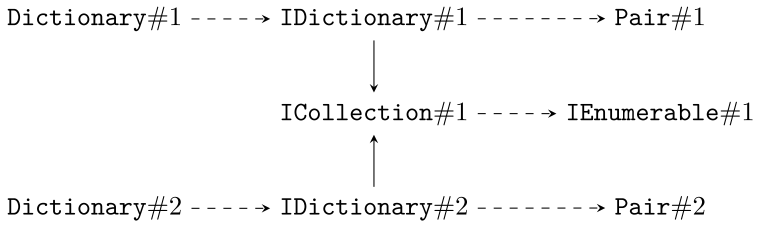


Figure 1 Type parameter dependency graph for Listing 1.

Ground subtyping is a partial order on a set of ground types. The ground subtyping relation has been shown to be undecidable in [8]. In the following, we introduce a notion of *non-expansive inheritance.*

▶ **Definition 2.6.** *A* type parameter dependency graph *is a directed graph with vertices that correspond to formal type parameters and two kinds of edges: for each class table entry* C<$\overline{x}$> $<::$ T *and for each subterm* D<$\overline{U}$> *of* T*,*

- *if* $U_j = x_i$*, then there is a* non-expansive *edge from* $C\#i$ *to* $D\#j$ *(depicted via a dotted arrow);*
- *if* x_i *is a proper subterm of* U_j*, then there is an* expansive *edge from* $C\#i$ *to* $D\#j$ *(depicted via a solid arrow).*

For instance, in Example 2.2, `IDictionary<x,y>` $<::$ `ICollection<Pair<x,y>>` introduces a non-expansive edge from `IDictonary#1` to `Pair#1` and an expansive edge to `ICollection#1`. The complete type parameter dependency graph is shown in Figure 1.

▶ **Definition 2.7.** *A class table is* expansive *if its type parameter dependency graph has a cycle with at least one expansive edge.*

▶ **Example 2.8.** The class table Listing 1 is non-expansive, as its type parameter dependency graph Figure 1 does not contain cycles.

▶ **Proposition 2.9.** *The non-expansive ground subtyping relation is decidable.*

▶ **Proposition 2.10.** *Ground subtyping is decidable if a class table has no contravariant constructors.*

Both results have been shown in [8].

3 The `SUBTYPE-SAT` problem

In this section, we formalize the subtype satisfiability problem and show some interesting examples.

In what follows, fix a class table CT. Let $\mathcal{C}$ be a set of constructors in CT. Let $\Sigma = (\mathcal{C}, \{<:\})$ be a first-order signature with equality. Function symbols are identified with constructors in CT. For convenience, the applications of a function symbol C to arguments $\overline{U}$ are still written as C<$\overline{U}$>, or just CU in the unary case. $<:$ is a binary predicate symbol written in infix style. For convenience, $\neg(T <: U)$ and $\neg(T = U)$ are written as $T \not<: U$ and $T \neq U$.

Let $I_{<:}$ be a Σ-structure with the domain $|I_{<:}|$ of all ground types defined by CT, interpreting $<:$ as the subtyping relation from Definition 2.4. Let $\mathcal{T}^{CT}_{<:}$ be a complete first-order Σ-theory of structure $I_{<:}$, i.e. the set of all first-order Σ-sentences which are satisfied by $I_{<:}$ (we assume a usual definition of satisfaction of ϕ by I, denoted $I \vDash \phi$). Given a Σ-sentence ϕ, we say that ϕ is *satisfiable modulo* $\mathcal{T}^{CT}_{<:}$, iff $I \vDash \phi$.

Let $\mathcal{V}$ be a countable set of variables. An *assignment of free variables* is any mapping $v : \mathcal{V} \to |I_{<:}|$ of variables to ground types. Note that free variable assignments are substitutions with a ground range. A formula with free variables ϕ is called satisfiable (valid, unsatisfiable) if $I, v \vDash \phi$ for some (any, no) free variable assignment v. We abbreviate the satisfiability in (I, v) and validity of ϕ with $v \vDash^{CT}_{<:} \phi$ and $\vDash^{CT}_{<:} \phi$ correspondingly.

▶ **Problem 3.1** (`SUBTYPE-SAT` problem)**.** *Given a class table CT and a formula ϕ over Σ, find such a free variable assignment v that $v \vDash^{CT}_{<:} \phi$ or prove its absence.*

We aim to show that although the ground subtyping relation is decidable for both non-expansive class tables and class tables without contravariant constructors, the `SUBTYPE-SAT` problem is undecidable even with both restrictions. We begin with a number of examples demonstrating the complexity of this problem.

▶ **Example 3.2.** Consider a class table

$$\begin{array}{lll} J\texttt{<}+x\texttt{>} & <:: & \\ C & <:: & JC \end{array}$$

and a formula

$$\phi \overset{\text{def}}{=} C <: x \wedge C <: y \wedge x \not<: y \wedge y \not<: x$$

Is ϕ satisfiable? Let us consider various possible candidates for the assignment v. Let $v(x) = C$ and $v(y) = C$. In this case, the atom $x \not<: y$ is falsified:

$$v(x) \not<: v(y) = C \not<: C \Leftrightarrow \perp$$

Let $v(x) = C$ and $v(y) = Jy'$ for some y'. Then

$$I(\phi) = C <: C \wedge C <: Jy' \wedge C \not<: Jy' \wedge Jy' \not<: C \Leftrightarrow \perp$$

The case $v(x) = Jx'$ and $v(y) = C$ is symmetrical. The last case is $I(x) = Jx'$ and $I(y) = Jy'$:

$$\begin{array}{l} v(\phi) = C <: Jx' \wedge C <: Jy' \wedge Jx' \not<: Jy' \wedge Jy' \not<: Jx' \Leftrightarrow \\ JC <: Jx' \wedge JC <: Jy' \wedge Jx' \not<: Jy' \wedge Jy' \not<: Jx' \Leftrightarrow \\ C <: x' \wedge C <: y' \wedge x' \not<: y' \wedge y' \not<: x' \end{array}$$

Note that $v(\phi)$ is exactly ϕ up to a renaming of variables. It means that every candidate variable substitution either falsifies the formula, or results in a formula to which the same reasoning applies. As an infinite chain of J is not a valid type, ϕ is unsatisfiable.

The unsatisfiability of ϕ can be intuitively explained in the following way. The satisfiability of ϕ would mean that C has two incomparable supertypes. A set of supertypes of C is exactly $\{J^n C \mid n \geq 0\}$. But for all n, m, $J^n C$ and $J^m C$ are comparable: $n \leq m$ iff $J^n C <: J^m C$.

▶ **Example 3.3.** Consider another class table

$$\begin{array}{lll} E & <:: & \\ J\texttt{<}+x\texttt{>} & <:: & \\ A_1 & <:: & J^{n_1}A_1,\ J^{n_1}E \\ & \vdots & \\ A_m & <:: & J^{n_m}A_m,\ J^{n_m}E, \end{array}$$

where m, $n_i \geq 1$, and the formula

$$\phi \overset{\text{def}}{=} \bigwedge_{1\leq i\leq m} A_i <: x$$

This formula is satisfied only by v such that

$$v(x) = J^{k\cdot lcm(n_1,\ldots,n_m)}E,$$

where $k \geq 1$ and $lcm(n_1, \ldots, n_m)$ is a least common multiple of $n_1, \ldots, n_m$.

▶ **Example 3.4.** If we replace the class table entry for E in Example 3.3 with

$$E \quad <:: \quad J^{n_1\cdot\ldots\cdot n_m}E$$

then the formula

$$\phi' \overset{\text{def}}{=} \phi \wedge E \not<: x$$

has a model if and only if the numbers n_1, …, n_m are not coprime.

▶ **Example 3.5.** Fix a class table CT and a finite partially ordered set $(P, \leq_P)$. Consider the formula

$$\phi \overset{\text{def}}{=} \bigwedge_{\substack{x,y\in P,\\ x\leq_P y}} x <: y \wedge \bigwedge_{\substack{x,y\in P,\\ x\not\leq_P y}} x \not<: y$$

ϕ has a model if and only if there exists an order-embedding map from P to the set of ground types defined by CT (partially ordered by ground subtyping relation).

▶ **Proposition 3.6.** *SUBTYPE-SAT is semidecidable.*

Proof. There is an algorithm that, given ϕ, enumerates all possible ground substitutions v of variables of ϕ and checks $v \vDash_{<:}^{CT} \phi$. Proposition 2.9 guarantees that if ϕ is satisfiable modulo $\mathcal{T}_{<:}^{CT}$, then this algorithm eventually terminates. ◀

In the following section, we show that the set of ground substitution v such that $v \nvDash_{<:}^{CT} \phi$ is not recursively enumerable.

4 SUBTYPE-SAT is undecidable

$$
\begin{array}{lll}
A_1\texttt{<}+x\texttt{>} & <:: & \\
 & \vdots & \\
A_m\texttt{<}+x\texttt{>} & <:: & \\
R\texttt{<}+x\texttt{>} & <:: & \\
S\texttt{<}+x\texttt{>} & <:: & \\
R_0 & <:: & RR_0,\ E \\
S_0 & <:: & SS_0,\ E \\
E & <:: & \\
U_1\texttt{<}+x\texttt{>} & <:: & \overline{A_{U_1}}x,\ W_1x,\ Sx,\ Rx \\
 & \vdots & \\
U_n\texttt{<}+x\texttt{>} & <:: & \overline{A_{U_n}}x,\ W_nx,\ Sx,\ Rx \\
V_1\texttt{<}+y\texttt{>} & <:: & \overline{A_{V_1}}y,\ W_1y,\ Sy,\ Ry \\
 & \vdots & \\
V_n\texttt{<}+y\texttt{>} & <:: & \overline{A_{V_n}}y,\ W_ny,\ Sy,\ Ry
\end{array}
$$

$$
\begin{array}{lll}
W_1\texttt{<}+x\texttt{>} & <:: & \\
 & \vdots & \\
W_n\texttt{<}+x\texttt{>} & <:: & \\
G & <:: & U_1G,\ldots,U_nG,\ E \\
H & <:: & V_1H,\ldots,V_nH,\ E \\
P & <:: & U_1P,\ldots,U_nP,\ E \\
Q & <:: & V_1Q,\ldots,V_nQ,\ E \\
W & <:: & W_1W,\ldots,W_nW,\ E \\
D & <:: & \overline{A_{U_1}}D,\ldots,\overline{A_{U_n}}D,\ E
\end{array}
$$

Listing 2 Class table `PCP-CT`.

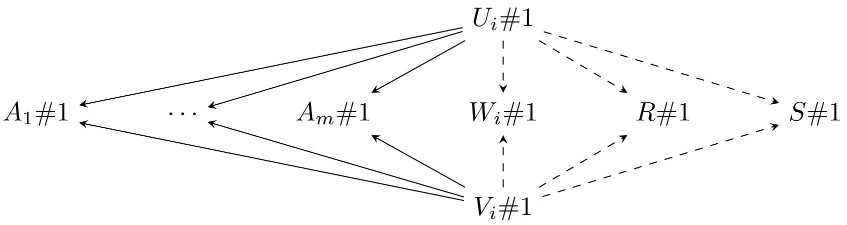


Figure 2 Type parameter dependency graph for `PCP-CT`.

We prove the undecidability of `SUBTYPE-SAT` via a reduction from the Post Correspondence Problem.

The Post Correspondence Problem

Let $\{(\overline{A_{U_1}}, \overline{A_{V_1}}), \ldots, (\overline{A_{U_n}}, \overline{A_{V_n}})\}$ be a set of pairs of non-empty words over a finite alphabet $\{\, A_1, \ldots, A_m \,\}$. The Post Correspondence Problem (PCP) is to determine whether or not there exists a sequence of indices $i_1, \ldots, i_r$ such that $\overline{A_{U_{i_1}}} \ldots \overline{A_{U_{i_r}}} = \overline{A_{V_{i_1}}} \ldots \overline{A_{V_{i_r}}}$.

It is a well-known fact that PCP is undecidable [13].

We use a class table from Listing 2 in our reduction. Note that it is non-expansive as its type parameter dependency graph (see Figure 2) has no cycles, and it has no contravariant constructors.

Consider the `SUBTYPE-SAT` problem for a formula ψ with type parameters x, y, z, q, p, t and class table `PCP-CT`, where ψ is defined as follows:

$$\begin{aligned}
&\phi_0 \overset{\text{def}}{=} R_0 <: p \wedge S_0 <: q \wedge W <: z\\
&\phi_1 \overset{\text{def}}{=} G <: x \wedge P <: x \wedge x <: p \wedge x <: q \wedge x <: z \wedge \phi_0\\
&\phi_2 \overset{\text{def}}{=} H <: y \wedge Q <: y \wedge y <: p \wedge y <: q \wedge y <: z \wedge \phi_0\\
&\psi \overset{\text{def}}{=} D <: t \wedge x <: t \wedge y <: t \wedge t \not<: E \wedge \phi_1 \wedge \phi_2
\end{aligned}$$

The main idea of this reduction is to represent the words in $\{ A_1, \ldots, A_m \}^+$ as chains of covariant constructors terminating with E. For example, words $\overline{A_{U_i}}$ and $\overline{A_{V_j}}$ are encoded as $\overline{A_{U_i}}E$ and $\overline{A_{V_j}}E$. The enumeration of PCP solutions is encoded into the `PCP-CT` and ψ. We demonstrate a non-deterministic process, consistently refining the type variables of ψ by replacing them with a type constructor applied to fresh variables, and then simplifying the new formula.

▶ **Definition 4.1.** *A* ground *substitution is a substitution with only ground types in its range. An* elementary *substitution is a substitution whose range contains only constructed types with type variables as their arguments. A substitution is* complete *for a formula f, if its domain is exactly all type variables of f.*

Substitutions may be *composed*. The composition of $u = [\, x_1 \mapsto a_1; \ldots; x_n \mapsto a_n \,]$ and $v = [\, y_1 \mapsto b_1; \ldots; y_m \mapsto b_m \,]$ is obtained by removing from the substitution $[\, x_1 \mapsto va_1; \ldots; x_n \mapsto va_n; y_1 \mapsto b_1; \ldots; y_m \mapsto b_m \,]$ those pairs $y_i \mapsto b_i$ for which $y_i \in \{x_1, \ldots, x_k\}$. For instance,

$$x \mapsto Cx' \odot x' \mapsto Dx'' = x \mapsto CDx''$$

We also define a composition of *substitution sets*:

$$\{\, \mathit{subst}_1^{\mathit{left}}; \ldots; \mathit{subst}_n^{\mathit{left}} \,\} \odot \{\, \mathit{subst}_1^{\mathit{right}}; \ldots; \mathit{subst}_k^{\mathit{right}} \,\} \overset{\text{def}}{=} \bigcup_{\substack{1 \le i \le n\\ 1 \le j \le k}} \{\, \mathit{subst}_i^{\mathit{left}} \odot \mathit{subst}_j^{\mathit{right}} \,\}$$

Note that as ψ is a conjunction of atoms, its satisfiability implies the satisfiability of its arbitrary subformula. Now let us consider complete substitutions that do not falsify the subformula ϕ_0.

▶ **Lemma 4.2.** *For the formula*

$$\phi_0 \overset{\text{def}}{=} R_0 <: p \wedge S_0 <: q \wedge W <: z,$$

a substitution does not falsify ϕ_0 iff it can be represented as a composition of substitutions

$$p \mapsto Rp,\ p \mapsto E,\ p \mapsto R_0,\ q \mapsto Sq,\ q \mapsto E,\ q \mapsto S_0,\ z \mapsto W_i z,\ z \mapsto E,\ z \mapsto W$$

with $1 \le i \le n$.

Proof. The type constructor R_0 only has nominal supertypes with head constructors R_0, R and E. Hence, after an application of a substitution different from $\{\, p \mapsto Rp;\ p \mapsto E;\ p \mapsto R_0 \,\}$ to ϕ_0, the Var and Super rules cannot be applied to simplify the formula, therefore ϕ_0 becomes false. The application of both $p \mapsto E$ and $p \mapsto R_0$ satisfy an atom $R_0 <: p$; an application of $p \mapsto Rp$ turns this atom into itself. Therefore, only the compositions of these substitutions do not falsify the formula.

A similar argument works for the q and z type variables. ◀

▶ **Lemma 4.3.** *For the formula*

$$\phi_1 \overset{\text{def}}{=} G <: x \wedge P <: x \wedge x <: p \wedge x <: q \wedge x <: z \wedge \phi_0,$$

only elementary complete substitutions

$$[\,p \mapsto Rp;\; q \mapsto Sq;\; z \mapsto W_i z;\; x \mapsto U_i x\,]$$

$$[\,p \mapsto E;\; q \mapsto E;\; z \mapsto E;\; x \mapsto E\,]$$

with $1 \le i \le n$ *do not falsify it.*

Proof. The proof is by a case splitting into possible substitutions.

As ϕ_0 is a subformula of ϕ_1, Lemma 4.2 implies that the candidate substitutions to z that do not falsify ϕ_1 immediately are the ones from the set $\{\, z \mapsto W_i z;\; z \mapsto E;\; z \mapsto W \,\}$.

- $[\,z \mapsto W\,]\phi_1 = G <: x \wedge x <: W \wedge \ldots$

  By transitivity of subtyping, this entails the $G <: W$, which is false. Therefore, the substitutions with $z \mapsto W$ falsify ϕ_1.

- $[\,z \mapsto W_i z\,]\phi_1 = G <: x \wedge P <: x \wedge x <: W_i z \wedge \ldots$

  In order for this formula to be satisfiable, the substitution should map x to a common supertype for G and P, and it should have a nominal supertype constructed with W_i. The only such substitutions are $\{\, x \mapsto U_i x;\; x \mapsto W_i x \,\}$.

  - $[\,x \mapsto W_i x\,]\phi_1 = R_0 <: p \wedge W_i x <: p \wedge \ldots$

    This formula has the atom $W_i x <: p$, which is not falsified only if the substitution $p \mapsto W_i p$ is applied to the type variable p. But by Lemma 4.2, the candidate substitutions into p are $\{\, p \mapsto Rp;\; p \mapsto E;\; p \mapsto R_0 \,\}$. Therefore, each substitution containing $[\,x \mapsto W_i x;\; z \mapsto W_i z\,]$ falsifies ϕ_1.

  - $[\,x \mapsto U_i x\,]\phi_1 = R_0 <: p \wedge U_i x <: p \wedge S_0 <: q \wedge U_i x <: q \wedge U_i x <: W_i z \wedge \ldots$

    Common supertypes of R_0 and U_i could have only one head constructor, namely R; symmetrically, the common supertypes S_0 and U_i could be constructed only by S. Therefore, the substitutions $p \mapsto Rp$ and $q \mapsto Sq$ do not falsify the ϕ_1, while substitutions from the set

    $$[\,z \mapsto W_i z;\; x \mapsto U_i x\,] \odot \{\, p \mapsto E;\; p \mapsto R_0 \,\} \odot \{\, q \mapsto E;\; q \mapsto S_0 \,\}$$

    falsify ϕ_1. Hence, in this case, only the substitution

    $$[\,z \mapsto W_i z;\; x \mapsto U_i x;\; p \mapsto Rp;\; q \mapsto Sq\,]$$

    does not falsify ϕ_1.

- $[\,z \mapsto E\,]\phi_1 = G <: x \wedge P <: x \wedge x <: E \wedge \ldots$

  In order for this formula to be satisfiable, the substitution should map x to a common supertype for G and P, which is a subtype of E. The only appropriate substitution is $x \mapsto E$. The application of the substitution $[\,z \mapsto E;\; x \mapsto E\,]$ to ϕ_1 gives

  $$G <: E \wedge P <: E \wedge E <: p \wedge E <: q \wedge E <: E \wedge R_0 <: p \wedge S_0 <: q \wedge W <: E,$$

which simplifies into

$$E <: p \wedge E <: q \wedge R_0 <: p \wedge S_0 <: q.$$

In order for this formula to be satisfiable, the substitution should map p to a common supertype of R_0 and E, and q should be mapped into a common supertype of S_0 and E. The only appropriate substitution is $[\,p \mapsto E;\ q \mapsto E\,]$. Therefore the substitution

$$[\,z \mapsto E;\ x \mapsto E;\ p \mapsto E;\ q \mapsto E\,]$$

does not falsify ϕ_1.

We have considered all possible cases, among which only the substitutions

$$[\,z \mapsto W_iz;\ x \mapsto U_ix;\ p \mapsto Rp;\ q \mapsto Sq\,]$$

$$[\,z \mapsto E;\ x \mapsto E;\ p \mapsto E;\ q \mapsto E\,]$$

with $1 \leq i \leq n$ do not falsify ϕ_1. ◀

▶ **Lemma 4.4.** *For the formula*

$$\phi_2 \overset{\text{def}}{=} H <: y \wedge Q <: y \wedge y <: p \wedge y <: q \wedge y <: z \wedge \phi_0,$$

only elementary complete substitutions

$$[p \mapsto Rp;\ q \mapsto Sq;\ z \mapsto W_iz;\ y \mapsto V_iy\,]$$

$$[p \mapsto E;\ q \mapsto E;\ z \mapsto E;\ y \mapsto E\,]$$

with $1 \leq i \leq n$ *do not falsify it.*

Proof. Similar to the proof of Lemma 4.3. ◀

Lemma 4.3 and Lemma 4.4 imply that only elementary complete substitutions

$$[\,p \mapsto Rp;\ q \mapsto Sq;\ z \mapsto W_iz;\ x \mapsto U_ix;\ y \mapsto V_iy\,]$$

$$[\,p \mapsto E;\ q \mapsto E;\ z \mapsto E;\ x \mapsto E;\ y \mapsto E\,]$$

do not falsify $\phi_1 \wedge \phi_2$.

$\phi_1 \wedge \phi_2$ has a very important property: the application of the substitution

$$[\,p \mapsto Rp;\ q \mapsto Sq;\ z \mapsto W_iz;\ x \mapsto U_ix;\ y \mapsto V_iy\,]$$

to it and the simplification of the resulting formula turn $\phi_1 \wedge \phi_2$ into itself:

$$\begin{array}{l}
G <: U_ix \wedge P <: U_ix \wedge U_ix <: Rp \wedge U_ix <: Sq \wedge U_ix <: W_iz \wedge \\
H <: V_iy \wedge Q <: V_iy \wedge V_iy <: Rp \wedge V_iy <: Sq \wedge V_iy <: W_iz \wedge \\
R_0 <: Rp \wedge S_0 <: Sq \wedge W <: W_iz \\
\Leftrightarrow \\
U_iG <: U_ix \wedge U_iP <: U_ix \wedge Rx <: Rp \wedge Sx <: Sq \wedge W_ix <: W_iz \wedge \\
V_iH <: V_iy \wedge V_iQ <: V_iy \wedge Ry <: Rp \wedge Sy <: Sq \wedge W_iy <: W_iz \wedge \\
RR_0 <: Rp \wedge SS_0 <: Sq \wedge W_iW <: W_iz \\
\Leftrightarrow \\
G <: x \wedge P <: x \wedge x <: p \wedge x <: q \wedge x <: z \wedge \\
H <: y \wedge Q <: y \wedge y <: p \wedge y <: q \wedge y <: z \wedge \\
R_0 <: p \wedge S_0 <: q \wedge W <: z \\
= \phi_1 \wedge \phi_2
\end{array}$$

Note also that application of the substitution

$$[\,p \mapsto E;\; q \mapsto E;\; z \mapsto E;\; x \mapsto E;\; y \mapsto E\,]$$

and simplification turn $\phi_1 \wedge \phi_2$ into a true.

Notation

Let $N = \{\,1, \ldots, n\,\}$. We denote the set of finite sequences in N by $N^{<\omega}$. For $J \in N^{<\omega}$, $J = j_1 \ldots j_r$, we denote by $U_J T$ a chain of type constructors $U_{j_1} \ldots U_{j_r} T$; we define $V_J T$ and $W_J T$ similarly. Sometimes we write J^r to emphasize that the length of J is r.

▶ **Theorem 4.5.** *The formula $\phi_1 \wedge \phi_2$ has a set of models with the interpretations I_J such that*

$$v_J(x) = U_J E, \quad v_J(y) = V_J E, \quad v_J(z) = W_J E,$$
$$v_J(p) = R^r E, \quad v_J(q) = S^r E$$

where $J \in N^{<\omega}$, and r is the length of J.

Proof. As we have shown, only the compositions of the following substitutions do not falsify the formula $\phi_1 \wedge \phi_2$ immediately:

$$\begin{aligned} subst_i &\overset{\text{def}}{=} [\,p \mapsto Rp;\; q \mapsto Sq;\; z \mapsto W_i z;\; x \mapsto U_i x;\; y \mapsto V_i y\,] \\ subst_{end} &\overset{\text{def}}{=} [\,p \mapsto E;\; q \mapsto E;\; z \mapsto E;\; x \mapsto E;\; y \mapsto E\,] \end{aligned}$$

Note that as free variable substitutions are ground substitutions, we may compose them. As $subst_i$ does not change $\phi_1 \wedge \phi_2$ after simplification, and $subst_{end}$ satisfies it, a satisfying substitution for $\phi_1 \wedge \phi_2$ may only be a composition of the finite number of $subst_i$, ending with the ground substitution $subst_{end}$, i.e. $subst_{j_1} \odot \ldots \odot subst_{j_r} \odot subst_{end}$.

Thus the only satisfying ground substitutions of $\phi_1 \wedge \phi_2$ are:

$$\begin{aligned} v_J &= subst_{j_1} \odot \ldots \odot subst_{j_r} \odot subst_{end} = \\ &= [\,p \mapsto R^r E;\; q \mapsto S^r E;\; z \mapsto W_J E;\; x \mapsto U_J E;\; y \mapsto V_J E\,] \end{aligned}$$ ◀

▶ **Lemma 4.6.** *Let $\overline{L}$ be a chain of "letter" constructors, i.e. constructors from $\{A_1, \ldots, A_m\}$, $J^r \in N^{<\omega}$ with $r > 0$. Then*

$$U_J E <: \overline{L}E \vee U_J E <: \overline{L}D$$

is satisfiable if and only if

$$\overline{A_{U_{j_1}}} \ldots \overline{A_{U_{j_r}}} E = \overline{L}E$$

Proof. We prove the claim by induction on r.

Base step: $r = 1$.

$$U_{j_1} E <: \overline{L}E \vee U_{j_1} E <: \overline{L}D \Leftrightarrow \overline{A_{U_{j_1}}} E <: \overline{L}E \vee \overline{A_{U_{j_1}}} E <: \overline{L}D$$

As $\overline{A_{U_{j_1}}} E$, $\overline{L}E$ and $\overline{L}D$ are constructed from covariant type constructors $A_1, \ldots, A_m$ without the right hand side of the class table (i.e. without the strict *nominal* supertypes), we may conclude that

$$\overline{A_{U_{j_1}}} E <: \overline{L}E \vee \overline{A_{U_{j_1}}} E <: \overline{L}D \Leftrightarrow \overline{A_{U_{j_1}}} E = \overline{L}E \vee \overline{A_{U_{j_1}}} E = \overline{L}D \Leftrightarrow \overline{A_{U_{j_1}}} E = \overline{L}E$$

Induction step

Let $r = k + 1$, $J = j_1 \cdot J'$, length of J' is k.

$$\begin{aligned} &U_{j_1}U_{J'}E <: \overline{L}E \vee U_{j_1}U_{J'}E <: \overline{L}D \Leftrightarrow \\ \Leftrightarrow\ &\overline{A_{U_{j_1}}}U_{J'}E <: \overline{L}E \vee \overline{A_{U_{j_1}}}U_{J'}E <: \overline{L}D \Leftrightarrow \end{aligned}$$

As $\overline{A_{U_{j_1}}}E$, $\overline{L}E$ and $\overline{L}D$ are constructed from covariant type constructors $A_1, \dots, A_m$, which do not have strict nominal supertypes, we must require $\overline{L} = \overline{A_{U_{j_1}}L'}$.

$$\begin{aligned} \Leftrightarrow\ &\overline{A_{U_{j_1}}}U_{J'}E <: \overline{A_{U_{j_1}}L'}E \vee \overline{A_{U_{j_1}}}U_{J'}E <: \overline{A_{U_{j_1}}L'}D \Leftrightarrow \\ \Leftrightarrow\ &U_{J'}E <: \overline{L'}E \vee U_{J'}E <: \overline{L'}D \Leftrightarrow \textit{(I.H.)} \\ \Leftrightarrow\ &\overline{A_{U_{j_2}}}\dots\overline{A_{U_{j_r}}}E = \overline{L'}E \Leftrightarrow \overline{A_{U_{j_1}}A_{U_{j_2}}}\dots\overline{A_{U_{j_r}}}E = \overline{L}E \end{aligned}$$

◀

▶ **Lemma 4.7.** *Let $\overline{L}$ be a chain of constructors from $\{A_1, \dots, A_m\}$, $J^r \in N^{<\omega}$ with $r > 0$. Then*

$$V_JE <: \overline{L}E \vee V_JE <: \overline{L}D$$

is satisfiable if and only if

$$\overline{A_{V_{j_1}}}\dots\overline{A_{V_{j_r}}}E = \overline{L}E$$

Proof. Similar to the proof of Lemma 4.6. ◀

▶ **Lemma 4.8.** *The formula*

$$\phi_{J^r} \overset{\text{def}}{=} D <: t \wedge U_JE <: t \wedge V_JE <: t \wedge t \not<: E$$

is satisfiable if and only if $r > 0$ and

$$\phi'_J \overset{\text{def}}{=} \overline{A_{V_{j_1}}}\dots\overline{A_{V_{j_r}}}E = \overline{A_{U_{j_1}}}\dots\overline{A_{U_{j_r}}}E$$

is valid.

Proof. Let J be an empty sequence. Then ϕ_J becomes

$$D <: t \wedge E <: t \wedge t \not<: E.$$

D and E have only one common supertype E. But the substitution of E into t falsifies the formula because of the atom $t \not<: E$. Hence if J is an empty sequence, ϕ_J is unsatisfiable.

Let J be a non-empty sequence.

($\Rightarrow$)

Let ϕ_J be satisfiable. Then D and U_JE should have a common supertype. It cannot be E, as for all i, E is not a supertype for U_i. Consider all other supertypes of D. Those are chains of constructors from $\{ A_1, \dots, A_m \}$, terminated by either D or E. In other words, the only candidate supertypes are $\overline{L}D$ and $\overline{L}E$, where $\overline{L}$ are non-empty chains of constructors from $\{ A_1, \dots, A_m \}$.

By Lemma 4.6 and Lemma 4.7, if $v_J \vDash^{CT}_{<:} \phi_J$, then

$$v_J(t) = \overline{A_{U_{j_1}}}\dots\overline{A_{U_{j_r}}}E = \overline{A_{V_{j_1}}}\dots\overline{A_{V_{j_r}}}E$$

As ϕ_J is satisfiable, ϕ'_J is true.

($\Leftarrow$)

Let ϕ'_J be valid, then the interpretation

$$v_J(t) = \overline{A_{U_{j_1}}} \dots \overline{A_{U_{j_r}}} E = \overline{A_{V_{j_1}}} \dots \overline{A_{V_{j_r}}} E,$$

satisfies ϕ_J:

$$v_J(\phi_J) = D <: v_J(t) \wedge U_J E <: v_J(t) \wedge V_J E <: v_J(t) \wedge v_J(t) \not<: E =$$

$$D <: \overline{A_{U_{j_1}}} \dots \overline{A_{U_{j_r}}} E \wedge U_J E <: \overline{A_{U_{j_1}}} \dots \overline{A_{U_{j_r}}} E \wedge$$
$$V_J E <: \overline{A_{V_{j_1}}} \dots \overline{A_{V_{j_r}}} E \wedge \overline{A_{U_{j_1}}} \dots \overline{A_{U_{j_r}}} E \not<: E \Leftrightarrow$$

$$\overline{A_{U_{j_1}}} \dots \overline{A_{U_{j_r}}} E <: \overline{A_{U_{j_1}}} \dots \overline{A_{U_{j_r}}} E \wedge$$
$$\overline{A_{V_{j_1}}} \dots \overline{A_{V_{j_r}}} E <: \overline{A_{V_{j_1}}} \dots \overline{A_{V_{j_r}}} E \wedge$$
$$\overline{A_{U_{j_1}}} \dots \overline{A_{U_{j_r}}} E \not<: E \Leftrightarrow \top$$ ◀

▶ **Theorem 4.9.** *The formula*

$$\phi_J \overset{\text{def}}{=} D <: t \wedge U_J E <: t \wedge V_J E <: t \wedge t \not<: E$$

is satisfiable if and only if J is a solution to PCP with the pairs of words

$$\{(\overline{A_{U_1}}, \overline{A_{V_1}}), \dots, (\overline{A_{U_n}}, \overline{A_{V_n}})\}$$

over the alphabet $\{ A_1, \dots, A_m \}$, i.e.

$$\overline{A_{U_{j_1}}} \dots \overline{A_{U_{j_r}}} = \overline{A_{V_{j_1}}} \dots \overline{A_{V_{j_r}}}$$

Proof. By Lemma 4.8, ϕ_J is satisfiable if and only if $r > 0$ and

$$\phi'_J \overset{\text{def}}{=} \overline{A_{V_{j_1}}} \dots \overline{A_{V_{j_r}}} E = \overline{A_{U_{j_1}}} \dots \overline{A_{U_{j_r}}} E$$

is valid.

($\Rightarrow$)

Let ϕ'_J be valid. Then J is such a non-empty sequence of indices that the concatenation of words $\overline{A_{U_{j_1}}}, \dots, \overline{A_{U_{j_r}}}$ equals the concatenation of words $\overline{A_{V_{j_1}}}, \dots, \overline{A_{V_{j_r}}}$. Therefore J is a solution to PCP. ($\Leftarrow$)

Let J be a solution to PCP. That means that the types $\overline{A_{V_{j_1}}} \dots \overline{A_{V_{j_r}}} E$ and $\overline{A_{U_{j_1}}} \dots \overline{A_{U_{j_r}}} E$ are equal. Besides, as J solves PCP, it is non-empty. This implies the validity of ϕ'_J. ◀

▶ **Theorem 4.10.** *The formula*

$$\psi \overset{\text{def}}{=} D <: t \wedge x <: t \wedge y <: t \wedge t \not<: E \wedge \phi_1 \wedge \phi_2$$

is satisfiable if and only if PCP with the pairs of words

$$\{(\overline{A_{U_1}}, \overline{A_{V_1}}), \dots, (\overline{A_{U_n}}, \overline{A_{V_n}})\}$$

over the alphabet $\{ A_1, \dots, A_m \}$ has a solution.

Proof. By Theorem 4.5, the formula $\phi_1 \wedge \phi_2$ has a set of solutions with the interpretations $v_J(I)$ such that:

$$v_J(x) = U_J E, \quad v_J(y) = V_J E, \quad v_J(z) = W_J E,$$
$$v_J(p) = R^r E, \quad v_J(q) = S^r E$$

Hence the satisfiability of ψ is equivalent to the satisfiability of

$$\psi' \overset{\text{def}}{=} \bigvee_{J \in N^{<\omega}} v_J(\psi) = \bigvee_{J \in N^{<\omega}} D <: t \wedge U_J E <: t \wedge V_J E <: t \wedge t \not<: E = \bigvee_{J \in N^{<\omega}} \phi_J$$

By Theorem 4.5, ϕ_J is satisfiable if and only if J is a solution to PCP. Therefore ψ' is satisfiable if and only if there exists a sequence of indices J that solves PCP. ◀

▶ **Corollary 4.11.** *`SUBTYPE-SAT` is undecidable.*

▶ **Corollary 4.12.** *`SUBTYPE-SAT` is undecidable even for non-expansive class tables without contravariant constructors.*

▶ **Corollary 4.13.** *`SUBTYPE-SAT` is undecidable even for non-expansive class tables with only constant and unary constructors.*

▶ **Corollary 4.14.** *`SUBTYPE-SAT` is undecidable even for quantifier-free conjunctions of literals.*

▶ **Corollary 4.15.** *`SUBTYPE-SAT` is undecidable even for quantifier-free conjunctions of positive literals.*

Proof. To show this, we simply need to exclude the trivial solution with all variables mapped to E without using the atom $t \not<: E$ in ψ.

This can be done, for instance, by adding a new entry

$$D_0 \quad <:: \quad \overline{A_{U_1}} D, \ldots, \overline{A_{U_n}} D$$

into `PCP-CT` and altering ψ to

$$\psi \overset{\text{def}}{=} D_0 <: t \wedge x <: t \wedge y <: t \wedge \phi_1 \wedge \phi_2$$ ◀

5 Decidable fragments of `SUBTYPE-SAT`

In this section, we introduce several fragments of the `SUBTYPE-SAT` and prove their decidability. Decidability could be achieved by restricting the class table or the formula. A semiground fragment constrains the formula, allowing the class table to be arbitrary (but non-expansive). Using the intuitions from the proof, we conjecture another decidable fragment that constraints the shape of the class table, leaving the formula to be arbitrary.

▶ **Definition 5.1.** *A* semiground *atom (literal) is an atom (literal) with at least one ground argument. A* normalized semiground *atom (literal) is an atom (literal) with one ground argument and one variable argument (note that the `SUBTYPE-SAT` language only has binary predicate symbols $<:$ and $=$).*

A (normalized) semiground formula is a quantifier-free formula that contains only (normalized) semiground atoms. A semiground fragment *is a set of semiground formulas.*

▶ **Theorem 5.2.** *Each semiground atom is logically equivalent to some normalized semiground formula.*

Proof. The theorem obviously holds in case an atom is already normalized. If both arguments of an atom are ground, it is equivalent to either $\top$ or $\bot$. It remains to consider only the atom $T <: U$ with both T and U constructed. In this case we apply the subtyping rules:

- Let $T = C\texttt{<}\overline{T}\texttt{>}$ and $U = C\texttt{<}\overline{U}\texttt{>}$. Then we simplify $T <: U$ using (Var) rule:

$$C\texttt{<}\overline{T}\texttt{>} <: C\texttt{<}\overline{U}\texttt{>} \Leftrightarrow \bigwedge_i T_i <:_{var(C\#i)} U_i$$

As either $C\texttt{<}\overline{T}\texttt{>}$ or $C\texttt{<}\overline{U}\texttt{>}$ is ground, either every T_i, or every U_i is ground. This means that every atom of the resulting formula is still semiground.

- Let $T = C\texttt{<}\overline{T}\texttt{>}$ and $U = D\texttt{<}\overline{U}\texttt{>}$ with $C \neq D$. Then we apply the (Super) rule:

$$C\texttt{<}\overline{T}\texttt{>} <: D\texttt{<}\overline{U}\texttt{>} \Leftrightarrow \bigvee_j D\texttt{<}\overline{W_j}\texttt{>} <: D\texttt{<}\overline{U}\texttt{>},$$

where $D\texttt{<}\overline{W_j}\texttt{>} = [\,\overline{x} \mapsto \overline{T}\,]V_j$ and $C\texttt{<}\overline{x}\texttt{>} <:: V_j$. Obviously, all atoms in this formula are still semiground.

As the non-deterministic application of subtyping rules with occurence checks is a decision procedure for non-expansive inheritance [8], the process is either terminating, resulting in a normalized formula, or eventually the chain of simplifications loops on some atom. In this case, the atom is tautologically false. ◀

▶ **Corollary 5.3.** *Each semiground formula is logically equivalent to some normalized semiground formula.*

▶ **Definition 5.4.** *Let $Q \overset{\text{def}}{=} A_1 \wedge \ldots \wedge A_n$ be a conjunct.* Maximal connected subconjuncts *is a set of conjuncts $\{Q_1, \ldots, Q_m\}$ such that*

1. $Q = Q_1 \wedge \ldots \wedge Q_m$
2. *Q_i and Q_j have distinct literals if $i \neq j$*
3. *If A_i and A_j have common variables then they occur in the same Q_k*

▶ **Lemma 5.5.** *If* SUBTYPE-SAT *is decidable for the conjunction of normalized semiground literals (with the only free variable x) of the form*

$$\bigwedge_i T_i <: x \wedge \bigwedge_j x <: U_j \wedge \bigwedge_k V_k \not<: x \wedge \bigwedge_l x \not<: W_l,$$

then SUBTYPE-SAT *is decidable for the whole semiground fragment.*

Proof. Let ψ be a semiground formula. By Corollary 5.3

$$\psi \sim^{CT}_{<:} \bigvee_i \bigwedge_j \psi_{i,j},$$

where $\psi_{i,j}$ is a normalized semiground literal. Each conjunct can be divided into maximal connected subconjuncts in such a way that each one of them contains only one variable. The algorithm then can check the satisfiability of each group separately. ◀

We define a set of substitutions

$$CtorSubsts = \{\, x \mapsto C\texttt{<}\overline{x}\texttt{>} \,\},$$

where C is a constructor from CT, and $\overline{x}$ is a vector of distinct variables.

▶ **Lemma 5.6.** *Let ψ be a conjunction of normalized semiground literals with only one free variable. Then $\psi(x)$ is equisatisfiable with*

$$\psi^{subst} \overset{\text{def}}{=} \bigvee_{subst \in CtorSubsts} subst\,\psi(x)$$

▶ **Lemma 5.7.** *The `SUBTYPE-SAT` problem is decidable for conjunctions of normalized semiground literals with one free variable.*

Proof. Let ψ_0 be a conjunction of normalized semiground literals with one free variable. Apply Lemma 5.6. ψ_0^{subst} is a semiground formula. If it is ground, we may check its satisfiablity and terminate. Otherwise, we may act as in Lemma 5.5: apply Corollary 5.3 and convert the formula to DNF. ψ_0^{subst} is satisfiable iff one of the conjuncts is satisfiable.

Choose the conjunct non-deterministically and divide it into maximal connected subconjuncts. The algorithm then checks the satisfiability of each subconjunct ϕ_1, i.e. we have reduced the problem to itself. The described procedure enumerates the (possibly infinite) sequence $\{ \psi_i \}_{i \in \mathbb{N}}$.

Without loss of generality, we may conclude that all literals in ψ_i are distinct and that all conjuncts have an identical free variable. Two conjuncts are equal if they are identical as sets of literals. Note that if $\psi_i = \psi_j$ for some $i \neq j$, then ψ_i is unsatisfiable, and we may terminate.

For the conclusion of the proof, we refer to the notion of inheritance closure from [8]. It is known that the inheritance closure of a finite set of types within a non-expansive class table is finite [8]. Now, note that all ground types of ψ_i are obtained by application of subtyping rules, thus they are elements of the inheritance closure for the set of ground types of ψ_0. That means that only a finite number of literals may occur in $\{ \psi_i \}_{i \in \mathbb{N}}$. As every literal occurs in each conjunct no more than once, eventually $\psi_i = \psi_j$ for $i < j$. Therefore, our procedure terminates. ◀

▶ **Theorem 5.8.** *`SUBTYPE-SAT` is decidable for queries with a semiground fragment.*

Proof. By Lemma 5.5 and Lemma 5.7. ◀

Formulas from Example 3.3 and Example 3.4 refer to the described fragment and their satisfiability can be checked. And the formula from Theorem 4.10 goes beyond the fragment, which is consistent with the undecidability of PCP.

The proof of Theorem 5.2 can be used to produce a generalized scheme for obtaining new decidable fragments. We describe this scheme below.

▶ **Definition 5.9.** *A* normalized *atom (literal) is an atom (literal) that does not have open constructed types. In other words, a normalized atom is either a normalized semiground atom, or $x <: y$, or $x = y$, where x and y are variables. A normalized formula is a quantifier-free formula that only contains normalized atoms.*

Consider a conjunct whose maximal connected subconjuncts consist of a single element. Acting similarly to Lemma 5.6, we simplify the obtained formula, convert it to DNF, and split it into maximal connected subconjuncts. If we can impose some restrictions on the class table or the formula so that the number of different obtained subconjuncts is finite, then we have a criterion for the termination of the procedure.

▶ **Example 5.10.** Let the class table be organized in such a way that for each pair of constructors C and D, the literals $C\text{<}\overline{x}\text{>} <: D\text{<}\overline{y}\text{>}$ and $C\text{<}\overline{x}\text{>} \not<: D\text{<}\overline{y}\text{>}$ are transformed via the application of Corollary 5.3 into such a disjunction of normalized conjuncts that each of them may be partitioned into maximal connected subconjuncts, containing no more than one variable from $\overline{x}$ and no more than one variable from $\overline{y}$. Applying the scheme described above to normalized conjuncts with n different free variables, we can only obtain normalized conjuncts which have no more than n free variables. Moreover, each quantifier-free formula can be simplified into a normalized formula. As the number of distinct types in an inheritance closure is bounded [8], the number of conjuncts is bounded, so the procedure will terminate.

This idea may be used to introduce some syntactic restrictions on the shape of the class table. We leave it for the future work.

6 Related work

There is a long line of research on decidability of ground subtyping of nominal type systems with variance. One of the latest studies has shown the Turing-completeness of Java subtyping [7]. C++ templates are also known to be Turing-complete [18]. Scala [12], OCaml [10, 15] and Haskell type systems with extensions are undecidable as well. However, .NET subtyping is decidable [5, 8]. These papers formalize and investigate *type checking* in certain programming languages. In contrast, we investigate the subtyping in the presence of open types, founded on the results on ground subtyping.

Constraint satisfiability

The most relevant recent work is [16]. Motivated by the same goals, it reduces the satisfiability problem of type-based partially ordered sets to the first-order satisfiability problem and proposes to use SMT-solvers to solve the constraints. Unlike our work, the type system under consideration is a nominal fragment of Java type system without generics.

The satisfiability problem for subtyping constraints and its computational complexity for more general (in comparison to [16]) fragments of type systems is explored in [14, 6, 9, 11]. These works explore constraints on finite and recursive types, structural and non-structural subtyping and type constructors with covariant and contravariant type parameters; our work studies a more general problem.

The paper [17] shows the undecidability of first-order subtyping constraints for non-structural subtyping. This result entails the undecidability of `SUBTYPE-SAT`, but it uses a significantly larger fragment of first-order logic (in particular, universally quantified formulas). Our proof uses only quantifier-free conjunctions of positive atoms and those features of nominal subtyping with variance which are not present in the non-structural case.

7 Conclusion

We have introduced the satisfiability problem of nominal subtyping with variance and studied some of its properties. The undecidability of the problem has been proven using a noticeably small fragment of the type system. We have also discussed a number of non-trivial decidable fragments and a scheme to obtain other decidable fragments. Finding more extensive decidable fragments is an open problem: for example, it could be done by introducing syntactic restrictions on the shape of the class table. In addition, it would be interesting to compare the decidable fragments with the fragment that is most widely used in practice. Another area of future work is the construction of effective procedures for solving the `SUBTYPE-SAT` problem. Subsequently, such decision procedures can be implemented in SMT-solvers, which makes them easy to use in a variety of SMT-based program analysis approaches.

References

1 Mike Barnett, K Rustan M Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *International Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, pages 49–69. Springer, 2004.

2 Clark Barrett and Cesare Tinelli. Satisfiability Modulo Theories. In *Handbook of Model Checking*, pages 305–343. Springer, 2018.

3 Leonardo Mendonça de Moura and Nikolaj Bjørner. Satisfiability modulo theories: introduction and applications. *Commun. ACM*, 54(9):69–77, 2011.

4 ECMA ECMA. 335: Common language infrastructure (CLI), 2005.

5 Burak Emir, Andrew Kennedy, Claudio V. Russo, and Dachuan Yu. Variance and Generalized Constraints for C# Generics. In *ECOOP*, volume 4067 of *Lecture Notes in Computer Science*, pages 279–303. Springer, 2006.

6 Alexandre Frey. Satisfying Subtype Inequalities in Polynomial Space. In *SAS*, volume 1302 of *Lecture Notes in Computer Science*, pages 265–277. Springer, 1997.

7 Radu Grigore. Java generics are Turing complete. In *POPL*, pages 73–85. ACM, 2017.

8 Andrew J Kennedy and Benjamin C Pierce. On decidability of nominal subtyping with variance, 2007.

9 Viktor Kuncak and Martin C. Rinard. Structural Subtyping of Non-Recursive Types is Decidable. In *LICS*, pages 96–107. IEEE Computer Society, 2003.

10 Mark Lillibridge. *Translucent sums: A foundation for higher-order module systems.* PhD thesis, Carnegie Mellon University, 1997.

11 Joachim Niehren, Tim Priesnitz, and Zhendong Su. Complexity of Subtype Satisfiability over Posets. In *ESOP*, volume 3444 of *Lecture Notes in Computer Science*, pages 357–373. Springer, 2005.

12 Martin Odersky. Scaling DOT to Scala–soundness, 2016.

13 Emil L Post. A variant of a recursively unsolvable problem. *Bulletin of the American Mathematical Society*, 52(4):264–268, 1946.

14 Vaughan R. Pratt and Jerzy Tiuryn. Satisfiability of Inequalities in a Poset. *Fundam. Inform.*, 28(1-2):165–182, 1996.

15 Andreas Rossberg. Undecidability of OCaml type checking, 1999.

16 Elena Sherman, Brady J. Garvin, and Matthew B. Dwyer. Deciding Type-Based Partial-Order Constraints for Path-Sensitive Analysis. *ACM Trans. Softw. Eng. Methodol.*, 24(3):15:1–15:33, 2015.

17 Zhendong Su, Alexander Aiken, Joachim Niehren, Tim Priesnitz, and Ralf Treinen. The first-order theory of subtyping constraints. In *POPL*, pages 203–216. ACM, 2002.

18 Todd L. Veldhuizen. C++ Templates are Turing complete, 2003.

19 Mirko Viroli. On the recursive generation of parametric types. Technical report, Technical Report DEIS-LIA-00-002, Universita di Bologna, 2000.

Static Analysis for Asynchronous JavaScript Programs

Thodoris Sotiropoulos[1]
Athens University of Economics and Business, Greece

Benjamin Livshits
Imperial College London, UK
Brave Software, London, UK

Abstract

Asynchrony has become an inherent element of JavaScript, as an effort to improve the scalability and performance of modern web applications. To this end, JavaScript provides programmers with a wide range of constructs and features for developing code that performs asynchronous computations, including but not limited to timers, promises, and non-blocking I/O.

However, the data flow imposed by asynchrony is implicit, and not always well-understood by the developers who introduce many asynchrony-related bugs to their programs. Worse, there are few tools and techniques available for analyzing and reasoning about such asynchronous applications. In this work, we address this issue by designing and implementing one of the first static analysis schemes capable of dealing with almost all the asynchronous primitives of JavaScript up to the 7th edition of the ECMAScript specification.

Specifically, we introduce the *callback graph*, a representation for capturing data flow between asynchronous code. We exploit the callback graph for designing a more precise analysis that respects the execution order between different asynchronous functions. We parameterize our analysis with one novel context-sensitivity flavor, and we end up with multiple analysis variations for building callback graph.

We performed a number of experiments on a set of hand-written and real-world JavaScript programs. Our results show that our analysis can be applied to medium-sized programs achieving 79% precision, on average. The findings further suggest that analysis sensitivity is beneficial for the vast majority of the benchmarks. Specifically, it is able to improve precision by up to 28.5%, while it achieves an 88% precision on average without highly sacrificing performance.

2012 ACM Subject Classification Theory of computation → Program analysis; Software and its engineering → Semantics

Keywords and phrases static analysis, asynchrony, JavaScript

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2019.8

Related Version A full version of the paper is available at `https://www.imperial.ac.uk/media/imperial-college/faculty-of-engineering/computing/public/1718-pg-projects/SotiropoulosT-Static-Analysis-for-Asynchronous-JavaScript-Programs.pdf`.

Funding The work described here was partially supported by the CROSSMINER Project, which has received funding from the European Union's Horizon 2020 Research and Innovation Programme under grant agreement No. 732223.

Acknowledgements We would like to thank the anonymous ECOOP reviewers, as well as Max Schlueter and Dimitris Mitropoulos for their insightful comments and feedback.

[1] The work of this author was mostly done while at Imperial College London.

33rd European Conference on Object-Oriented Programming (ECOOP 2019).
Editor: Alastair F. Donaldson; Article No. 8; pp. 8:1–8:29
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

JavaScript is an integral part of web development. Since its initial release in 1995, it has evolved from a simple scripting language – primarily used for interacting with web pages – into a complex and general-purpose programming language used for developing both client- and server-side applications. The emergence of Web 2.0 along with the dynamic features of JavaScript, which facilitate a flexible and rapid development, have led to a dramatic increase in its popularity. Indeed, according to the annual statistics provided by Github, which is the leading platform for hosting open-source software, JavaScript is by far the most popular and active programming language from 2014 to 2018 [14].

Although the dominance of JavaScript is impressive, the community has widely criticized it because it poses many concerns as to the security or correctness of the programs [36]. JavaScript is a language with a lot of dynamic and metaprogramming features, including but not limited to prototype-based inheritance, dynamic property lookups, implicit type coercions, dynamic code loading, and more. Many developers often do not understand or do not properly use these features. As a result, they introduce errors to their programs – which are difficult to debug – or baleful security vulnerabilities. In this context, JavaScript has attracted many engineers and researchers over the past decade to: (1) study and reason about its peculiar characteristics, and (2) develop new tools and techniques – such as type analyzers [19, 23, 21], IDE and refactoring tools [6, 7, 8, 11], or bug and vulnerability detectors [28, 15, 34, 31, 5, 37] – to assist developers with the development and maintenance of their applications. Program analysis, and especially static analysis, plays a crucial role in the design of such tools [38].

Additionally, preserving the scalability of modern web applications has become more critical than ever. As an effort to improve the throughput of web programs, JavaScript has started to adopt an event-driven programming paradigm [4]. In this context, code is executed *asynchronously* in response to certain events, e.g., user input, a response from a server, data read from disk, etc. In the first years of JavaScript, someone could come across that asynchrony mainly in a browser environment e.g., DOM events, AJAX calls, timers, etc. However, in recent years, asynchrony has become a salient and intrinsic element of the language, as newer versions of the language's core specification (i.e., ECMAScript) have introduced more and more asynchronous features. For example, ECMAScript 6 introduces promises; an essential element of asynchronous programming that allows developers to track the state of an asynchronous computation easily. Specifically, the state of a promise object can be: (1) *fulfilled* when the associated operation is complete, and the promise object tracks its resulting value, (2) *rejected* when the associated operation has failed, and (3) *pending* when the associated operation has been neither completed nor failed.

Promises are particularly useful for asynchronous programming because they provide an intuitive way for creating chains of asynchronous computation that facilitate the enforcement of program's execution order as well as error propagation [12, 11]. Depending on their state, promises trigger the execution of certain functions (i.e., *callbacks*) asynchronously. To do so, the API of promises provides the method `x.then(f1, f2)` for registering new callbacks (i.e., `f1` and `f2`) on a promise object `x`. For example, we call the callback `f1` when the promise is fulfilled, while we trigger the callback `f2` once the promise is rejected. The method `x.then()` returns a new promise which the return value of the provided callbacks (i.e., `f1, f2`) fulfills. Since their emergence, JavaScript developers have widely embraced promises. For example, a study in 2015 showed that 75% of JavaScript frameworks use promises [12].

```
 1  asyncRequest(url, options)
 2    .then(function (response) {
 3      honoka.response = response.clone();
 4
 5      switch (options.dataType.toLowerCase()) {
 6        case "arraybuffer":
 7          return honoka.response.arrayBuffer();
 8        case "json":
 9          return honoka.response.json();
10        ...
11        default:
12          return honoka.response.text();
13      }
14    })
15    .then(function (responseData) {
16      if (options.dataType === "" || options.dataType === "auto") {
17        const contentType = honoka.response.headers.get("Content-Type");
18        if (contentType && contentType.match("/application\/json/i")) {
19          responseData = JSON.parse(responseData);
20        }
21      }
22      ...
23    });
```

Figure 1 Real-world example that mixes promises with asynchronous I/O.

Building upon promises, newer versions of ECMAScript have added more language features related to asynchrony. Specifically, in ECMAScript 8, we have the `async/await` keywords. The `async` keyword declares a function as asynchronous that returns a promise fulfilled with its return value, while `await x` defers the execution of an asynchronous function until the promise object `x` is settled (i.e., it is either fulfilled or rejected). The latest edition of ECMAScript (ECMAScript 9) adds asynchronous iterators and generators that allow developers to iterate over asynchronous data sources.

Beyond promises, many JavaScript applications are written to perform non-blocking I/O operations. Unlike traditional statements, when we perform a non-blocking I/O operation, the execution is not interrupted until I/O terminates. By contrast, the I/O operation is done asynchronously, which means that the execution proceeds to the next tasks while I/O takes place. Programmers often mix asynchronous I/O with promises. For instance, consider the real-world example of Figure 1. At line 1, the code performs an asynchronous request and returns a promise object that is fulfilled asynchronously once the request succeeds. Then, this promise object can be used for processing the response of the server asynchronously. For instance, at lines 2–23, we create a promise chain. The first callback of this chain (lines 2–14) clones the response of the request, and assigns it to the property `response` of the object `honoka` (line 3). Then, it parses the body of the response, and finally, the return value of the callback fulfills the promise object allocated by the first invocation of `then()` (lines 5–13). The second callback (lines 15–23) retrieves the headers of the response – which the statement at line 3 assigns to `honoka.response` – and if the content type is "application/json", it converts the data of the response into a JSON object (lines 17–19).

Like the other characteristics of JavaScript, programmers do not always clearly understand asynchrony, as a large number of asynchrony-related questions issued in popular sites like `stackoverflow.com`[2] [27, 26], or the number of bugs reported in open-source repositories [39, 5] indicate. However, existing tools and techniques have limited (and in many cases no)

[2] `https://stackoverflow.com/`

support for asynchronous programs. Designing static analysis for asynchrony involves several challenges not addressed by previous work. In particular, existing tools mainly focus on the event system of client-side JavaScript applications [18, 33], and they lack the support of the more recent features added to the language, such as promises. Also, many previous tools conservatively consider that all the asynchronous callbacks processed by the *event loop* – the program point that continuously waits for new events to come and is responsible for the scheduling and execution of callbacks – can be called in any order [18, 33, 21]. However, such an approach may lead to imprecision and false positives. Back to the example of Figure 1, it is easy to see that an analysis that does not respect the execution order between the first and the second callback will report a type error at line 17 (access of `honoka.response.headers.get (" Content - Type ")`). Specifically, an imprecise analysis assumes that the callback defined at lines 15–23 might be executed first; therefore, `honoka.response`, assigned at line 3, might be uninitialized.

In this work, we tackle the issues above, by designing and implementing a static analysis that deals with asynchronous JavaScript programs. To do so, we first propose a model for understanding and expressing a wide range of asynchronous constructs found in JavaScript. Based on this model, we design our static analysis. We propose a new representation, which we call callback graph, that provides information about the execution order of the asynchronous code. The callback graph proposed in this work tries to shed light on how data flow between asynchronous code is propagated. Contrary to previous works, we leverage the callback graph and devise a more precise analysis that respects the execution order of asynchronous functions. Furthermore, we parameterize our analysis with one novel context-sensitivity option designed for asynchronous code. Specifically, we distinguish data flow between asynchronous callbacks based on the promise object that they belong to, or the next computation that the execution proceeds to.

Contributions. Our work makes the following four contributions:

- We propose a calculus, i.e., $\lambda_{\mathtt{q}}$, for modeling asynchrony. Our calculus unifies different asynchronous idioms into a single model. Thus, we can express promises, timers, asynchronous I/O, and other asynchronous features found in the language (§2).
- We design and implement a static analysis that is capable of handling asynchronous JavaScript programs by exploiting the abstract version of $\lambda_{\mathtt{q}}$. To the best of our knowledge, our analysis is the first to deal with JavaScript promises (§3.1).
- We propose the callback graph, a representation that illustrates the execution order between asynchronous callbacks. Building on that, we propose a more precise analysis, (i.e., callback-sensitive analysis) that internally consults the callback graph to retrieve information about the temporal relations of asynchronous functions so that it propagates data flow accordingly. Besides that, we parameterize our analysis with a novel context-sensitivity option (i.e., QR-sensitivity) used for distinguishing asynchronous callbacks. (§3.2, §3.3).
- We evaluate the performance and the precision of our analysis on a set of micro benchmarks and a set of real-world JavaScript modules. For the impatient reader, we find that our prototype is able to analyze medium-sized asynchronous programs, and the analysis sensitivity is beneficial for improving the analysis precision. The results showed that our analysis is able to achieve a 79% precision for the callback graph, on average. The analysis sensitivity (i.e. callback- and QR-sensitivity) can further improve callback graph precision by up to 28.5% and reduce the total number of type errors by 13.9% as observed in the real-world benchmarks (§4).

$$
\begin{aligned}
&v \in \mathit{Val} ::= \ \ldots \mid \bot\\
&e \in \mathit{Exp} ::= \ \ldots\\
&\qquad \mid \mathsf{newQ}() \mid e.\mathsf{fulfill}(e) \mid e.\mathsf{reject}(e) \mid e.\mathsf{registerFul}(e, e, \ldots) \mid e.\mathsf{registerRej}(e, e, \ldots)\\
&\qquad \mid \mathsf{append}(e) \mid \mathsf{pop}() \mid \bullet\\
&E ::= \ \ldots\\
&\qquad \mid E.\mathsf{fullfill}(e) \mid v.\mathsf{fulfill}(E) \mid E.\mathsf{reject}(e) \mid v.\mathsf{reject}(E)\\
&\qquad \mid E.\mathsf{registerFul}(e, e, \ldots) \mid v.\mathsf{registerFul}(v, \ldots, E, e, \ldots)\\
&\qquad \mid E.\mathsf{registerRej}(e, e, \ldots) \mid v.\mathsf{registerRej}(v, \ldots, E, e, \ldots)\\
&\qquad \mid \mathsf{append}(E)
\end{aligned}
$$

Figure 2 The syntax of $\lambda_{\mathtt{q}}$.

2 Modeling Asynchrony

As a starting point, we need to define a model to express asynchrony. The goal of this model is to provide us with the foundations for gaining a better understanding of the asynchronous primitives and ease the design of a static analysis for asynchronous JavaScript programs. This model is expressed through a calculus called $\lambda_{\mathtt{q}}$; an extension of $\lambda_{\mathtt{js}}$ which is the core calculus for JavaScript developed by Guha et. al. [16]. Our calculus is inspired by previous work [26, 25], however, it is designed to be flexible so that we can express promises, timers, asynchronous I/O, and other sources of asynchrony found in the language up to ECMAScript 7, such as thenables. Also unlike previous work – as we will see later on – our calculus enables us to model the effects of the exceptions trapped by the event loop. Additionally, we are able to handle callbacks that are invoked with arguments passed during callback registration; something that is not supported by the previous models. However, note that λ_q does not handle the `async/await` keywords and the asynchronous iterators/generators introduced in the recent editions of the language.

2.1 The $\lambda_{\mathtt{q}}$ calculus

The key component of our model is *queue objects*. Queue objects are closely related to JavaScript promises. Specifically, a queue object – like a promise – tracks the state of an asynchronous job, and it can be in one of the following states: (1) *pending*, (2) *fulfilled* or (3) *rejected*. A queue object may trigger the execution of callbacks depending on its state. Initially, a queue object is pending. A pending queue object can transition to a fulfilled or a rejected queue object. A queue object is fulfilled or rejected with a value. This value is later passed as an argument of the corresponding callbacks. Once a queue object is either fulfilled or rejected, its state is final and cannot be changed. We keep the same terminology as promises, so if a queue object is either fulfilled or rejected, we call it *settled*.

2.1.1 Syntax and Domains

Figure 2 illustrates the syntax of $\lambda_{\mathtt{q}}$. For brevity, we present only the new constructs added to the language. Specifically, we add eight new expressions:

- $\mathsf{newQ}()$: This expression creates a fresh queue object in a pending state. It has no callbacks associated with it.
- $e_1.\mathsf{fulfill}(e_2)$: This expression fulfills the receiver (i.e., the expression e_1) with the value of e_2.

$$
\begin{aligned}
a \in \mathit{Addr} &= \{l_i \mid i \in \mathbb{Z}^*\} \cup \{l_{time}, l_{io}\} \\
\pi \in \mathit{Queue} &= \mathit{Addr} \hookrightarrow \mathit{QueueObject} \\
q \in \mathit{QueueObject} &= \mathit{QueueState} \times \mathit{Callback}^* \times \mathit{Callback}^* \times \mathit{Addr}^* \\
s \in \mathit{QueueState} &= \{\mathsf{pending}\} \cup (\{\mathsf{fulfilled}, \mathsf{rejected}\} \times \mathit{Val}) \\
\mathit{clb} \in \mathit{Callback} &= \mathit{Addr} \times F \times \mathit{Val}^* \\
\kappa \in \mathit{ScheduledCallbacks} &= \mathit{Callback}^* \\
\kappa \in \mathit{ScheduledTimerIO} &= \mathit{Callback}^* \\
\phi \in \mathit{QueueChain} &= \mathit{Addr}^*
\end{aligned}
$$

Figure 3 The concrete domains of λ_q.

- e_1.reject(e_2): This expression rejects the receiver (i.e., the expression e_1) with the value of e_2.
- e_1.registerFul($e_2, e_3, \ldots$): This expression registers the callback e_2 to the receiver. This callback is executed *only* when the receiver is fulfilled. This expression also expects another queue object passed as the second argument, i.e., e_3. This queue object will be fulfilled with the return value of the callback e_2. This allows us to model chains of promises where a promise resolves with the return value of another promise's callback. This expression might receive optional parameters (i.e., expressed through "$\ldots$") with which e_2 is called when the queue object is fulfilled with $\bot$ value. We will justify the intuition behind that later on.
- e_1.registerRej($e_2, e_3, \ldots$): The same as e.registerFul($\ldots$), but this time the given callback is executed once the receiver is rejected.
- append(e): This expression appends the queue object e to the top of the current queue chain. As we will see later, the top element of a queue chain corresponds to the queue object that is needed to be rejected when the execution encounters an uncaught exception.
- pop(): This expression pops the top element of the current queue chain.
- The last expression $\bullet$ stands for the event loop.

Observe that we use evaluation contexts [9, 16, 27, 25, 26] to express how the evaluation of an expression proceeds. The symbol E denotes which sub-expression is currently being evaluated. For instance, E.fulfill(e) describes that we evaluate the receiver of fulfill, whereas v.fulfill(E) implies that the receiver has been evaluated to a value v, and the evaluation now lies on the argument of fulfill. Beyond those expressions, the λ_q calculus introduces a new value, that is, $\bot$. This value differs from `null` and `undefined` because it expresses the absence of value. Thus, it does not have correspondence with any JavaScript value.

Figure 3 presents the semantic domains of λ_q. In particular, a queue is a partial map of addresses to queue objects. The symbol l_i – where i is a positive integer – indicates an address. Notice that the set of the addresses also includes two special reserved addresses, i.e., l_{time}, l_{io}. We use these two addresses to store the queue objects responsible for keeping the state of callbacks related to timers and asynchronous I/O respectively (Section 2.1.4 explains how we model those JavaScript features). A queue object is described by its state – recall that a queue object is either pending or fulfilled and rejected with a value – a sequence of callbacks executed on fulfillment, and a sequence of callbacks called on rejection. The last element of a queue object is a sequence of addresses. These addresses correspond to the queue objects that are dependent on the current. A queue object q_1 depends on q_2, when q_1 is settled whenever q_2 is settled. This means that when the queue object q_2 is fulfilled (rejected), q_1 is also fulfilled (rejected) with the same value as q_2. We create such dependencies when we settle a queue object with another queue object. In this case, the receiver is dependent on the queue object used as an argument.

Moving to the domains of callbacks, we see that a callback consists of an address, a function, and a list of values (i.e., arguments of the function). Note that the first component denotes the address of the queue object that is fulfilled with the return value of the function. In the list of callbacks $\kappa \in \mathit{ScheduledCallbacks}$, we keep the order in which callbacks are scheduled. Note that we maintain one more list of callbacks (i.e., $\tau \in \mathit{ScheduledTimerIO}$) where we store the callbacks registered on the queue objects located at the addresses l_{time}, l_{io}. We defer the discussion about why we keep two separate lists until Section 2.1.3.

A queue chain $\phi \in \mathit{QueueChain}$ is a sequence of addresses. In a queue chain, we store the queue object that we reject, when there is an uncaught exception in the current execution. Specifically, when we encounter an uncaught exception, we inspect the top element of the queue chain, and we reject it. If the queue chain is empty, we propagate the exception to the call stack as usual.

2.1.2 Semantics

Equipped with the appropriate definitions of the syntax and domains, in Figure 4, we present the small-step semantics of λ_{q} which is an adaptation of previous calculi [25, 26]. Note that we demonstrate the most representative rules of our semantics; we omit some rules for brevity. For what follows, the binary operation denoted by the symbol $\cdot$ means the addition of an element to a list, the operation indicated by :: stands for list concatenation, while $\downarrow_i$ means the projection of the i^{th} element.

The rules of our semantics adopt the following form:

$$\pi, \phi, \kappa, \tau, E[e] \rightarrow \pi', \phi', \kappa', \tau', E[e']$$

That form expresses that a given queue π, a queue chain ϕ, two sequences of callbacks κ and τ, and an expression e in the evaluation context E lead to a new queue π', a new queue chain ϕ', two new sequences of callbacks κ' and τ', and a new expression e' in the same evaluation context E, assuming that the expression e is reduced to e' (i.e., $e \hookrightarrow e'$). The [E-CONTEXT] rule describes this behavior.

The [NEWQ] rule creates a new queue object and adds it to the queue using a fresh address. This new queue object is pending, and it does not have any callbacks related to it.

The [FULFILL-PENDING] rule demonstrates the case when we fulfill a pending queue object with the value v, where $v \neq \bot$, and $v \notin \mathit{dom}(\pi)$ (i.e., it does not correspond to any queue object). First, we change the state of the receiver object from “pending” to “fulfilled”. Second, we update the already registered callbacks (if any) by setting the value v as the only argument of them (forming the list t'). Third, we asynchronously fulfill any queue object that depend on the current one (see the list d). To do so, we form the list of callbacks f. Every element $(\alpha, \lambda x.x, [v]) \in f$ contains the identity function $\lambda x.x$ that is invoked with the value v. Upon exit, the identity function fulfills the queue object α, where $\alpha \in d$. Then, we add the updated callbacks t' and the list of functions f to the list of scheduled callbacks κ. Notice that the receiver must be neither l_{time} nor l_{io}. Also, note that the callbacks of f are scheduled before those included in t' (i.e., $\kappa' = \kappa :: (f :: t')$). This means that we fulfill any dependent queue objects before the execution of callbacks.

E-CONTEXT

$$\frac{e \hookrightarrow e'}{\pi, \phi, \kappa, \tau, E[e] \rightarrow \pi', \phi', \kappa', \tau', E[e']}$$

NEWQ

$$\frac{\text{fresh}\alpha \qquad \pi' = \pi[\alpha \mapsto (\text{pending}, [], [], [])]}{\pi, \phi, \kappa, \tau, E[\text{newQ}()] \rightarrow \pi', \phi, \kappa, \tau, E[\alpha]}$$

FULFILL-PENDING

$$\frac{\begin{array}{c} v \neq \bot \qquad (\text{pending}, t, k, d) = \pi(p) \qquad v \notin dom(\pi) \\ t' = \langle(\alpha, f, [v]) \mid (\alpha, f, a) \in t\rangle \qquad f = \langle(\alpha, \lambda x.x, [v]) \mid \alpha \in d\rangle \\ \kappa' = \kappa :: (f :: t') \qquad \chi = (\text{fulfilled}, v) \qquad \pi' = \pi[p \mapsto (\chi, [], [], [])] \\ p \neq l_{time} \wedge p \neq l_{io} \end{array}}{\pi, \phi, \kappa, \tau, E[p.\text{fulfill}(v)] \rightarrow \pi', \phi, \kappa', \tau, E[\text{undef}]}$$

FULFILL-PEND-PEND

$$\frac{\begin{array}{c} \pi(p) \downarrow_1 = \text{pending} \qquad v \in dom(\pi) \\ (\text{pending}, t, k, d) = \pi(v) \qquad \pi' = \pi[v \mapsto (\text{pending}, t, k, d \cdot p)] \end{array}}{\pi, \phi, \kappa, \tau, E[p.\text{fulfill}(v)] \rightarrow \pi', \phi, \kappa, \tau, E[\text{undef}]}$$

FULFILL-PEND-FUL

$$\frac{\pi(p) \downarrow_1 = \text{pending} \qquad v \in dom(\pi) \qquad \pi(v) \downarrow_1 = (\text{fulfilled}, v')}{\pi, \phi, \kappa, \tau, E[p.\text{fulfill}(v)] \rightarrow \pi, \phi, \kappa, \tau, E[p.\text{fulfill}(v')]}$$

FULFILL-SETTLED

$$\frac{\pi(p) \downarrow_1 \neq \text{pending}}{\pi, \phi, \kappa, \tau, E[p.\text{fulfill}(v)] \rightarrow \pi, \phi, \kappa, \tau, E[\text{undef}]}$$

REGISTERFUL-PENDING

$$\frac{\begin{array}{c} (\text{pending}, t, k, d) = \pi(p) \qquad t' = t \cdot (p', f, [n_1, n_2, \dots, n_n]) \\ \pi' = \pi[p \mapsto (\text{pending}, t', k, d)] \end{array}}{\pi, \phi, \kappa, \tau, E[p.\text{registerFul}(f, p', n_1, n_2, \dots, n_n)] \rightarrow \pi', \phi, \kappa, \tau, E[\text{undef}]}$$

REGISTERFUL-FULFILLED

$$\frac{\begin{array}{c} p \neq l_{time} \wedge p \neq l_{io} \qquad \pi(p) \downarrow_1 = (\text{fulfilled}, v) \\ v \neq \bot \qquad \kappa' = \kappa \cdot (p', f, [v]) \end{array}}{\pi, \phi, \kappa, \tau, E[p.\text{registerFul}(f, p', n_1, n_2, \dots, n_n)] \rightarrow \pi, \phi, \kappa', \tau, E[\text{undef}]}$$

REGISTERFUL-FULFILLED-⊥

$$\frac{\begin{array}{c} p \neq l_{time} \wedge p \neq l_{io} \qquad \pi(p) \downarrow_1 = (\text{fulfilled}, \bot) \\ \kappa' = \kappa \cdot (p', f, [n_1, n_2, \dots, n_n]) \end{array}}{\pi, \phi, \kappa, \tau, E[p.\text{registerFul}(f, p', n_1, n_2, \dots, n_n)] \rightarrow \pi, \phi, \kappa', \tau, E[\text{undef}]}$$

REGISTERFUL-TIMER-IO-⊥

$$\frac{\begin{array}{c} p = l_{time} \vee p = l_{io} \qquad \pi(p) \downarrow_1 = (\text{fulfilled}, \bot) \\ \tau' = \tau \cdot (p', f, [n_1, n_2, \dots, n_n]) \end{array}}{\pi, \phi, \kappa, \tau, E[p.\text{registerFul}(f, p', n_1, n_2, \dots, n_n)] \rightarrow \pi, \phi, \kappa, \tau', E[\text{undef}]}$$

APPEND

$$\frac{p \in dom(\pi) \qquad \phi' = p \cdot \phi}{\pi, \phi, \kappa, \tau, E[\text{append}(p)] \rightarrow \pi, \phi', \kappa, \tau, E[\text{undef}]}$$

POP

$$\frac{}{\pi, p \cdot \phi, \kappa, \tau, E[\text{pop}()] \rightarrow \pi, \phi, \kappa, \tau, E[\text{undef}]}$$

ERROR

$$\frac{\phi = p \cdot \phi'}{\pi, \phi, \kappa, \tau, E[\text{err } v] \rightarrow \pi, \phi', \kappa, \tau, E[p.\text{reject}(v)]}$$

Figure 4 The semantics of λ_q.

Remark. When we fulfill a queue object with a $\perp$ value (i.e., $v = \perp$), we do not update the arguments of the callbacks registered on the queue object p. In other words, we follow all the steps described in the [FULFILL-PENDING] rule except for creating the list t'. We omit the corresponding rule for brevity.

The [FULFILL-PEND-PEND] describes the scenario of fulfilling a pending queue object p with another pending queue object v. In this case, we do not fulfill the queue object p synchronously. Instead, we make it dependent on the queue object v given as an argument. To do so, we update the queue object v by adding p to its list of dependent queue objects (i.e., $d \cdot p$). Notice that both p and v remain pending.

The [FULFILL-PEND-FUL] rule demonstrates the case when we try to fulfill a pending queue object p with the fulfilled queue object v. Then, p resolves with the same value as the queue object v. This is expressed by the resulting expression $p.\mathsf{fulfill}(v')$.

The [FULFILL-SETTLED] rule illustrates the case when we try to fulfill a settled queue object. This rule does not update the state.

The [REGISTERFUL-PENDING] rule adds the provided callback f to the list of callbacks that we should execute once the queue object p is fulfilled. Note that this rule also associates this callback with the queue object p' given as the second argument. This means that p' is fulfilled upon the termination of f. Also, this rule adds any extra arguments passed in registerFul as the arguments of f.

The [REGISTERFUL-FULFILLED] rule adds the given callback f to the list κ (assuming that the receiver is neither l_{time} nor l_{io}). We use the fulfilled value of the receiver as the only argument of the given function. Like the previous rule, it relates the provided queue object p' with the execution of the callback. This time we do ignore any extra arguments passed in registerFul, as we fulfill the queue object p with a value that is not $\perp$.

The [REGISTERFUL-FULFILLED-$\perp$] rule describes the case where we register a callback f on a queue object fulfilled with a $\perp$ value. Unlike the [REGISTERFUL-FULFILLED] rule, this rule does not neglect any extra arguments passed in registerFul. In particular, it makes them parameters of the given callback. This distinction allows us to pass arguments explicitly to a callback. Most notably, these arguments are not dependent on the value with which a queue object is fulfilled or rejected. For example, this rules enables us to model extra arguments passed in a timer- or asynchronous I/O-related callback (e.g., `setTimeout(func, 10, arg1, arg2)`, etc.).

The [REGISTERFUL-TIMER-IO-$\perp$] rule is the same as the previous one, but this time we deal with queue objects located either at l_{time} or l_{io}. Thus, we add the given callback f to the list τ instead of κ.

The [APPEND] rule appends the element p to the front of the current queue chain. Note that this rule requires the element p to be a queue object (i.e., $p \in dom(\pi)$). On the other hand, the [POP] rule removes the top element of the queue chain.

The [ERROR] rule demonstrates the case when we encounter an uncaught exception, and the queue chain is not empty. In that case, we do not propagate the exception to the caller, but we pop the queue chain and get the top element. In turn, we reject the queue object p specified in that top element. In this way, we capture the actual behavior of the uncaught exceptions triggered during the execution of an asynchronous callback.

2.1.3 Modeling the Event Loop

A reader might wonder why do we keep two separate lists, i.e., the list τ for holding callbacks coming from the l_{time} or l_{io} queue objects, and the list κ for callbacks stemming from any other queue object. The intuition behind this design choice is that it is convenient for us to

$$\textsc{event-loop}\quad \frac{\kappa = (q, f, a) \cdot \kappa' \qquad \phi = [] \qquad \phi' = q \cdot \phi}{\pi, \phi, \kappa, \tau, E[\bullet] \to \pi, \phi', \kappa', \tau, q.\mathsf{fulfill}(E[f(a)]); \mathsf{pop}(); \bullet}$$

$$\textsc{event-loop-timers-io}\quad \frac{\begin{array}{c}\texttt{pick}\ (q, f, a)\texttt{from}\ \tau \\ \tau' = \langle \rho \ \mid \forall \rho \in \tau.\rho \neq (q, f, a)\rangle \qquad \phi = [] \qquad \phi' = q \cdot \phi\end{array}}{\pi, \phi, [], E[\bullet] \to \pi, \phi', [], \tau', q.\mathsf{fulfill}(E[f(a)]); \mathsf{pop}(); \bullet}$$

Figure 5 The semantics of the event loop.

$$\begin{array}{l} e \in \mathit{Exp} ::= \ \ldots \\ \qquad \mid \mathsf{addTimerCallback}(e_1, e_2, e_3, \ldots) \mid \mathsf{addIOCallback}(e_1, e_2, e_3, \ldots) \end{array}$$

$$\begin{array}{l} E ::= \ \ldots \\ \qquad \mid \mathsf{addTimerCallbackCallback}(E, e, \ldots) \mid \mathsf{addTimerCallback}(v, \ldots, E, e, \ldots) \\ \qquad \mid \mathsf{addIOCallback}(E, e, \ldots) \mid \mathsf{addIOCallback}(v, \ldots, E, e, \ldots) \end{array}$$

Figure 6 Extending the syntax of λ_{q} to deal with timers and asynchronous I/O.

model the concrete semantics of the event loop correctly. In particular, the implementation of the event loop assigns different priorities to the callbacks depending on their kind [25, 32]. For example, the event loop processes a callback of a promise object before any timer- or asynchronous I/O-related callback regardless of their registration order.

In this context, Figure 5 demonstrates the semantics of the event loop. The [EVENT-LOOP] rule pops the first scheduled callback from the list κ. We get the queue object q included in that callback, and we attach it to the front of the queue chain. Adding q to the top of the queue chain allows us to reject that queue object, when there is an uncaught exception during the execution of f. In this case, the evaluation of fulfill will not have any effect on the already rejected queue object q (recall the [FULLFILL-SETTLED] rule). Furthermore, observe how the event loop is reduced, i.e., $q.\mathsf{fulfill}(f(a)); \texttt{pop}(); \bullet$. Specifically, once we execute the callback f and fulfill the dependent queue object q with the return value of f, we evaluate the `pop()` expression. This means that we pop the top element of the queue chain before re-evaluating the event loop. This is an invariant of the semantics of the event loop: every time we evaluate it, the queue chain is always empty.

The [EVENT-LOOP-TIMERS-IO] rule handles the case when the list κ is empty. In other words, the rule states that when there are not any callbacks that neither come from the l_{time} nor the l_{io} queue object, inspect the list τ, and pick *non-deterministically* one of those. Selecting a callback non-deterministically allows us to over-approximate the actual behavior of the event loop regarding its different execution phases [25]. Overall, the rule describes the scheduling policy presented in the work of Loring et. al. [25], where initially we look for any promise-related callback (if any). Otherwise, we choose any callback associated with timers or asynchronous I/O at random.

2.1.4 Modeling Timers & Asynchronous I/O

To model timers and asynchronous I/O, we follow a similar approach to the work of Loring et. al. [25]. Specifically, we start with an initial queue π that contains two queue objects: the q_{time}, and q_{io} that are located at l_{time} and l_{io} respectively. Both q_{time} and q_{io} are initialized as $((\mathsf{fulfilled}, \bot), [], [], [])$. We extend the syntax of λ_{q} by adding two more expressions. Figure 6

ADD-TIMER-CALLBACK

$$\frac{q = \pi(l_{time})}{\pi, \phi, \kappa, \tau, \mathsf{addTimerCallback}(f,\ n_1, \dots) \rightarrow \pi, \phi, \kappa, q.\mathsf{registerFul}(f,\ q,\ n_1, \dots)}$$

ADD-IO-CALLBACK

$$\frac{q = \pi(l_{io})}{\pi, \phi, \kappa, \tau, \mathsf{addIOCallback}(f,\ n_1, \dots) \rightarrow \pi, \phi, \kappa, q.\mathsf{registerFul}(f,\ q,\ n_1, \dots)}$$

Figure 7 Extending the semantics of λ_q to deal with timers and asynchronous I/O.

shows the extended syntax of λ_q to deal with timers and asynchronous I/O, while Figure 7 presents the rules related to those expressions.

The new expressions have high correspondence to each other. Specifically, the $\mathsf{addTimerCallback}(\dots)$ construct adds the callback e_1 to the queue object located at the address l_{time}. The arguments of that callback are any optional parameters passed in $\mathsf{addTimerCallback}$, i.e., e_2, e_3, and so on. From Figure 7, we observe that the [ADD-TIMER-CALLBACK] rule retrieves the queue object q corresponding to the address l_{time}. Recall again that the l_{time} can be found in the initial queue. Then, the given expression is reduced to $q.\mathsf{registerFul}(f,\ q,\ n_1, \dots)$. In particular, we add the new callback f to the queue object found at l_{time}. Observe that we pass the same queue object (i.e., q) as the second argument of $\mathsf{registerFul}$. Recall from Figure 4, according to the [FULFILL-SETTLED] rule, trying to fulfill (and similarly to reject) a settled queue object does not have any effect on the state. Beyond that, since q is fulfilled with $\bot$, the extra arguments (i.e., $n_1, \dots$) are also passed as arguments in the invocation of f.

The semantics of the $\mathsf{addIOCallback}(\dots)$ primitive is the same with that of $\mathsf{addTimerCallback}(\dots)$; however, this time, we use the queue object located at l_{io}.

2.2 Expressing Promises in Terms of λ_q

The queue objects and their operations introduced in λ_q are very closely related to JavaScript promises. Therefore, the translation of promises' operations into λ_q is straightforward. We model every property and method (except for `Promise.all()`) by faithfully following the ECMAScript specification.

```
Promise.resolve = function(value) {
  var promise = newQ();
  if (typeof value.then === "function") {
    var t = newQ();
    t.fulfill(⊥);
    t.registerFul(value.then, t, promise.fulfill, promise.reject);
  } else
    promise.fulfill(value);
  return promise;
}
```

Figure 8 Expressing `Promise.resolve` in terms of λ_q.

Example – Modeling Promise.resolve(). In Figure 8, we see how we model the `Promise.resolve()` function in terms of λ_q[3]. The JavaScript `Promise.resolve()` function creates a new promise, and resolves it with the given value. According to ECMAScript, if the given `value` is a *thenable*, (i.e., an object that has a property named "then" and that property is a callable), the created promise resolves asynchronously. Specifically, we execute the function `value.then()` asynchronously, and we pass the resolving functions (i.e., fulfill, reject) as its arguments. Observe how the expressiveness of λ_q can model this source of asynchrony (lines 4–6), which we cannot model through the previous work [26, 25]. First, we create a fresh queue object `t`, and we fulfill it with $\perp$ (lines 4, 5). Then, at line 6, we schedule the execution of `value.then()` by registering it on the newly created queue object `t`. Notice that we also pass `promise.fulfill` and `promise.reject` as extra arguments. This means that those functions will be the actual arguments of `value.then()` because `t` is fulfilled with $\perp$. On the other hand, if `value` is not a thenable, we synchronously resolve the created promise using the `promise.fulfill` construct at line 8.

3 The Core Analysis

The λ_q calculus presented in Section 2 is the touchstone of the static analysis proposed for asynchronous JavaScript programs. The analysis is designed to be sound; thus, we devise abstract domains and semantics that over-approximate the behavior of λ_q. Currently, there are few implementations available for asynchronous JavaScript, and previous efforts mainly focus on modeling the event system of client-side applications [18, 33]. To the best of our knowledge, it is the first static analysis for ES6 promises. The rest of this section describes the details of the analysis.

3.1 The Analysis Domains

$$
\begin{aligned}
l \in \widehat{Addr} &= \{l_i \mid i \text{ is an allocation site}\} \cup \{l_{time}, l_{io}\} \\
\pi \in \widehat{Queue} &= \widehat{Addr} \hookrightarrow \mathcal{P}(\widehat{QueueObject}) \\
q \in \widehat{QueueObject} &= \widehat{QueueState} \times \mathcal{P}(\widehat{Callback}) \times \mathcal{P}(\widehat{Callback}) \times \mathcal{P}(\widehat{Addr}) \\
qs \in \widehat{QueueState} &= \{\mathsf{pending}\} \cup (\{\mathsf{fulfilled}, \mathsf{rejected}\} \times Value) \\
clb \in \widehat{Callback} &= \widehat{Addr} \times F \times Value^{*} \\
\kappa \in \widehat{ScheduledCallbacks} &= (\mathcal{P}(\widehat{Callback}))^{*} \\
\tau \in \widehat{ScheduledTimerIO} &= (\mathcal{P}(\widehat{Callback}))^{*} \\
\phi \in \widehat{QueueChain} &= (\mathcal{P}(\widehat{Addr}))^{*}
\end{aligned}
$$

Figure 9 The abstract domains of λ_q.

[3] For brevity, Figure 8 omits some steps described in the specification of the `Promise.resolve()` function. For example, according to ECMAScript, if `this` value of the `Promise.resolve()` function is not an object, then a `TypeError` is thrown. In the implementation, though, we follow all the steps that are described in the specification.

Figure 9 presents the abstract domains of the $\lambda_{\texttt{q}}$ calculus that underpin our static analysis. Below we make a summary of our primary design choices.

Abstract Addresses: As a starting point, we employ allocation site abstraction for modeling the space of addresses. It is the standard way used in literature for abstracting addresses that keeps the domain finite [19, 27]. Notice that we still define two internal addresses, i.e., l_{time}, l_{io}, corresponding to the addresses of the queue objects responsible for timers and asynchronous I/O respectively.

Abstract Queue: We define an abstract queue as the partial map of abstract addresses to an element of the power set of abstract queue objects. Therefore, an address might point to multiple queue objects. This abstraction over-approximates the behavior of $\lambda_{\texttt{q}}$ and allows us to capture all possible program's behaviors that might stem from the analysis imprecision.

Abstract Queue Objects: A tuple consisting of an abstract queue state – observe that the domain of abstract queue states is the same as $\lambda_{\texttt{q}}$ – two sets of abstract callbacks (executed on fulfillment and rejection respectively), and a set of abstract addresses (used to store the queue objects that are dependent on the current one) represents an abstract queue object. Notice how this definition differs from that of $\lambda_{\texttt{q}}$. First, we do not keep the registration order of callbacks; therefore, we convert the two lists into two sets. The programming pattern related to promises supports our design decision. Specifically, developers often use promises as a chain; registering two callbacks on the same promise object is quite uncommon. Madsen et. al. [27] made similar observations for the event-driven programs.

This abstraction can negatively affect precision only when we register multiple callbacks on a *pending* queue object. Recall from Figure 4, when we register a callback on a settled queue object, we can precisely track its execution order since we directly add it to the list of scheduled callbacks.

Finally, we define the last component of abstract queue objects as a set of addresses; something that enables us to track all possible dependent queue objects soundly.

Abstract Callback: An abstract callback comprises one abstract address, one function, and a list of values that stands for the arguments of the function. Recall that the abstract address corresponds to the queue object that the return value of the function fulfills.

Abstract List of Scheduled Callbacks: We use a list of sets to abstract the domain that is responsible for maintaining the callbacks that are ready for execution (i.e., $\widehat{ScheduledCallbacks}$ and $\widehat{ScheduledTimerIO}$). In this context, the i^{th} element of a list denotes the set of callbacks that are executed after those placed at the $(i-1)^{th}$ position and before the callbacks located at the $(i+1)^{th}$ position of the lists. The execution of callbacks of the same set is not known to the analysis; they can be called in any order. For example, consider the following sequence $[\{x\}, \{y,\ z\}, \{w\}]$, where $x, y, z, w \in \widehat{Callback}$. We presume that the execution of elements y, z succeeds that of x, and precedes that of w, but we cannot compare y with z, since they are elements of the same set; thus, we might execute y before z and vice versa.

Note that a critical requirement of our domains' definition is that they should be finite so that the analysis is guaranteed to terminate. Keeping the lists of scheduled callbacks bound is tricky because the event loop might process the same callback multiple times. Therefore, we have to add it to the lists κ or τ more than one time. For that reason, those lists monitor the execution order of callbacks up to a certain limit n. The execution order of the callbacks scheduled after that limit is not preserved; thus, the analysis places them into the same set.

Abstract Queue Chain: The analysis uses the last component of our abstract domains to capture the effects of uncaught exceptions during the execution of callbacks. We define it as a sequence of sets of addresses. Based on the abstract translation of the semantics of $\lambda_{\texttt{q}}$, when the analysis reaches an uncaught exception, it inspects the top element of the abstract queue chain and rejects all the queue objects found in that element. If the abstract queue chain is empty, the analysis propagates the exception to the caller function as usual. Note that the queue chain is guaranteed to be bound. In particular, during the execution of a callback, the size of the abstract queue chain is always one because the event loop executes only one callback at a time. The only case when the abstract queue chain contains multiple elements is when we have nested promise executors. A promise executor is a function passed as an argument in a promise constructor. However, since we cannot have an unbound number of nested promise executors, the size of the abstract queue chain remains finite.

3.1.1 Tracking the Execution Order

Promises. Estimating the order in which the event loop executes promise-related callbacks is straightforward because it is a direct translation of the corresponding semantics of $\lambda_{\mathtt{q}}$. In particular, there are two possible cases:

- *Settle a promise that has registered callbacks:* When we settle (i.e., either fulfill or reject) a promise object that has registered callbacks, we schedule those callbacks associated with the next state of the promise by putting them on the tail of the list κ. For instance, if we fulfill a promise, we append all the callbacks triggered on fulfillment on the list κ. A reader might observe that when there are multiple callbacks registered on the same promise object, we put them on the same set which is the element that we finally add to κ. This is justified by the fact that an abstract queue object does not keep the registration order of its callbacks.
- *Register a callback on an already settled promise:* When we encounter a statement of the form `x.then(f1, f2)`, where `x` is a settled promise, we schedule either callback `f1` or `f2` (i.e., we add it to the list κ) depending on the state of that promise, i.e., we schedule the callback `f1` if `x` is fulfilled and `f2` if `x` is rejected.

Timers & Asynchronous I/O. A static analysis is not able to reason about the external environment. For instance, it cannot decide when an operation on a file system or a request to a server is complete. Similarly, it is not able to deal with time. For that purpose, we adopt a conservative approach for tracking the execution order between callbacks related to timers and asynchronous I/O. In particular, we assume that the execution order between those callbacks is unspecified; thus, the event loop might process them in any order. However, we *do* keep track the execution order between nested callbacks.

3.2 Callback Graph

In this section, we introduce the concept of *callback graph*; a fundamental component of our analysis that captures how data flow is propagated between different asynchronous callbacks. A callback graph is defined as an element of the following power set:

$$cg \in \mathit{CallbackGraph} = \mathcal{P}(\mathit{Node} \times \mathit{Node})$$

We define every node of a callback graph as $n \in \mathit{Node} = C \times F$, where C is the domain of contexts while F is the set of all the functions of the program. Every element of a callback graph $(c_1, f_1, c_2, f_2) \in cg$, where $cg \in \mathit{CallbackGraph}$ has the following meaning: *the function f_2 in context c_2 is executed after the function f_1 in context c_1*. We can treat the above statement as the following expression: $f_1(\dots); f_2(\dots);$

▶ **Definition 1.** *Given a callback graph $cg \in \mathit{CallbackGraph}$, we define the binary relation $\rightarrow_{cg}$ on nodes of the callback graph $n_1, n_2 \in \mathit{Node}$ as:*

$$n_1 \rightarrow_{cg} n_2 \Rightarrow (n_1, n_2) \in cg$$

▶ **Definition 2.** *Given a callback graph $cg \in \mathit{CallbackGraph}$, we define the binary relation $\rightarrow^{+}_{cg}$ on nodes of the callback graph $n_1, n_2 \in \mathit{Node}$ as the transitive closure of $\rightarrow_{cg}$:*

$$n_1 \rightarrow_{cg} n_2 \Rightarrow n_1 \rightarrow^{+}_{cg} n_2$$
$$n_1 \rightarrow^{+}_{cg} n_2 \wedge n_2 \rightarrow^{+}_{cg} n_3 \Rightarrow n_1 \rightarrow^{+}_{cg} n_3, \quad \textit{where } n_3 \in \mathit{Node}$$

Definition 1 and Definition 2 introduce the concept of path between two nodes in a callback graph $cg \in \mathit{CallbackGraph}$. In particular, the relation $\rightarrow_{cg}$ denotes that there is path of length one between two nodes n_1, n_2, i.e., $(n_1, n_2) \in cg$. On the other hand, the relation $\rightarrow^{+}_{cg}$ describes that there is a path of unknown length between two nodes. Relation $\rightarrow^{+}_{cg}$ is very important as it allows us to identify the *happens-before* relation between two nodes n_1, n_2 even if n_2 is executed long after n_1, that is $(n_1, n_2) \notin cg$. A property of a callback graph is that it does not have any cycles, i.e.,

$$\forall n_1, n_2 \in \mathit{Node}.\, n_1 \rightarrow^{+}_{cg} n_2 \Rightarrow n_2 \not\rightarrow^{+}_{cg} n_1$$

Notice that if $n_1 \not\rightarrow^{+}_{cg} n_2$, and $n_2 \not\rightarrow^{+}_{cg} n_1$ hold, the analysis cannot estimate the execution order between n_1 and n_2. Therefore, we presume that n_1 and n_2 can be called in any order.

Callback graph is computed on the fly as the analysis progresses. Callback graph exploits both the lists κ and τ, and constructs the $\rightarrow_{cg}$ relations between callbacks by respecting their execution order as specified in those lists.

Previous work has proposed similar program representations for asynchrony. Madsen et. al. [27] introduce the event-based call graph that abstracts the data-flow of the event-based JavaScript programs. However, it does not support promises. More recently, promise graph [26, 1] has been used for debugging promise-related programs. Our callback graph is distinguished from promise graph, as it also captures callbacks that stem from timers or asynchronous I/O. Therefore, we can handle common programming patterns where we mix promises with asynchronous I/O (Figure 1). Also, since the promise graph aims to detect anti-patterns related to promise code (e.g., unsettled promises), it does not track the order in which promises are settled. Therefore, it misses the happens-before relations between the corresponding callbacks.

3.3 Analysis Sensitivity

Here, we introduce two methods for boosting the analysis precision of asynchronous code.

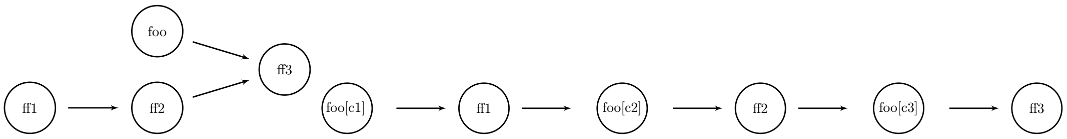
3.3.1 Callback Sensitivity

Knowing the temporal relations between asynchronous callbacks enables us to capture how data flow is propagated precisely. Typically, a naive flow-sensitive analysis, which exploits the control flow graph (CFG), represents the event loop as a single program point with only one context corresponding to it. Therefore – unlike traditional function calls – the analysis misses the happens-before relations between callbacks because they are triggered by the same program location (i.e., the event loop).

To address those issues, we exploit the callback graph to devise a more precise analysis, which we call *callback-sensitive* analysis. The callback-sensitive analysis propagates the state with regards to the $\rightarrow_{cg}$ and $\rightarrow^{+}_{cg}$ relations found in a callback graph $cg \in \mathit{CallbackGraph}$. Specifically, when the analysis needs to propagate the resulting state from the exit point of a

```
function foo() { ... }

var x = Promise.resolve()
  .then(foo)
  .then(function ff1() { ... })
  .then(foo)
  .then(function ff2() { ... })
  .then(foo)
  .then(function ff3() { ... });
```

Figure 10 An example program where we create a promise chain. Notice that we register the function `foo` multiple times across the chain.



(a) QR-insensitive analysis **(b)** QR-sensitive analysis

Figure 11 Callback graph of program of Figure 10 produced by the QR-insensitive and QR-sensitive analysis respectively.

callback x, instead of propagating that state to the caller (note that the caller of a callback is the event loop), it propagates it to the entry points of the next callbacks, i.e., all callback nodes $y \in Node$ where $x \to_{cg} y$ holds. In other words, the edges of a callback graph reflect how the state is propagated from the exit point of a callback node x to the entry point of a callback node y. Obviously, if there is not any path between two nodes in the graph, that is, $x \not\to^{+}_{cg} y$, and $y \not\to^{+}_{cg} x$, we propagate the state coming from the exit point of x to the entry point of y and vice versa.

Remark. Callback-sensitivity does not work with contexts to improve the precision of the analysis. We still represent the event loop as a single program point. As a result, the state produced by the last executed callbacks is propagated to the event loop, leading to the join of this state with the initial one. The join of those states is then again propagated across the nodes of the callback graph until convergence. Therefore, there is still some imprecision. However, callback-sensitivity minimizes the number of those joins, as they are only caused by the callbacks invoked last.

3.3.2 Context-Sensitivity

Recall from Section 3.2 that a callback graph is defined as $\mathcal{P}(Node \times Node)$, where $n \in Node = C \times F$. It is possible to increase the precision of a callback graph by distinguishing callbacks based on the context in which they are invoked. Existing flavors of context-sensitivity are not so useful in differentiating asynchronous functions from each other. For instance, object-sensitivity [30, 24], which separates invocations based on the value of the receiver – and has been proven to be particularly effective for the analysis of object-oriented languages – is not fruitful in the context of asynchronous callbacks because in most cases the receiver of callbacks corresponds to the global object. Similarly, previous work in the static analysis of JavaScript [19, 21] creates a context with regards to the arguments of a function. Such a strategy might not be effective in cases where a callback expects no arguments or the arguments from two different calls are indistinguishable.

We introduce one novel context-sensitivity flavor – which we call *QR-sensitivity* – as an effort to boost the analysis precision. QR-sensitivity separates callbacks according to: (1) the queue object that they belong to (Q), and (2) the queue object their return value fulfills (R). In this case, the domain of contexts is given by:

$$c \in C = \widehat{Addr} \times \widehat{Addr}$$

In other words, every context is a pair $(l_q, l_r) \in \widehat{Addr} \times \widehat{Addr}$, where l_q stands for the allocation site of callback's queue object, and l_r is the abstract address of the queue object that the return value of the callback fulfills. Notice that this domain is finite, so the analysis always terminates.

As a motivating example, consider the program of Figure 10. This program creates a promise chain where we register different callbacks at every step of the asynchronous computation. At line 1, we define the function `foo()`. We asynchronously call `foo()` multiple times, i.e., at lines 4, 6, and 8. Recall that chains of promises enable us to enforce a deterministic execution of the corresponding callbacks. Specifically, based on the actual execution, the event loop invokes the callbacks in the following order: `foo()` → `ff1()` → `foo()` → `ff2()` → `foo()` → `ff3()`. Figure 11a presents the callback graph of the program of our example produced by a QR-insensitive analysis. In this case, the analysis considers the different invocations of `foo()` as identical. As a result, the analysis loses the temporal relation between `foo()` and `ff1()`, `ff2()` – indicated by the fact that the respective nodes are not connected to each other – because `foo()` is called both before and after `ff1()` and `ff2()`. On the contrary, a QR-sensitive analysis ends up with an entirely precise callback graph as shown in Figure 11b. The QR-sensitive analysis distinguishes the different invocations of `foo()` from each other because it creates three different contexts; one for every call of `foo()`. Specifically, we have $c_1 = (l_3, l_4), c_2 = (l_5, l_6), c_3 = (l_7, l_8)$, where l_i stands for the promise object allocated at line i. For example, the second invocation of `foo()` is related to the promise object created by the call of `then()` at line 5, and its return value fulfills the promise object allocated by the invocation of `then()` at line 6.

3.4 Implementation

Our prototype implementation[4] extends *TAJS* [19, 20, 18]; a state-of-the-art static analyzer for JavaScript. TAJS analysis is implemented as an instance of the abstract interpretation framework [3], and it is designed to be sound. It uses a lattice specifically designed for JavaScript that is capable of handling the vast majority of JavaScript's complicated features and semantics. TAJS analysis is both flow- and context-sensitive. The output of the analysis is the set of all reachable states from an initial state along with a call graph. TAJS can detect various type-related errors such as the use of a non-function variable in a call expression, property access of `null` or `undefined` variables, inconsistencies caused by implicit type conversions, and many others [19].

Prior to our extensions, TAJS consisted of approximately 83,500 lines of Java code. The size of our additions is roughly 6,000 lines of Java code. Our implementation is straightforward and is guided by the design of our analysis. Specifically, we first incorporate the domains presented in Figure 9 into the definition of the abstract state of TAJS. Then, we provide models for promises written in Java by faithfully following the ECMAScript specification. Recall again that our models exploit the λ_{q} calculus presented in Section 2 and they produce

[4] `https://github.com/theosotr/async-tajs`

```
 1  function open(filename, flags, mode, callback) {
 2     TAJS_makeContextSensitive(open, 3);
 3     var err = TAJS_join(TAJS_make("Undef"), TAJS_makeGenericError());
 4     var fd = TAJS_join(TAJS_make("Undef"), TAJS_make("AnyNum"));
 5     TAJS_addAsyncIOCallback(callback, err, fd);
 6  }
 7
 8  var fs = {
 9    open: open
10    ...
11  }
```

Figure 12 A model for `fs.open` function. All functions starting with `TAJS_` are special functions whose body does not correspond to any node in the CFG. They are just hooks for producing side-effects to the state or evaluating to some value, and their models are implemented in Java. For instance, `TAJS_make("AnyStr")` evaluates to a value that can be any string.

side-effects that over-approximate the behavior of JavaScript promises. Beyond that, we implement models for the special constructs of λ_{q} (i.e., addTimerCallback, addIOCallback) that are used for adding callbacks to the timer- and asynchronous I/O-related queue objects respectively. We implement the models for timers in Java; however, we write JavaScript models for asynchronous I/O operations, when it is necessary.

For example, Figure 12 shows the JavaScript code that models the function `open()` of the `fs` Node.js module. In particular, `open()` asynchronously opens a given file. When I/O operation completes, the callback provided by the developer is called with two arguments: (1) `err` that is not `undefined` when there is an error during I/O, (2) `fd` which is an integer indicating the file descriptor of the opened file. Note that `fd` is `undefined`, when any error occurs. Our model first makes `open()` parameter-sensitive on the third argument that corresponds to the callback provided by the programmer. Then, at lines 3 and 4, it initializes the arguments of the callback, (i.e., `err` and `fd`). Observe that we initialize those arguments so that they capture all the possible execution scenarios, i.e., `err` might be `undefined` or point to an error object, and `fd` might be `undefined` or any integer reflecting all possible file descriptors. Finally, at line 5, we call the special function `TAJS_addAsyncIOCallback()` that registers the given callback on the queue object responsible for I/O operations, implementing the semantics of the addIOCallback primitive from our λ_{q} calculus.

3.5 Limitations

Although our analysis aims to support all the asynchronous features of JavaScript up to the 7th version of ECMAScript, it does not handle the `Promise.all()` function of the Promise API. This function expects an iterable of promises, and it creates a new object that is fulfilled whenever all promises included in that iterable are fulfilled. Statically capturing all program's behaviors that stem from `Promise.all()` is challenging because the analysis imprecision might cause the number of behaviors to grow exponentially. However, `Promise.all()` is less common than other functions of the Promise API such as `Promise.resolve()` or `Promise.reject()`.

Some of our design choices about analysis abstractions might lead to imprecision. For example, we do not track the registration order of a pending promise's callbacks. Therefore, when we settle such a promise, the analysis assumes that all its registered callbacks can be invoked in any order. However, as we mentioned in Section 3.1.1, that programming pattern (i.e., adding multiple callbacks to the same pending object) is quite rare.

Table 1 List of the selected macro-benchmarks and their description. Each benchmark is described by its lines of code (LOC), its lines of code including its dependencies (ELOC), number of files, number of dependencies, number of promise-related statements (e.g., `Promise.resolve()`, `Promise.reject()`, `then()`, etc.), and number of statements associated with timers (e.g., `setTimeout()`, `setImmediate()`, etc.) or asynchronous I/O (e.g., asynchronous file system or network operations etc.).

Benchmark	LOC	ELOC	Files	Dependencies	Promises	Timers/Async I/O
controlled-promise	225	225	1	0	4	1
fetch	517	1,118	1	1	12	4
honoka	324	1,643	6	6	4	1
axios	1,733	1,733	26	0	13	2
pixiv-client	1,031	3,469	1	2	64	2
node-glob	1,519	6,131	3	6	0	5

The analysis sensitivity options introduced in Section 3.3.2 might not be so effective when dealing with timers or asynchronous I/O. Since we follow a conservative approach for modeling the execution order of timers and asynchronous I/O – regardless of the registration order of their callbacks – keeping a more precise state does not necessarily lead to a more precise callback graph.

4 Evaluation

In this section, we evaluate our static analysis on a set of hand-written micro-benchmarks and a set of real-world JavaScript modules. Then, we experiment with different parameterizations of the analysis, and report the precision and performance metrics.

4.1 Experimental Setup

To test that our technique behaves as expected we first wrote a number of micro-benchmarks. Each of those programs consists of approximately 20–50 lines of code and examines certain parts of the analysis. Beyond micro-benchmarks, we evaluate our analysis on 6 real-world JavaScript modules. The most common macro-benchmarks for static analyses used in the literature are those provided by JetStream[5], and V8 engine[6][19, 21, 22]. However, those benchmarks are not suitable for our case because they are not asynchronous. To find interesting benchmarks, we developed an automatic mechanism for collecting and analyzing Github repositories. First, we collected a large number of Github repositories using two different options. The first option extracted the Github repositories of the most depended-upon `npm` packages[7]. The second option employed the Github API[8] to find JavaScript repositories that are related to promises. We then investigated the Github repositories that we collected at the first phase by computing various metrics such as lines of code, number of promise-, timer- and asynchronous IO-related statements. We manually selected the 6 JavaScript modules presented in Table 1. Most of them are libraries for performing HTTP requests or file system operations.

[5] `https://browserbench.org/JetStream/`
[6] `http://www.netchain.com/Tools/v8/`
[7] `https://www.npmjs.com/browse/depended`
[8] `https://developer.github.com/v3/`

Table 2 Precision on micro-benchmarks.

	Analyzed Callbacks				Callback Graph Precision				Type Errors			
Benchmark	**NC-No**	**NC-QR**	**C-No**	**C-QR**	**NC-No**	**NC-QR**	**C-No**	**C-QR**	**NC-No**	**NC-QR**	**C-No**	**C-QR**
micro01	5	5	4	4	0.8	0.8	1.0	1.0	2	2	0	0
micro02	3	3	3	3	1.0	1.0	1.0	1.0	1	1	0	0
micro03	2	2	2	2	1.0	1.0	1.0	1.0	1	1	0	0
micro04	4	4	4	4	0.5	0.5	0.5	0.5	1	1	1	1
micro05	8	8	7	7	0.96	0.96	1.0	1.0	3	3	0	0
micro06	11	11	11	11	1.0	1.0	1.0	1.0	3	3	1	1
micro07	14	14	13	13	0.86	0.87	1.0	1.0	1	1	0	0
micro08	5	5	5	5	0.8	0.8	0.8	0.8	1	1	0	0
micro09	5	5	4	4	0.9	0.9	1.0	1.0	1	1	0	0
micro10	3	3	3	3	1.0	1.0	1.0	1.0	1	1	1	1
micro11	4	4	4	4	0.83	0.83	0.83	0.83	5	5	5	5
micro12	5	5	5	5	0.9	0.9	1.0	1.0	2	2	0	0
micro13	4	4	3	3	0.83	0.83	1.0	1.0	1	1	0	0
micro14	6	6	5	5	0.8	0.8	1.0	1.0	2	2	0	0
micro15	6	6	6	6	0.8	0.8	1.0	1.0	0	0	0	0
micro16	6	6	6	6	1.0	1.0	1.0	1.0	1	1	0	0
micro17	3	3	3	3	0.67	0.67	0.67	0.67	2	2	2	2
micro18	4	3	4	3	0.83	1.0	0.83	1.0	1	0	1	0
micro19	14	7	14	7	0.73	0.93	0.74	1.0	0	0	0	0
micro20	6	6	6	6	0.93	0.93	1.0	1.0	0	0	0	0
micro21	5	5	4	4	0.9	0.9	1.0	1.0	1	1	0	0
micro22	6	6	5	5	0.87	0.87	0.9	0.9	1	1	0	0
micro23	6	6	5	5	0.87	0.87	1.0	1.0	3	3	1	1
micro24	3	3	3	3	1.0	1.0	1.0	1.0	2	2	1	1
micro25	8	8	8	8	0.79	0.79	0.79	0.79	1	1	0	0
micro26	9	9	7	7	0.89	0.89	1.0	1.0	3	3	1	1
micro27	3	3	3	3	1.0	1.0	1.0	1.0	1	1	1	1
micro28	7	7	7	7	0.81	0.81	0.81	0.81	1	1	1	1
micro29	4	4	4	4	0.5	1.0	0.5	1.0	0	0	0	0
Average	**5.83**	**5.55**	**5.45**	**5.17**	**0.85**	**0.88**	**0.91**	**0.94**	**1.45**	**1.41**	**0.55**	**0.52**
Total	**169**	**161**	**158**	**150**					**42**	**41**	**16**	**15**

We experiment with 4 different analyses: (1) an analysis that is neither callback- nor QR-sensitive (NC-No), (2) a callback-insensitive but QR-sensitive analysis (NC-QR), (3) a callback-sensitive but QR-insensitive analysis (C-No), and (4) a both callback- and QR-sensitive analysis (C-QR). Note that recency abstraction [2] – which is a technique for minimizing weak updates and is natively supported by TAJS – is enabled for every analysis. For every analysis, we use object-sensitivity, while we enable parameter-sensitivity in certain functions for further boosting the precision of top-level code. Finally, the lists κ and τ are bounded by $n = 30$.

We evaluate the precision of each analysis in terms of the number of the analyzed callbacks, the precision of the computed callback graph, and the number of reported type errors triggered by the execution of asynchronous callbacks. We define the precision of a callback graph as the quotient between the number of callback pairs whose execution order is determined and the total number of callback pairs. Also, we embrace a client-based precision metric, i.e., the number of reported type errors as in the work of Kashyap et. al. [21]. The fewer type errors an analysis reports, the more precise it is. The same applies to the number of callbacks inspected by the analysis. To compute the performance characteristics of every analysis, we run every experiment ten times to get reliable measurements. All the experiments were run on a machine with an Intel i7 2.4GHz quad-core processor and 8GB of RAM.

Table 3 Precision on macro-benchmarks.

	Analyzed Callbacks				Callback Graph Precision				Type Errors			
Benchmark	**NC-No**	**NC-QR**	**C-No**	**C-QR**	**NC-No**	**NC-QR**	**C-No**	**C-QR**	**NC-No**	**NC-QR**	**C-No**	**C-QR**
controlled-promise	6	6	6	6	0.866	0.905	0.866	0.905	3	3	2	2
fetch	22	22	19	19	0.829	0.956	0.822	0.972	8	8	7	7
honoka	8	8	6	6	0.929	0.929	1.0	1.0	1	1	0	0
axios	15	15	14	14	0.678	0.83	0.686	0.871	2	2	1	1
pixiv-client	18	18	17	15	0.771	0.803	0.794	0.863	3	3	3	2
node-glob	3	3	3	3	0.667	0.667	0.667	0.667	19	19	19	19
Average	**12**	**12**	**10.8**	**10.5**	**0.79**	**0.848**	**0.805**	**0.88**	**6**	**6**	**5.1**	**5**
Total	**72**	**72**	**65**	**63**					**36**	**36**	**32**	**31**

4.2 Results

Micro-benchmarks. Table 2 shows how precise every analysis is on every micro-benchmark. Starting with callback-insensitive analyses (see the NC-No and NC-QR columns), we observe that in general QR-sensitivity improves the precision of the callback graph by 3.6%, on average. That small boost of QR-sensitivity is explained by the fact that *only* 3 out of 29 micro-benchmarks invoke the same callback multiple times.

Recall from Section 3.3.2 that QR-sensitivity is used to distinguish different calls of the same callback. Therefore, if one program does not use a specific callback multiple times, QR-sensitivity does not make any difference. However, if we focus on the results of the micro-benchmarks where we come across such behaviors, i.e. micro18, micro19, and micro29, we get a significant divergence of the precision of callback graph. Specifically, QR-sensitivity improves the precision by 20.5%, 27.4% and 100% in micro18, micro19 and micro29 respectively. Besides that, in micro19, there is a striking decrease in the number of the analyzed callbacks: the QR-insensitive analyses inspect 14 callbacks, while the QR-sensitive analyses examine only 7.

The results regarding the number of type errors are almost identical for every analysis: a QR-insensitive analysis reports 42 type errors in total, whereas all the other QR-sensitive analyses produce warnings for 41 cases.

Moving to callback-sensitive analyses, the results indicate clear differences. First, a callback-sensitive but QR-insensitive analysis reports only 16 type errors in total (i.e., 61.9% fewer type errors than callback-insensitive analyses), and amplifies the average precision of the callback graph from 0.85 to 0.91. As before, the QR-sensitive analyses boost the precision of the callback graph by 20.4%, 35.1%, and 100% in micro18, micro19, and micro29 respectively. Finally, a callback-sensitive and QR-insensitive analysis decreases the total number of the analyzed callbacks from 169 to 158. Notice that when callback- and QR-sensitivity are combined, the total number of callbacks is reduced by 11.2%.

Macro-benchmarks. Table 3 reports the precision metrics of every analysis on the macro-benchmarks. First, we make similar observations as those of micro-benchmarks. In general, QR-sensitivity leads to a more precise callback graph for 4 out of 6 benchmarks. The improvement ranges from 4.6% to 26.9%. On the other hand, callback-sensitive analyses contribute to fewer type errors for 5 out of 6 benchmarks reporting 13.9% fewer type errors in total. Additionally, when we combine QR- and callback-sensitivity, we can boost the analysis precision for 5 out of 6 benchmarks. Specifically, the QR- and callback-sensitive analysis improves the callback graph precision by up to 28.5% (see the `axios` benchmark), and achieves a 88% callback graph precision, on average. On the other hand, the naive analysis (neither QR- nor callback-sensitive) reports only a 79% precision for callback graph, on average.

Table 4 Times of different analyses in seconds.

	Average Time				Median			
Benchmark	**NC-No**	**NC-QR**	**C-No**	**C-QR**	**NC-No**	**NC-QR**	**C-No**	**C-QR**
controlled-promise	2.3	2.22	2.27	2.28	2.29	2.26	2.25	2.31
fetch	8.53	7.97	7.07	6.98	8.52	8.26	7.46	7.22
honoka	4.14	4.05	3.86	3.94	4.12	4.0	3.61	3.81
axios	6.99	7.86	6.74	8.32	7.02	8.0	6.94	8.37
pixiv-client	22.11	24.92	24.77	28.89	22.19	25.16	24.65	29.2
node-glob	15.55	16.71	15.46	14.47	16.62	16.71	16.17	15.74

By examining the results for the `node-glob` benchmark, we see that every analysis produces identical results. `node-glob` uses only timers and asynchronous I/O operations. As we mentioned in Section 3.5, in such cases, neither callback- nor QR-sensitivity is effective as we follow a conservative approach for modelling timers and asynchronous I/O. For example, we assume that two callbacks x and y are executed in any order, even if x is scheduled before y (and vice versa).

Table 4 gives the running times of every analysis on macro-benchmarks. We notice that in some benchmarks (see fetch) a more precise analysis decreases the running times by 3%–18%. This is justified by the fact that a more precise analysis might compress the state faster than an imprecise analysis. For instance, in fetch, an imprecise analysis led to the analysis of 3 spurious callbacks, yielding to a higher analysis time. The results appear to be consistent with those of the recent literature that suggest that precision might lead to a faster analysis in some cases [33]. On the other hand, we observe a non-trivial fall in the analysis performance in only one benchmark. Specifically, the analysis sensitivity increased the running times of `pixiv-client` by 12%–30.6%. However, such an increase seems to be acceptable.

4.3 Case Studies

In this section, we describe some case studies coming from the macro-benchmarks.

fetch. Figure 13 shows a code fragment taken from fetch[9]. Note that we omit irrelevant code for brevity. The function `Body()` defines a couple of methods (e.g., `text()`, `formData()`) for manipulating the body of a response. Observe that those methods are registered on the prototype of `Response` using the function `Function.prototype.call()` at line 45. Note that `Body` also contains a method (i.e., `_initBody()`) for initializing the body of a response according to the type of the input. To this end, the `Response` constructor takes a body as a parameter and initializes it through the invocation of `_initBody()` (lines 41, 43). The function `text()` reads the body of a response in a text format (lines 20–34). When the body of the response has been already read, `text()` returns a rejected promise (lines 3, 22–23). Otherwise, it marks the property `bodyUsed` of the response object as true (line 5), and then it returns a promise object depending on the type of the body of the given response (lines 25–33). The function `formData()` (lines 36–38) asynchronously reads the body of a response in a text format, and then it parses it into a `FormData` object[10] through the call of the function `decode()`. The function `fetch()` (lines 47–56) makes a new asynchronous request.

[9] `https://github.com/github/fetch`

[10] `https://developer.mozilla.org/en-US/docs/Web/API/FormData`

```
1  function consumed(body) {
2      if (body.bodyUsed) {
3          return Promise.reject(new TypeError("
               Already read"));
4      }
5      body.bodyUsed = true;
6  }
7  ...
8  function Body() {
9    ...
10   this.bodyUsed = false;
11   this._bodyInit = function() {
12     ...
13     if (typeof body === "string") {
14           this._bodyText = body;
15     } else if (Blob.prototype.isPrototypeOf(
           body)) {
16         this._bodyBlob = body;
17     }
18     ...
19   }
20   this.text = function text() {
21         var rejected = consumed(this);
22         if (rejected) {
23             return rejected;
24         }
25         if (this._bodyBlob) {
26             return readBlobAsText(
                   this._bodyBlob);
27         } else if (this._bodyArrayBuffer) {
28             return Promise.resolve(
                   readArrayBufferAsText(
                   this._bodyArrayBuffer));
29         } else if (this._bodyFormData) {
30             throw new Error("could not read
                   FormData body as text");
31         } else {
32             return Promise.resolve(
                   this._bodyText);
33         }
34     };
35   ...
36   this.formData = function formData() {
37       return this.text().then(decode);
38   }
39 }
40 ...
41 function Response(body) {
42   ...
43   this._bodyInit(body);
44 }
45 Body.call(Response.prototype);
46 ...
47 function fetch(input, init) {
48   return new Promise(function (resolve, reject
         ) {
49     ...
50     var xhr = new XMLHttpRequest();
51     xhr.onload = function onLoad() {
52       ...
53       resolve(new Response(xhr.response));
54     }
55   });
56 }
```

Figure 13 Code fragment taken from fetch.

Listing 1 Case 1.

```
1      fetch("/helloWorld").then(function foo(
             value) {
2        var formData = value.formData();
3        // Do something with form data.
4      })
```

Listing 2 Case 2.

```
1      var response = new Response("foo=bar");
2      var formData = response.formData();
3      var response2 = new Response(new Blob("foo=
             bar"));
4      var formData2 = response2.formData();
```

Figure 14 Code fragments which use the `fetch` API.

When the request completes successfully, the callback `onLoad()` is executed asynchronously (line 51). This callback finally fulfills the promise returned by `fetch()` with a response object that contains the response of the server (line 53).

In Listing 1, we make an asynchronous request to the endpoint "/helloWorld" using the `fetch` API. Upon success, we schedule the callback `foo()`. Recall that the parameter `value` of `foo()` corresponds to the response object coming from line 53 (Figure 13). In `foo()`, we convert the response of the server into a `FormData` object (line 2). A callback-insensitive analysis, which considers that the event loop executes all callbacks in any order, merges all the data flow stemming from those callbacks into a single point. As a result, the side effects of `onLoad()` and `foo()` are directly propagated to the event loop. In turn, the event loop propagates the resulting state again to those callbacks. This is repeated until convergence. Specifically, the callback `foo()` calls `value.formData()` that updates the property `bodyUsed` of the response object to true (Figure 13, line 5). The resulting state is propagated to the event loop where is joined with the state that stems from the callback `onLoad()`. Notice that the state of `onLoad()` indicates that `bodyUsed` is false because the callback `onLoad()` creates a fresh response object (Figure 13, lines 10, 53). The join of those states changes the abstract value of `bodyUsed` to $\top$. That change is propagated again to `foo()`.

This imprecision makes the analysis to consider both the `if` and `else` branches at lines 2–5. Thus, the analysis allocates a rejected promise at line 3, as it mistakenly considers that the body has been already consumed. This makes `consumed()` return a value that is either `undefined` or a rejected promise at line 23. The value returned by `consumed()` is finally propagated to `formData()` at line 37, where the analysis reports a false positive; a property access of an `undefined` variable (access of the property "then"), because `text()` might return an undefined variable due to the return statement at line 26. A callback-sensitive analysis neither reports a type error at line 43 nor creates a rejected promise at line 4. It respects the execution order of callbacks, that is, the callback `foo()` is executed *after* the callback `onLoad()`. Therefore, the analysis propagates a more precise state to the entry of `foo()`: the state resulted by the execution of `onLoad()`, where a new response object is initialized with the field `bodyUsed` set to false.

In Listing 2, we initialize a response object with a body that has a string type (line 1). In turn, by calling the `formData()` method, we first read the body of the response in a text format, and then we decode it into a `FormData` object by asynchronously calling the `decode()` function (Figure 13, line 37). Since the body of the response is already in a text format, `text()` returns a fulfilled promise (Figure 13, line 32). At the same time, at line 4 of Listing 2, we allocate a fresh response object whose body is an instance of `Blob`[11]. Therefore, calling `formData()` schedules function `decode()` again. However this time, the callback `decode()` is registered on a different promise because the second call of `text()` returns a promise created by the function `readBlobAsText()` (Figure 13, line 26). A QR-sensitive analysis – which creates a context according to the queue object a callback belongs to – is capable of separating the two invocations of `decode()` because the first call of `decode()` is registered on the promise object that comes from line 32, whereas the second call of `decode()` is added to the promise created by `readBlobAsText()` at line 26.

honoka. We return back to Figure 1. Recall that a callback-insensitive analysis reports a spurious type error at line 17 when we try to access the property `headers` of `honoka.response` because it considers the case where the callback defined at lines 15–23 is executed before that defined at lines 2–14. Thus, `honoka.response` might be uninitialized (recall that `honoka.response` is initialized during the execution of the first callback at line 3). On the other hand, a callback-sensitive analysis consults the callback graph when it is time to propagate the state from the exit point of a callback to the entry point of the next one. In particular, when we analyze the exit node of the first callback, we propagate the current state to the second callback. Therefore, the entry point of the second function has a state that contains a precise value for `honoka.response`, that is, the object coming from the assignment at line 3.

4.4 Threats to Validity

Below we pinpoint the main threats to the validity of our results:

- Our analysis is an extension of TAJS. Therefore, the precision and performance of TAJS play an important role on the results of our work.
- Even though our analysis is designed to be sound, it models some native functions of the JavaScript language unsoundly. For instance, we unsoundly model the native function

[11] `https://developer.mozilla.org/en-US/docs/Web/API/Blob`

`Object.freeze()`, which is used to prevent an object from being updated. Specifically, the model of `Object.freeze()` simply returns the object given as argument.

- We provide manual models for some built-in Node.js modules like `fs`, `http`, etc. or other APIs used in client-side applications such as `XMLHttpRequest`, `Blob`, etc. However, manual modeling might neglect some of the side-effects that stem from the interaction with those APIs, leading to unsoundness [15, 33].
- Our macro-benchmarks consist of JavaScript libraries. Therefore, we need to write some test cases that invoke the API functions of those benchmarks. We provided both hand-written test cases and test cases or examples taken from their documentation, trying to test the main APIs that exercise asynchrony in JavaScript.

5 Related Work

In this section, we briefly present previous work related to formalization and program analysis for (asynchronous) JavaScript.

Semantics. Maffeis et al. [29] presented one of the first formalizations of JavaScript by designing small-step operational semantics for a subset of the 3rd version of ECMAScript. In subsequent work, Guha et al.[16] expressed the semantics of the 3rd edition of ECMAScript through a different approach; they developed a lambda calculus called λ_{JS}, and provided a desugaring mechanism for converting JavaScript code into λ_{JS}. We used λ_{JS} as a base for modeling asynchronous JavaScript. Later, Gardner et al. [13] introduced a program logic for reasoning about client-side JavaScript programs that support ECMAScript 3. They presented big-step operational semantics on the basis of that proposed by Maffeis et. al. [29], and they introduced inference rules for program reasoning which are highly inspired from separation logic [35]. More recently, Madsen et al. [26] and Loring et al. [25] extended λ_{JS} for modeling promises and asynchronous JavaScript respectively. Our model is a variation of their work; our modifications enable us to model different asynchronous features. Some of them are not handled by their models.

Static Analysis for JavaScript. Guarnieri et al. [15] proposed a pointer analysis for a subset of JavaScript. They precluded the use of `eval`-family functions from their analysis as their work focused on widgets where the use of `eval` is not common. It was one of the first attempts that managed to model some of the most peculiar features of JavaScript, such as prototype-based inheritance. TAJS [19, 20, 18] is a typer analyzer for JavaScript which is implemented as a classical dataflow analysis. Our work is implemented as an extension of TAJS. SAFE [23] is a static analysis framework that provides three different formal representations of JavaScript programs: an abstract syntax tree (AST), an intermediate language (IR) and a control-flow graph (CFG). SAFE implements a default analysis phase that is plugged after the construction of CFG. This analysis adopts a similar approach with that of TAJS, i.e., a flow- and context-sensitive analysis that operates on top of CFG. JSAI [21] implements an analysis through the abstract interpretation framework [3]. Specifically, it employs a different approach compared to other existing tools. Unlike TAJS and SAFE, JSAI operates on top of AST rather than CFG; it is flow-sensitive though. To achieve this, the abstract semantics is specified on a CESK abstract machine [10], which provides small-step reduction rules and an explicit data structure (i.e., continuation) which describes the rest of computation, unwinding the flow of the program in this way. The analysis is configurable with different flavors of context-sensitivity which are plugged into the analysis through the widening operator used in the fix-point calculation [17].

Existing static analyses provide sufficient support for precisely modeling browser environment. Jensen et al. [18] modeled HTML DOM by creating a hierarchy of abstract states that reflect the actual HTML object hierarchy. Before the analysis begins, an initial heap is constructed that contains the set of the abstract objects corresponding to the HTML code of the page. Park et al. [33] followed a similar approach for modeling HTML DOM. They also provided a more precise model that respects the actual tree hierarchy of the DOM. For example, their model distinguishes whether one DOM node is nested to another or not.

Program Analysis for Asynchronous JavaScript Programs. The majority of static analyses for JavaScript treat asynchronous programs conservatively [19, 23, 21] – they assume that the event loop processes all the asynchronous callbacks in any order – leading to the analysis imprecision. Also, they focus on the client-side applications, where asynchrony mainly appears in DOM events and AJAX calls.

Madsen et al. [27] proposed one of the first static analysis for server-side event-driven programs. Although their approach is able to handle asynchronous I/O operations – unlike our work – they do not provide support for ES6 promises. Additionally, their work introduced a context-sensitivity strategy that tries to imitate the different iterations of the event loop. However, it imposes a large overhead on the analysis; it is able to handle only small programs (less than 400 lines of code). In our work, we propose callback-sensitivity that improves precision without highly sacrificing performance. More recently, Alimadadi et al. [1] presented a dynamic analysis technique for detecting promise-related errors and anti-patterns in JavaScript programs. Specifically, their approach exploits the promise graph; a representation designed for debugging promise-based programs. Beyond promises, our work also handles a broad spectrum of asynchronous features.

Race detection. Zheng et al. [40] presented one of the first race detectors by employing a static analysis for identifying concurrency issues in asynchronous AJAX calls. The aim of their analysis was to detect data races between the code that pre-processes an AJAX request and the callback invoked when the response of the server is received. A subsequent work [34] adopted a dynamic analysis to detect data races in web applications. They first proposed a happens-before relation model to capture the execution order between different operations that are present in a client-side application, such as the loading of HTML elements, execution of scripts, etc. Using this model, their analyses reports data races, by detecting memory conflicts between unordered functions, However, their approach introduced a lot of false positives because most data races did not lead to severe concurrency bugs. Mutlu et al. [31] combined both dynamic and static analysis and primarily focused on detecting data races that have pernicious consequences on the correctness of applications, such as those that affect the browser storage. Initially, they collected the execution traces of an application, and then, they applied a dataflow analysis on those traces to identify data races. Their approach effectively managed to report a very small number of false positives.

6 Conclusions & Future Work

Building upon previous works, we presented the λ_q calculus for modeling asynchrony in JavaScript. Our calculus λ_q is flexible enough so that we can express almost every asynchronous primitive in the JavaScript language up to the 7th edition of the ECMAScript. We then presented an abstract version of λ_q that over-approximates the semantics of our calculus.

By exploiting the abstract version of $\lambda_\mathtt{q}$, we designed and implemented what is, to the best of our knowledge, the first static analysis for dealing with a wide range of asynchrony-related features. At the same time, we introduced the concept of callback graph; a directed acyclic graph that represents the temporal relations between the execution of asynchronous callbacks, and we proposed a more precise analysis, i.e., callback-sensitive analysis that respects the execution order of callbacks. We parameterized our analysis with a new context-sensitivity option that is specifically used for asynchronous callbacks.

We then experimented with different parameterizations of our analysis on a set of hand-written and real-world programs. The results revealed that we can analyze medium-sized JavaScript programs. The analysis sensitivity (i.e., both callback- and context-sensitivity) was able to ameliorate the analysis precision without highly sacrificing performance. Specifically, as observed in the real-world modules, our analysis achieved a 79% precision for the callback graph, on average. When we combined callback- and QR-sensitivity, we could further improve the callback graph precision by up to 28.5%, and reduce the total number of type errors by 13.9%.

Our work constitutes a general technique that can be used as a base for further research. Specifically, recent studies showed that concurrency bugs found in JavaScript programs may sometimes be caused by asynchrony [39, 5]. We could leverage our work to design a client analysis on top of it so that it statically detects data races in JavaScript programs. Our callback graph might be an essential element for such an analysis because we could inspect it to identify callbacks whose execution might be non-deterministic, i.e., unconnected nodes in the callback graph.

References

1 Saba Alimadadi, Di Zhong, Magnus Madsen, and Frank Tip. Finding Broken Promises in Asynchronous JavaScript Programs. *Proc. ACM Program. Lang.*, 2(OOPSLA):162:1–162:26, 2018. doi:10.1145/3276532.

2 Gogul Balakrishnan and Thomas Reps. Recency-Abstraction for Heap-allocated Storage. In *Proceedings of the 13th International Conference on Static Analysis*, SAS'06, pages 221–239, 2006. doi:10.1007/11823230_15.

3 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, 1977. doi:10.1145/512950.512973.

4 Frank Dabek, Nickolai Zeldovich, Frans Kaashoek, David Mazières, and Robert Morris. Event-driven programming for robust software. In *Proceedings of the 10th workshop on ACM SIGOPS European workshop*, pages 186–189. ACM, 2002.

5 James Davis, Arun Thekumparampil, and Dongyoon Lee. Node.Fz: Fuzzing the server-side event-driven architecture. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 145–160, 2017. doi:10.1145/3064176.3064188.

6 Asger Feldthaus, Todd Millstein, Anders Møller, Max Schäfer, and Frank Tip. Tool-supported Refactoring for JavaScript. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 119–138, 2011. doi:10.1145/2048066.2048078.

7 Asger Feldthaus and Anders Møller. Semi-automatic Rename Refactoring for JavaScript. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages &#38; Applications*, OOPSLA '13, pages 323–338, 2013. doi:10.1145/2509136.2509520.

8 Asger Feldthaus, Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. Efficient Construction of Approximate Call Graphs for JavaScript IDE Services. In *Proceedings of*

the 2013 International Conference on Software Engineering, ICSE '13, pages 752–761, 2013. doi:10.1109/ICSE.2013.6606621.

9 Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics engineering with PLT Redex.* Mit Press, 2009.

10 Mattias Felleisen and D. P. Friedman. A Calculus for Assignments in Higher-order Languages. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '87, pages 314–, 1987. doi:10.1145/41625.41654.

11 K. Gallaba, Q. Hanam, A. Mesbah, and I. Beschastnikh. Refactoring Asynchrony in JavaScript. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 353–363, 2017. doi:10.1109/ICSME.2017.83.

12 K. Gallaba, A. Mesbah, and I. Beschastnikh. Don't Call Us, We'll Call You: Characterizing Callbacks in Javascript. In *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–10, 2015. doi:10.1109/ESEM.2015.7321196.

13 Philippa Anne Gardner, Sergio Maffeis, and Gareth David Smith. Towards a Program Logic for JavaScript. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 31–44, 2012. doi:10.1145/2103656.2103663.

14 Github. GitHub Octoverse 2017 | Highlights from the last twelve months. https://octoverse.github.com/, 2017. [Online; accessed 08-January-2019].

15 Salvatore Guarnieri and Benjamin Livshits. GATEKEEPER: Mostly static enforcement of security and reliability policies for Javascript code. In *Proceedings of the 18th Conference on USENIX Security Symposium*, SSYM'09, pages 151–168, 2009.

16 Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. The Essence of Javascript. In *Proceedings of the 24th European Conference on Object-oriented Programming*, ECOOP'10, pages 126–150, 2010.

17 Ben Hardekopf, Ben Wiedermann, Berkeley Churchill, and Vineeth Kashyap. Widening for Control-Flow. In Kenneth L. McMillan and Xavier Rival, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 472–491. Springer Berlin Heidelberg, 2014. doi:10.1007/978-3-642-54013-4_26.

18 Simon Holm Jensen, Magnus Madsen, and Anders Møller. Modeling the HTML DOM and Browser API in Static Analysis of JavaScript Web Applications. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 59–69, 2011. doi:10.1145/2025113.2025125.

19 Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type Analysis for JavaScript. In *Proceedings of the 16th International Symposium on Static Analysis*, SAS '09, pages 238–255, 2009. doi:10.1007/978-3-642-03237-0_17.

20 Simon Holm Jensen, Anders Møller, and Peter Thiemann. Interprocedural Analysis with Lazy Propagation. In Radhia Cousot and Matthieu Martel, editors, *Static Analysis*, pages 320–339. Springer Berlin Heidelberg, 2010.

21 Vineeth Kashyap, Kyle Dewey, Ethan A. Kuefner, John Wagner, Kevin Gibbons, John Sarracino, Ben Wiedermann, and Ben Hardekopf. JSAI: A static analysis platform for JavaScript. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 121–132, 2014. doi:10.1145/2635868.2635904.

22 Y. Ko, H. Lee, J. Dolby, and S. Ryu. Practically Tunable Static Analysis Framework for Large-Scale JavaScript Applications (T). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 541–551, 2015. doi:10.1109/ASE.2015.28.

23 Hongki Lee, Sooncheol Won, Joonho Jin, Junhee Cho, and Sukyoung Ryu. SAFE: Formal specification and implementation of a scalable analysis framework for ECMAscript. In *FOOL 2012: 19th International Workshop on Foundations of Object-Oriented Languages*, page 96, 2012.

24 Ondřej Lhoták and Laurie Hendren. Evaluating the Benefits of Context-sensitive Points-to Analysis Using a BDD-based Implementation. *ACM Trans. Softw. Eng. Methodol.*, 18(1):3:1–3:53, 2008. doi:10.1145/1391984.1391987.

25 Matthew C. Loring, Mark Marron, and Daan Leijen. Semantics of Asynchronous JavaScript. In *Proceedings of the 13th ACM SIGPLAN International Symposium on on Dynamic Languages*, DLS 2017, pages 51–62, 2017. doi:10.1145/3133841.3133846.

26 Magnus Madsen, Ondřej Lhoták, and Frank Tip. A Model for Reasoning About JavaScript Promises. *Proc. ACM Program. Lang.*, 1(OOPSLA):86:1–86:24, 2017. doi:10.1145/3133910.

27 Magnus Madsen, Frank Tip, and Ondřej Lhoták. Static Analysis of Event-driven Node.Js JavaScript Applications. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 505–519, 2015. doi:10.1145/2814270.2814272.

28 S. Maffeis and A. Taly. Language-Based Isolation of Untrusted JavaScript. In *2009 22nd IEEE Computer Security Foundations Symposium*, pages 77–91, 2009. doi:10.1109/CSF.2009.11.

29 Sergio Maffeis, John C. Mitchell, and Ankur Taly. An Operational Semantics for JavaScript. In *Proceedings of the 6th Asian Symposium on Programming Languages and Systems*, APLAS '08, pages 307–325, 2008. doi:10.1007/978-3-540-89330-1_22.

30 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.

31 Erdal Mutlu, Serdar Tasiran, and Benjamin Livshits. Detecting JavaScript Races That Matter. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 381–392, 2015. doi:10.1145/2786805.2786820.

32 Node.js. The Node.js Event Loop, Timers, and process.nextTick(). https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick/, 2018. [Online; accessed 04-June-2018].

33 C. Park, S. Won, J. Jin, and S. Ryu. Static Analysis of JavaScript Web Applications in the Wild via Practical DOM Modeling (T). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 552–562, 2015. doi:10.1109/ASE.2015.27.

34 Boris Petrov, Martin Vechev, Manu Sridharan, and Julian Dolby. Race Detection for Web Applications. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 251–262, 2012. doi:10.1145/2254064.2254095.

35 J. C. Reynolds. Separation logic: a logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74, 2002. doi:10.1109/LICS.2002.1029817.

36 Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. An Analysis of the Dynamic Behavior of JavaScript Programs. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 1–12, 2010. doi:10.1145/1806596.1806598.

37 Cristian-Alexandru Staicu, Michael Pradel, and Benjamin Livshits. SYNODE: understanding and automatically preventing injection attacks on Node.Js. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*, 2018.

38 Kwangwon Sun and Sukyoung Ryu. Analysis of JavaScript Programs: Challenges and Research Trends. *ACM Comput. Surv.*, 50(4):59:1–59:34, 2017. doi:10.1145/3106741.

39 J. Wang, W. Dou, Y. Gao, C. Gao, F. Qin, K. Yin, and J. Wei. A comprehensive study on real world concurrency bugs in Node.js. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 520–531, 2017. doi:10.1109/ASE.2017.8115663.

40 Yunhui Zheng, Tao Bao, and Xiangyu Zhang. Statically Locating Web Application Bugs Caused by Asynchronous Calls. In *Proceedings of the 20th International Conference on World Wide Web*, WWW '11, pages 805–814, 2011. doi:10.1145/1963405.1963517.

A Program Logic for First-Order Encapsulated WebAssembly

Conrad Watt
University of Cambridge, UK
conrad.watt@cl.cam.ac.uk

Petar Maksimović
Imperial College London, UK
Mathematical Institute SASA, Serbia
p.maksimovic@imperial.ac.uk

Neelakantan R. Krishnaswami
University of Cambridge, UK
nk480@cl.cam.ac.uk

Philippa Gardner
Imperial College London, UK
p.gardner@imperial.ac.uk

Abstract

We introduce Wasm Logic, a sound program logic for first-order, encapsulated WebAssembly. We design a novel assertion syntax, tailored to WebAssembly's stack-based semantics and the strong guarantees given by WebAssembly's type system, and show how to adapt the standard separation logic triple and proof rules in a principled way to capture WebAssembly's uncommon structured control flow. Using Wasm Logic, we specify and verify a simple WebAssembly B-tree library, giving abstract specifications independent of the underlying implementation. We mechanise Wasm Logic and its soundness proof in full in Isabelle/HOL. As part of the soundness proof, we formalise and fully mechanise a novel, big-step semantics of WebAssembly, which we prove equivalent, up to transitive closure, to the original WebAssembly small-step semantics. Wasm Logic is the first program logic for WebAssembly, and represents a first step towards the creation of static analysis tools for WebAssembly.

2012 ACM Subject Classification Theory of computation → Separation logic

Keywords and phrases WebAssembly, program logic, separation logic, soundness, mechanisation

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2019.9

Acknowledgements We would like to thank the reviewers, whose comments were valuable in improving the paper. All authors were supported by the EPSRC Programme Grant "REMS: Rigorous Engineering for Mainstream Systems" (EP/K008528/1). In addition: Watt was supported by an EPSRC DTP award (EP/N509620/1); Maksimović was supported by the Serbian Ministry of Education and Science through the Mathematical Institute SASA, projects ON174026 and III44006; Krishnaswami was supported by the EPSRC Programme Grant "Semantic Foundations for Interactive Programs" (EP/N02706X/2); and Gardner was supported by the EPSRC Fellowship "VeTSpec: Verified Trustworthy Software Specification" (EP/R034567/1).

1 Introduction

WebAssembly [16] is a stack-based, statically typed bytecode language. It is the first new language to be natively supported on the Web in nearly 25 years, following JavaScript (JS). It was created to act as the safe, fast, portable low-level code of the Web, in answer to the growing sophisticated, computationally intensive demands of the Internet of today, such as 3D visualisation, audio/video processing, and games. For years, developers wishing to

33rd European Conference on Object-Oriented Programming (ECOOP 2019).
Editor: Alastair F. Donaldson; Article No. 9; pp. 9:1–9:30
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

execute calculation-heavy programs written in C/C++ on the Web had to compile them to asm.js [17], a subset of JS. In time, such code has become widespread [53, 25, 11], but the fundamental limitations of JS as a compilation target have become too detrimental to ignore. WebAssembly is designed from the ground up to be an efficient, Web-compatible compilation target, obsoleting asm.js and other similar endeavours, such as Native Client [52]. All major browser vendors, including Google, Microsoft, Apple, and Mozilla, have pledged to support WebAssembly, and the past two years have seen a flurry of implementation activity [49].

These facts alone would be enough to motivate that WebAssembly will be an important technology, and a worthy target for formal methods. The designers of WebAssembly have anticipated this, and have specified WebAssembly using a precise formal small-step semantics, combined with a sound type system. Moreover, WebAssembly's semantics, type system, and soundness have already been fully mechanised [46], and the WebAssembly Working Group requires any further additions to WebAssembly to be formally specified.

The main use case for WebAssembly is to inter-operate with JS in creating content for the Web. More precisely, WebAssembly functions can be grouped into *modules*, which provide interfaces through which users can call WebAssembly code, and self-contained (encapsulated) modules can be used as drop-in replacements for their existing JS counterparts and already constitute a major design pattern in WebAssembly. We believe that having a formalism for describing and reasoning about WebAssembly modules and their interfaces is essential, in line with WebAssembly's emphasis on formal methods. Thus far, very little work has been done on static analysis for WebAssembly (cf. §6).

We present Wasm Logic, a sound program logic for reasoning about first-order, encapsulated WebAssembly modules, such as data structure libraries. Enabled by the strong guarantees of WebAssembly's type system, we design a novel assertion syntax, tailored to WebAssembly's stack-based semantics. We further adapt the standard separation logic triple and proof rules in a principled way to capture WebAssembly's uncommon structured control flow.

Having a program logic for WebAssembly is valuable for several reasons. First, as WebAssembly programs are distributed without their originating source code, any client-side verification would have to rely on a WebAssembly-level logic. Similarly, verification techniques such as proof-transforming compilation [31, 1, 26, 37] rely on the existence of a program logic for the target language. Finally, some fundamental data structure libraries are expected to be implemented directly in WebAssembly for efficiency reasons. For example, the structure of B-trees strongly aligns with the way in which WebAssembly memory is managed (cf. §4.2).

To demonstrate the usability of Wasm Logic, we implement, specify, and verify a simple WebAssembly B-tree library. In doing so, we discuss how the new and adapted Wasm Logic proof rules can be used in practice. The specifications that we obtain are abstract, in that they do not reveal any details about the underlying implementation.

We mechanise Wasm Logic and its soundness proof in full in Isabelle/HOL, building on a previous WebAssembly mechanisation of Watt [46]. We prove Wasm Logic sound against a novel, big-step semantics of WebAssembly, and also mechanise a proof of equivalence between the transitive closure of the original small-step semantics and our big-step semantics. Our mechanisation totals ~10,400 lines of non-comment, non-whitespace Isabelle code, not counting code inherited from the existing mechanisation.

2 A Brief Overview of WebAssembly

We give the syntax and an informal description of the semantics of WebAssembly. A precise account of its semantics is given through our program logic in §3 and also through our big-step semantics, introduced in §5 and presented in full in [47].

2.1 WebAssembly Syntax

WebAssembly has a human-readable *text format* based on s-expressions, which we use throughout. The abstract syntax of WebAssembly programs [16], is given in full in Figure 1. As we consider first-order, encapsulated modules, we grey out the remaining, non-relevant syntax. We describe the semantics of the instructions informally in §2.3, and additional syntax as it arises in the paper. A full description of WebAssembly can be found in [16].

$$
\begin{array}{lrcl}
\text{(constants)} & k & ::= & \ldots \\
\text{(immediates)} & im & ::= & i, a, o \in nat \\
\text{(packed types)} & pt & ::= & \mathsf{i8} \mid \mathsf{i16} \mid \mathsf{i32} \\
\text{(value types)} & t & ::= & \mathsf{i32} \mid \mathsf{i64} \mid \mathsf{f32} \mid \mathsf{f64} \\
\text{(function types)} & ft & ::= & t^* \to t^* \\
\text{(global types)} & gt & ::= & \mathsf{mut}^? \; t
\end{array}
$$

$$
\begin{array}{rcl}
unop_{\mathsf{i}N} & ::= & \mathsf{clz} \mid \mathsf{ctz} \mid \mathsf{popcnt} \\
unop_{\mathsf{f}N} & ::= & \mathsf{neg} \mid \mathsf{abs} \mid \mathsf{ceil} \mid \mathsf{floor} \mid \mathsf{trunc} \mid \mathsf{nearest} \mid \mathsf{sqrt} \\
binop_{\mathsf{i}N} & ::= & \mathsf{add} \mid \mathsf{sub} \mid \mathsf{mul} \mid \mathsf{div\_}sx \mid \mathsf{rem\_}sx \mid \mathsf{and} \mid \\
 & & \mathsf{or} \mid \mathsf{xor} \mid \mathsf{shl} \mid \mathsf{shr\_}sx \mid \mathsf{rotl} \mid \mathsf{rotr} \\
binop_{\mathsf{f}N} & ::= & \mathsf{add} \mid \mathsf{sub} \mid \mathsf{mul} \mid \mathsf{div} \mid \\
 & & \mathsf{min} \mid \mathsf{max} \mid \mathsf{copysign} \\
testop_{\mathsf{i}N} & ::= & \mathsf{eqz} \\
relop_{\mathsf{i}N} & ::= & \mathsf{eq} \mid \mathsf{ne} \mid \mathsf{lt\_}sx \mid \mathsf{gt\_}sx \mid \\
 & & \mathsf{le\_}sx \mid \mathsf{ge\_}sx \\
relop_{\mathsf{f}N} & ::= & \mathsf{eq} \mid \mathsf{ne} \mid \mathsf{lt} \mid \mathsf{gt} \mid \mathsf{le} \mid \mathsf{ge} \\
cvtop & ::= & \mathsf{convert} \mid \mathsf{reinterpret} \\
sx & ::= & \mathsf{s} \mid \mathsf{u}
\end{array}
$$

$$
\begin{array}{lrcl}
\text{(instructions)} & e & ::= & t.\mathsf{const}\; k \mid \mathsf{drop} \mid \mathsf{nop} \mid \mathsf{select} \mid \mathsf{unreachable} \mid \\
 & & & t.unop_t \mid t.binop_t \mid t.testop_t \mid t.relop_t \mid \\
 & & & t.cvtop_{t\_sx^?} \mid \mathsf{get\_local}\; im \mid \mathsf{set\_local}\; im \mid \\
 & & & \mathsf{tee\_local}\; im \mid \mathsf{get\_global}\; im \mid \\
 & & & \mathsf{set\_global}\; im \mid t.\mathsf{store}\; pt^? \; a \; o \mid \\
 & & & t.\mathsf{load}\; (pt\_sx)^? \; a \; o \mid \mathsf{mem.size} \mid \mathsf{mem.grow} \mid \\
 & & & \mathsf{block}\; ft\; e^*\; \mathsf{end} \mid \mathsf{loop}\; ft\; e^*\; \mathsf{end} \mid \\
 & & & \mathsf{if}\; ft\; e^*\; \mathsf{else}\; e^*\; \mathsf{end} \mid \\
 & & & \mathsf{br}\; im \mid \mathsf{br\_if}\; im \mid \mathsf{br\_table}\; im^+ \mid \\
 & & & \mathsf{return} \mid \mathsf{call}\; im \mid \mathsf{call\_indirect}\; ft
\end{array}
$$

$$
\begin{array}{lrcl}
\text{(functions)} & func & ::= & ex^*\; \mathsf{func}\; ft\; \mathsf{local}\; t^*\; e^* \mid ex^*\; \mathsf{func}\; ft\; imp \\
\text{(globals)} & glob & ::= & ex^*\; \mathsf{global}\; gt\; e^* \mid ex^*\; \mathsf{global}\; gt\; imp \\
\text{(tables)} & tab & ::= & ex^*\; \mathsf{table}\; n\; im^* \mid ex^*\; \mathsf{table}\; n\; imp \\
\text{(memories)} & mem & ::= & ex^*\; \mathsf{memory}\; n \mid ex^*\; \mathsf{memory}\; n\; imp \\
\text{(imports)} & imp & ::= & \mathsf{import}\; \text{“}name\text{”}\; \text{“}name\text{”} \\
\text{(exports)} & ex & ::= & \mathsf{export}\; \text{“}name\text{”} \\
\text{(modules)} & mod & ::= & \mathsf{module}\; func^*\; glob^*\; tab^?\; mem^?
\end{array}
$$

Note: we denote lists with a $*$ superscript: for example, t^* denotes a list of types.

Figure 1 WebAssembly Abstract Syntax of [16], with aspects not relevant to this work greyed out.

2.2 The WebAssembly Memory Model

Values. WebAssembly values, v, may have one of four *value types*, representing 32- and 64-bit IEEE-754 integers and floating-point numbers: i32, i64, f32, or f64. We denote values using their type: for example, a 32-bit representation of the integer 42 is denoted 42_{i32}. If the type of a value is not given, it is assumed to be i32 by default.

Local and Global Variables. WebAssembly programs have access to statically declared variables, which may be *local* or *global.* Local variables are declared per-function. They live in local variable stores, which exist only in the body of their declaring function. They include function arguments, followed by a number of “scratch” local variables initialised to zero when the function is called. Global variables are declared by the enclosing module. They live in a global variable store, are initialised to zero at the beginning of the execution, and are accessible by all of the functions of the module.

In contrast to most standard programming languages, WebAssembly variables cannot be referenced by name. Instead, both the global and local variable stores are designed as mappings from natural numbers to WebAssembly values, and variables are referenced by their index in the corresponding variable store, as shown in §2.3.

Stack. WebAssembly computation is based on a stack machine: all instructions pop their arguments from and push their results onto a stack of WebAssembly values. By convention, stack concatenation is implicit and the top of the stack is written on the right-hand side: for example, a stack with a 32-bit 0 at its top followed by m WebAssembly values would be denoted as v^m 0. Note that the type system of WebAssembly allows us to statically know both the number of elements on the stack and their types at every point of program execution.

Memory. WebAssembly has a linear memory model. A WebAssembly memory is an array of bytes, indexed by i32 values, which are interpreted as offsets. Memory is allocated in units of *pages*, and each page is exactly 64k bytes in size.

2.3 WebAssembly Instructions

WebAssembly has a wide array of instructions, which we divide into: basic instructions, variable management instructions, memory management instructions, function-related instructions, and control flow instructions, all of which we discuss below. Every instruction consumes its arguments from the stack, carries out its operation, and pushes any resulting value back onto the stack. Moreover, every instruction is typed, with its type describing the types of its arguments and result. We illustrate how this works in Figure 2, which describes WebAssembly addition of two 32-bit integers starting from an empty stack. In particular, the **i32.const** command, whose type is $[] \to [\mathsf{i32}]$, does not require any arguments and puts the given value on the stack, whereas the **i32.add** instruction, whose type is $[\mathsf{i32}, \mathsf{i32}] \to [\mathsf{i32}]$, takes two arguments from the stack and returns their sum.

WebAssembly gives two official, equivalent, semantics: a semi-formal prose semantics and an entirely formal small-step semantics [50]. In this paper, we introduce an additional, equivalent, big-step semantics as part of the soundness proof of our logic. Most of our diagrams and explanatory text throughout the paper follow the style of the prose semantics, as its treatment of the value stack is most useful in explaining the behaviour of the logic. We denote prose-style execution steps using $\rightsquigarrow$, and introduce the other semantics as necessary.

Basic Instructions. WebAssembly values can be declared using the t.**const** command, typed $[] \to [t]$, in the style of (**i32.const** 2) of Figure 2. The (**drop**) command, typed $[t] \to []$, pops and discards the top stack item, while (**nop**), typed $[] \to []$ has no effect. The (**select**) instruction, typed $[t, t, \mathsf{i32}] \to []$, takes three values from the stack, v_1, v_2, and c. If c is non-zero, v_1 is pushed back onto the stack, and v_2 otherwise. The (**unreachable**) instruction, typed $[] \to t*$, causes the program to halt with a runtime error, which is represented in WebAssembly by a special **Trap** execution result (cf. §2.4).

WebAssembly also provides a variety of (type-annotated) unary and binary arithmetic operations (Figure 1, *unop* and *binop*, respectively), unary and binary logical operations (Figure 1, *testop* and *relop*, respectively), and casting operations (Figure 1, *cvtop*). Some of these operations can cause a **Trap**: for example, if we attempt division by zero or try to convert a floating-point number to an integer when the result is not representable. Their meaning is detailed in [16], and we address them in this paper by need.

Variable Management Instructions. Local and global variables can be read from and written to using the appropriate **get** and **set** instructions, and all variable accesses are performed using static indexes. For example, (**get_local** i), typed $[] \to [t]$ (where t is the statically known type of the i-th local variable), will push the value of the i-th declared local variable of the current function onto the stack, and (**set_global** i), typed $[t] \to []$, will set

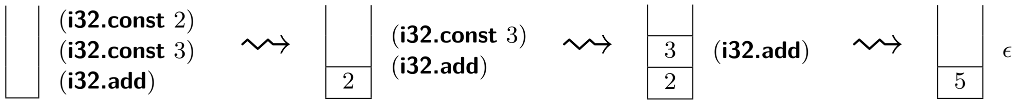


Figure 2 Addition in WebAssembly.

the value of the i-th declared global variable to the value at the top of the stack, which is consumed in the process. It is also possible to set a local variable without consuming this value from the stack by using the **tee_local** instruction, typed $[t] \to [t]$.

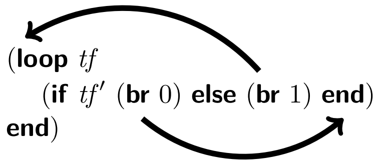
Memory Management Instructions. Stack values may be serialised and copied into the appropriate number of bytes in memory through the type-annotated **store** instruction. The (t.**store**) instruction, typed $[\mathsf{i32}, t] \to []$, interprets its i32 argument as an index into the memory, while the second is serialised into the appropriate number of bytes to be stored sequentially, starting from the indexed memory location.

Conversely, the type-annotated **load** instruction reads bytes from the memory and produces the appropriate stack value. (t.**load**), typed $[\mathsf{i32}] \to [t]$, will consume a single i32 value (the address), and then read the appropriate number of bytes starting from that address, leaving the corresponding value of type t on the top of the stack. WebAssembly specifies that every value can be serialised, and every byte sequence of the appropriate length can be interpreted as a value; there are no trap representations for values.

The size of the memory can be inspected by executing the (**mem.size**) instruction, typed $[] \to [\mathsf{i32}]$, which returns an 32-bit integer denoting the current memory size in pages. The WebAssembly memory may also be grown by executing the (**mem.grow**) instruction, typed $[\mathsf{i32}] \to [\mathsf{i32}]$, which takes a single i32 value from the top of the stack and attempts to grow the memory by that many pages, returning the previous size of the memory, in pages, as a 32-bit integer if successful. (**mem.grow**) is always allowed to fail non-deterministically, to represent some memory limitation of the host environment. In this case, the memory is not altered, and the value -1_{i32} is returned.

Control Flow Instructions. Most WebAssembly features have many similarities to other bytecodes, such as that of the Java Virtual Machine [24]. WebAssembly's approach to control flow, however, is uncommon. WebAssembly does not allow unstructured control flow in the style of a **goto** instruction. Instead, it has three control constructs that implement structured control flow: (**block** ft e^* **end**), (**loop** ft e^* **end**), and (**if** ft e^* **else** e^* **end**). Each of these control constructs is annotated with a function type ft of the form $t^m \to t^n$, meaning that its body, e^*, requires m elements from the stack and places back n elements onto the stack on exit. The semantics guarantees that this type precisely describes the effect the construct will have on the stack after it/its body terminates. For example, a (**loop** $(t^m \to t^n)$ e^* **end**), no matter the behaviour of its body, will always leave precisely n additional values on the stack upon termination. Control constructs may be nested within each other in the intuitive way. The execution of a control construct consists of executing its body to termination.

Within the body of a control construct, a break instruction, (**br** i), may be executed. As control constructs can be nested, **br** is parameterised by a static index i, indicating the control construct that it targets (indexing inner to outer). The behaviour of **br** depends on the type of its target. When targeting a **block** or an **if**, **br** acts as a “break” statement of a high-level language, which transfers control to the matching **end** opcode, jumping out of all intervening constructs. When targeting a **loop**, the break instruction acts like a “continue”



```
(loop tf
   (if tf′ (br 0) else (br 1) end)
end)
```

Figure 3 Example of WebAssembly control flow. Executing (**br** 0) jumps to the end of the **if**, while (**br** 1) jumps to the *start* of the **loop**.

statement, transferring control back to the beginning of the loop. If the body of a **loop** terminates without executing a **br**, the loop terminates with the result of the body. The **br** instruction is, therefore, required for loop iteration. We illustrate this in Figure 3. The first break, (**br** 0), targets its enclosing **if** instruction, meaning that control should be transferred to the end of that **if** instruction. The second break, (**br** 1), targets the outer **loop** instruction, meaning that control should be transferred to the beginning of that loop.

WebAssembly also has two instructions for conditional breaking: **br_if** and **br_table**. The (**br_if** i) instruction takes one i32 value off the stack and, if this value is not equal to zero, behaves as (**br** i), and as (**nop**) otherwise. On the other hand, the (**br_table** $i_0 \dots i_n$ i) instruction acts like a switch statement. It takes one i32 value v off the stack and then: if $0 \leqslant v \leqslant n$, it behaves as (**br** i_v); otherwise, it behaves as (**br** i).

Function-related Instructions. WebAssembly supports two types of functions. First, the host environment wil supply *import* functions for use by the WebAssembly module. These functions may be JavaScript *host* functions or may come from other WebAssembly modules. Second, the module itself will define its own native WebAssembly functions.

Functions are called using the (**call** i) instruction, which executes the i-th function, indexing imports first, followed by module-native functions in order of declaration. As WebAssembly functions are declared with a precise type annotation, (**call** i) also takes the type of the i-th function. WebAssembly also provides a mechanism for dynamic dispatch through the **call_indirect** instruction.

Our core logic does not support imported functions, as well as the **call_indirect** dynamic dispatch, as all of these features require JavaScript intervention for non-trivial use. Without **call_indirect**, WebAssembly provides no mechanism for higher-order code – this is why we characterise our logic as supporting “first-order, encapsulated WebAssembly”. We view these features as part of further work on JavaScript/WebAssembly interoperability and discuss the ramifications of providing support for them in §7.

Finally, the (**return**) instruction is analogous to **br**, except that it breaks out of *all* enclosing constructs, concluding the execution of the function.

Modules. A WebAssembly program is represented as a module, which consists of: a list of functions; a list of global variables; the (optional) **call_indirect** table; and the (optional) linear memory. Formally, this is written as **module** $func^*$ $glob^*$ $tab^?$ $mem^?$. Functions are made of a function type ft, a series of typed local variable declarations t^*, and a function body e^*. Globals are made up of a type declaration gt (including an optional immutable flag for declaring constants) and an initializer expression e^*. Tables collect a list of function indexes for use by the **call_indirect** instruction. Memories declare their initial size measured in pages. Functions, globals, tables, and memories may be shared between modules through a system of imports and exports, but we do not support this in our current logic, in large part because WebAssembly modules cannot satisfy each other's imports natively, but must currently rely on JavaScript "glue code" to compose together.

2.4 WebAssembly Semantics

WebAssembly's official specification [16] provides a formal small-step semantics, mechanised in Isabelle/HOL by Watt [46]. As part of the soundness proof of our program logic, we define and mechanise in Isabelle/HOL a WebAssembly big-step semantics that we formally prove equivalent, up to transitive closure, to the mechanised small-step semantics of [46]. We introduce a fine-grained semantics of the **br** and **return** instructions, which is independent of the style of semantics chosen and streamlines formal reasoning.

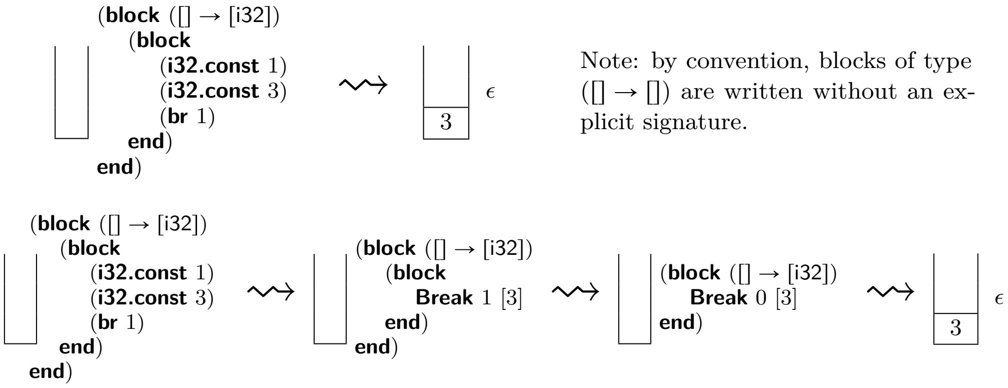
Execution Results. WebAssembly executions terminate with one of the following results:

- **Normal** $v*$, representing standard termination with a list of values $v*$ (in future, we often elide the **Normal** constructor and consider it to be the default result type);
- **Trap**, representing a runtime error (cf. §2.3 for examples of instructions that can trap);
- **Break** n $v*$, describing an in-progress **br** instruction;
- **Return** $v*$, describing an in-progress **return** instruction.

Whereas the first two types of results are introduced by Haas et al. in [16], the last two are introduced by us in this paper. The reason for this is that the WebAssembly formal semantics of [16] gives a very coarse-grained semantics to the **br** and **return** instructions. A **br** instruction targetting a control construct is defined as breaking to it immediately in a single step, discarding everything in between, including all other nested control constructs.

This complicates inductive proofs over the semantics, impairing formal reasoning [46]. In fact, this semantics is too coarse-grained for our proof system and we need to introduce the notions of "in-progress" **br** and **return** instructions as explicit execution results.

We illustrate the difference between the approach of Haas et al. [16] and our approach in Figure 4. The top reduction follows the official semantics of [16]. There, (**br** 1) breaks out of two blocks in a single step, transferring exactly one value out of the **block**, in order to satisfy the targeted block's type signature. We make this semantics more granular by introducing an auxiliary **Break** result type. Concretely, **Break** n $v*$ denotes an in-progress **br** instruction, with n remaining contexts to break out of, in the process of transferring $v*$ values to the



```
(block ([] → [i32])
  (block
    (i32.const 1)
    (i32.const 3)
    (br 1)
  end)
end)
```

Note: by convention, blocks of type ([] → []) are written without an explicit signature.

```
(block ([] → [i32])
  (block
    (i32.const 1)
    (i32.const 3)
    (br 1)
  end)
end)
```

```
(block ([] → [i32])
  (block
    Break 1 [3]
  end)
end)
```

```
(block ([] → [i32])
  Break 0 [3]
end)
```

Figure 4 Granularity of **br** executions: Haas et al. [16] (top); our approach (bottom).

target context, as shown in the bottom reduction of Figure 4. Similarly, **Return** v^* represents an in-progress **return** instruction, with the only difference being that **Return** does not require a remaining context count, as it breaks out of all enclosing constructs.

Big-Step Semantic Judgement. The judgement of our big-step semantics is of the form

$$(s, loc^*, v_e^*\, e^*) \Downarrow_{inst}^{labs,ret} (s', loc'^*, res).$$

On the left-hand side of the judgement, we have *configurations* of the form $(s, loc^*, v_e^*\, e^*)$, where s is a *store* containing whole-program runtime information (e.g. global variables and the memory), loc^* is the list of current local variables, and v_e^* is a value stack v^* lifted to **const** instructions, which is then directly concatenated with e^*, the list of instructions to execute.[1] Configuration execution yields an updated store s', updated local variables loc'^*, and a result res, which has one of the four above-mentioned result types.

Additionally, execution is defined with respect to a subscript *inst*. This is the *run-time instance*, a record which keeps track of which elements of s have been allocated by the current program. In the case of the encapsulated modules that we consider, its role in the formalism is trivial, but its full role is described in the official specification [16], and we give a full definition in [47], along with our big-step semantics.

Finally, execution is also defined with respect to a list of break label arities *labs* (a *nat list*), and a return label arity *ret* (a single *nat*). As depicted in Fig. 4, **Break** and **Return** results must transfer precisely the correct number of values to satisfy the type of the context it is targeting. The *labs* and *ret* parameters keep track of the number of values required, so that, for example, if *res* is of the form **Break** $k\ v^n$, then $labs!k = n$. Similarly, if *res* is of the form **Return** v^n, then $ret = n$.

Equivalence Result. We recall the original formal small-step semantic judgement of [16], which is of the form $(s, loc^*, v_e^*\, e^*) \hookrightarrow_{inst} (s', loc'^*, v_e'^*\, e'^*)$. This judgement does not include our break or return labels.

We state our equivalence result in Theorem 1 and mechanise its proof in Isabelle/HOL. We denote the transitive closure of the small-step semantics by $\hookrightarrow^*$. Both $\hookrightarrow$ and $\Downarrow$ are subscripted by the instance *inst*, the big-step derivation starts with empty *labs* ([]) and empty *ret* (ϵ) components, and v'^* denotes the list of values obtained from $v_e'^*$ by removing their leading **const**s.

▶ **Theorem 1** (reduce_trans_equiv_reduce_to)**.**

$$(s, loc^*, v_e^*\, e^*) \hookrightarrow_{inst}^* (s', loc'^*, v_e'^*) \Longleftrightarrow (s, loc^*, v_e^*\, e^*) \Downarrow_{inst}^{[],\epsilon} (s', loc'^*, \textbf{Normal}\ v'^*)\ \wedge$$
$$(s, loc^*, v_e^*\, e^*) \hookrightarrow_{inst}^* (s', loc'^*, [\textbf{trap}]) \Longleftrightarrow (s, loc^*, v_e^*\, e^*) \Downarrow_{inst}^{[],\epsilon} (s', loc'^*, \textbf{Trap})$$

Theorem 1 relates terminal states (values e^* or a trap result [**trap**]) in the small step semantics with execution results in the big-step semantics. In particular, it shows that the small- and big-step semantics give equivalent results for all terminating programs. The proof also requires auxiliary lemmas about how the big-step **Break** and **Return** execution results correspond to behaviours in the small-step semantics. These lemmas are not included here for space, but can be found in the mechanisation.

[1] This treatment of the value stack is a key difference between the official prose and formal semantics. In the prose semantics, the stack is represented as a list of values v^*, together with an executing list of instructions e^*, which modifies the stack. In the formal semantics, the value stack is represented as a list of **const** instructions, and directly concatenated with the executing list of instructions to form a single list. Reduction rules are defined between configurations, pattern-matching between **const** instructions and other instructions, such as **add**, without ever explicitly manipulating a separate value stack.

3 Wasm Logic

We present Wasm Logic, a program logic for first-order, encapsulated WebAssembly modules. We define a novel assertion syntax, with a highly structured stack assertion which takes advantage of WebAssembly's strict type system. Our proof rules for the WebAssembly **br** and **return** instructions are inspired by a foundational proof rule for "structured **goto**" by Clint and Hoare [7], and extend their work to the world of separation logic [35]. We fully mechanise and prove soundness of Wasm Logic in Isabelle/HOL, as detailed in §5.

3.1 Assertion Language

Wasm Logic assertions encode information about WebAssembly runtime states. Their semantic interpretation is formally described in §5, in the context of our soundness result.

In many programming languages, program state is made up of the values stored in *variables* and the values stored in the *heap*. In this case, it is natural for assertions to be expressed using a *separation logic*, which extends predicate logic with connectives for reasoning about resource separation, and is useful for modular client reasoning [35].

WebAssembly, however, also allows values to be stored in the stack. Given how the WebAssembly's type system provides static knowledge of the stack size and of the types of each of its elements at every program point, we believe that reasoning about the WebAssembly stack *should* be simple: that is, it should not result in proofs more complicated than those of traditional separation logic. We manage to achieve this thanks to our structured stack assertion and the associated proof rules. While one's first instinct could be to treat assertions about stack values like assertions about local variables, such a system would require substantial bookkeeping, since the stack changes shape during execution. Benton [3] uses this approach for a language with a similar typed-value stack, but ends up describing the resulting proofs as "fussily baroque" and "extremely tedious to construct by hand".

$$
\begin{array}{llcl}
\text{constants} & c \in \mathcal{C}onst & ::= & \mathrm{c}_{\mathrm{i32}} \mid \mathrm{c}_{\mathrm{i64}} \mid \mathrm{c}_{\mathrm{f32}} \mid \mathrm{c}_{\mathrm{f64}} \\
\text{variables (logical/local/global)} & \nu \in \mathcal{V}ar & ::= & x \mid l_i \mid g_i, \text{where } i \in \mathbb{N} \\
\text{terms} & \tau \in \mathcal{T}erm & ::= & c \mid \nu \mid f(\tau_1 \dots \tau_n) \\
\text{heap assertions} & H, H' \in \mathcal{A}_{ph} & ::= & \bot \mid \neg H \mid H \wedge H' \mid \\
 & & & \exists x.\ H \mid p(\tau_1 \dots \tau_n) \mid \\
 & & & \mathbf{emp} \mid H * H' \mid \underset{\tau_1 < \ x \ < \tau_2}{\circledast} H \mid \\
 & & & \tau_1 \mapsto \tau_2 \mid \mathbf{size}(\tau) \\
\text{stack assertions} & S \in \mathcal{A}_s & ::= & [] \ \mid \ S :: \tau \\
\text{assertions} & P, Q \in \mathcal{A} & ::= & (S \mid H) \ \mid \ \exists x.\ P
\end{array}
$$

$$[\exists \overrightarrow{x}.\ (S \mid H)] \otimes H_f \ \triangleq\ \exists \overrightarrow{x}.\ (S \mid H * H_f) \quad \text{iff } fv(H_f) \cap \overrightarrow{x} = \varnothing$$

Figure 5 Syntax of Wasm Logic assertions.

The syntax of Wasm Logic assertions is defined in Fig. 5. Constants, c, can have one of the four WebAssembly value types. Next we have logical, local, and global variables, with local/global variables having dedicated variable names, l_i/g_i, where $i \in \mathbb{N}$. Terms can either be constants, or variables, or functions (for example, unary and binary operators).

Heap assertions are mostly familiar from traditional separation logic [35]. First, we have the pure assertions of predicate logic, including predicates $p(\tau_1 \dots \tau_n)$ over terms (for example, term equality). We also have the standard spatial assertions: $\texttt{emp}$ describes an empty heap, $H * H'$ is the separating conjunction (star), and the iterated star operator, $\circledast$, aggregates assertions composed by $*$ in the same way that $\sum$ aggregates arithmetic expressions composed by $+$. Finally, we have two WebAssembly-specific spatial assertions: the cell assertion $\tau_1 \mapsto \tau_2$ describes a single heap cell at address denoted by τ_1 with contents denoted by τ_2, and the $\textsf{\textbf{size}}(\tau)$ assertion states that the number of pages currently allocated is denoted by τ.

A stack assertion, denoted by S, is a list of terms, each of which represents the value of the corresponding stack position in the value stack. This is possible due to the size of the WebAssembly stack always being precisely known statically. Were this not true, the stack assertion would need to be able to represent that the stack may have multiple sizes, and could not be represented purely as a single list of terms. The list appends on the right, to match the conventions of the WebAssembly type system.

Finally, a Wasm Logic assertion is a two-part, possibly existentially quantified assertion consisting of a stack assertion S, and a pure/heap assertion H. We define an operator, $\otimes$, for distributing heap frames through Wasm Logic assertions, which will be used later in §3.3 to define our frame rule. The notation $\exists \overrightarrow{x}$ is a shorthand for some set of outer existentially quantified variables, while $fv(H_f)$ returns the set of *free variables* in the heap assertion H_f.

Notation. For clarity of presentation, we introduce the following notational conventions:

- (Stack Length) We denote by P_n an assertion whose stack part is of length n.
- (Type Annotations in Cell Assertions) The cell assertion $\tau_1 \mapsto \tau_2$ encodes the value of a single byte in memory. As WebAssembly values normally take up either four or eight bytes, it is convenient for us to define the corresponding shorthand, which we do by annotating the arrow with the appropriate type: $\tau_1 \mapsto_t \tau_2$. For example, we have that $\tau_1 \mapsto_{i32} \tau_2 \triangleq \tau_1 \mapsto b_0 * (\tau_1 + 1) \mapsto b_1 * (\tau_1 + 2) \mapsto b_2 * (\tau_1 + 3) \mapsto b_3$, where b_k denotes the k^{th} least significant byte of the 32-bit representation of τ_2.
- (Operator Domain) To avoid clutter, we overload all mathematical operators (e.g., $+$, $\cdot$, $\leqslant$, $\dots$) instead of explicitly stating their domain ($\mathsf{i32}$, $\mathsf{i64}$, $\mathsf{f32}$, $\mathsf{f64}$, $\mathbb{N}$, $\mathbb{Z}$, or $\mathbb{R}$) on each use. When required, we state the domain either of a single operator (e.g., $+_{\mathsf{i32}}$, $+_{\mathsf{i64}}$, $\dots$) or of a parenthesised expression (e.g., $(3.14 - 2.71 \cdot x)_{\mathsf{f64}})$), in which case the domain applies to all operators and operands of the expression. The default domain is $\mathsf{i32}$.

3.2 Wasm Logic Triple

We define a program logic for first-order, encapsulated WebAssembly modules. We base our encoding of program behaviour on *Hoare triples* [18]. Wasm Logic triples are of the form

$$\Gamma \vdash \{P\}\ e^*\ \{Q\}$$

where e^* is the WebAssembly program to be executed, P is its *pre-condition*, Q is its *post-condition*, and Γ represents the context in which the program is executed.

Before giving the interpretation of the Wasm Logic triple, we have to explain the context Γ in detail. A context contains four fields: **(1)** the *functions* field, F, containing a list of all function definitions of the module; **(2)** the *assumptions* field, A, containing a set of assertions of the form $\{P\}$ **call** i $\{Q\}$, used by the [call] rule to correctly capture mutually recursive functions; **(3)** the *labels* field, L, containing a list of assertions used to describe the behaviour of the **br** instruction; and **(4)** the *return* field, R, containing an optional return assertion, used to describe the behaviour of the **return** instruction. A context may be alternatively presented as (F, A, L, R), and any of its fields may be referenced directly: for example, $\Gamma.F$ refers to the functions field of the context. We use $P; \Gamma$ as syntactic shorthand for Γ with P appended to the head of its labels field, since this pattern occurs commonly.

Interpretation of Wasm Logic Triples. The meaning of the triple $\Gamma \vdash \{P\}\ e^*\ \{Q\}$ is, informally, as follows. Let e^* be executed from a state satisfying P. Then: if e^* terminates normally, it will terminate in a state satisfying Q; if it terminates with a **Return** v^* result, the resulting state must satisfy $\Gamma.R$; and if it terminates with a **Break** $i\ v^*$ result, the resulting state must satisfy the i-th assertion of $\Gamma.L$. A formal definition is given in §5.

3.3 Proof Rules

Basic Instructions. The proof rules for basic instructions are given in Figure 6. These rules manipulate only the stack and pure logical assertions, and can be intuitively motivated by their effects on the stack. In particular, the effect of the [select] rule is conditional on the value of τ_3: we know that it has placed exactly one value on the stack, but whether it is τ_1 or τ_2 depends on whether or not $\tau_3 \neq 0$. These rules, despite manipulating the WebAssembly stack, appear very standard: this is precisely due to our structured stack assertions.

Variable Management Instructions. We give the proof rules for variable management instructions in Figure 7 (left). Just like the rules for basic instructions, these also require an empty heap. By observing these rules, we can understand how the dedicated local/global variable names are manipulated. For example, (**get_local** i) simply puts the variable l_i on

$$\frac{}{\Gamma \vdash \{[] \mid \mathbf{emp}\}\ t.\mathbf{const}\ c\ \{[c] \mid \mathbf{emp}\}}\ [\text{const}] \qquad \frac{}{\Gamma \vdash \{[] \mid \bot\}\ \mathbf{unreachable}\ \{Q\}}\ [\text{unreachable}]$$

$$\frac{}{\Gamma \vdash \{[] \mid \mathbf{emp}\}\ \mathbf{nop}\ \{[] \mid \mathbf{emp}\}}\ [\text{nop}] \qquad \frac{}{\Gamma \vdash \{[\tau] \mid \mathbf{emp}\}\ \mathbf{drop}\ \{[] \mid \mathbf{emp}\}}\ [\text{drop}]$$

$$\frac{}{\Gamma \vdash \{[\tau_1, \tau_2, \tau_3] \mid \mathbf{emp}\}\ \mathbf{select}\ \{\exists x.\ [x] \mid \mathbf{emp} \land (\tau_3 \neq 0 \to x = \tau_1) \land (\tau_3 = 0 \to x = \tau_2)\}}\ [\text{select}]$$

$$\frac{}{\Gamma \vdash \{[\tau] \mid \mathbf{emp}\}\ t.\mathit{unop}\ \{[\mathit{unop}(\tau)] \mid \mathbf{emp}\}}\ [\text{unop}] \qquad \frac{}{\Gamma \vdash \{[\tau] \mid \mathbf{emp}\}\ t.\mathit{testop}\ \{[\mathit{testop}(\tau)] \mid \mathbf{emp}\}}\ [\text{testop}]$$

$$\frac{}{\Gamma \vdash \{[\tau_1, \tau_2] \mid \text{defined}(\mathit{binop}, \tau_1, \tau_2) \land \mathbf{emp}\}\ t.\mathit{binop}\ \{[\mathit{binop}(\tau_1, \tau_2)] \mid \mathbf{emp}\}}\ [\text{binop}]$$

$$\frac{}{\Gamma \vdash \{[\tau_1, \tau_2] \mid \mathbf{emp}\}\ t.\mathit{relop}\ \{[\mathit{relop}(\tau_1, \tau_2)] \mid \mathbf{emp}\}}\ [\text{relop}]$$

$$\frac{}{\Gamma \vdash \{[\tau] \mid \text{defined}(\mathit{cvtop}, \tau) \land \mathbf{emp}\}\ t.\mathit{cvtop}\ \{[\mathit{cvtop}(\tau)] \mid \mathbf{emp}\}}\ [\text{cvtop}]$$

Note: The defined$(\mathit{binop}, \tau_1, \tau_2)$ and defined(cvtop, τ) predicates describe conditions sufficient for binary and conversion operators to be non-trapping.

Figure 6 Proof Rules: Basic Instructions.

$$\frac{\text{isDeclaredLocal } i}{\Gamma \vdash \{[] \mid \mathbf{emp}\}\ \textbf{get\_local}\ i\ \{[l_i] \mid \mathbf{emp}\}}\ [\text{get\_local}]$$

$$\frac{\text{isDeclaredLocal } i}{\Gamma \vdash \{[x] \mid \mathbf{emp}\}\ \textbf{set\_local}\ i\ \{[] \mid \mathbf{emp} \wedge l_i = x\}}\ [\text{set\_local}]$$

$$\frac{\text{isDeclaredLocal } i}{\Gamma \vdash \{[x] \mid \mathbf{emp}\}\ \textbf{tee\_local}\ i\ \{[x] \mid \mathbf{emp} \wedge l_i = x\}}\ [\text{tee\_local}]$$

$$\frac{\text{isDeclaredGlobal } i}{\Gamma \vdash \{[] \mid \mathbf{emp}\}\ \textbf{get\_global}\ i\ \{[g_i] \mid \mathbf{emp}\}}\ [\text{get\_global}]$$

$$\frac{\text{isDeclaredGlobal } i}{\Gamma \vdash \{[x] \mid \mathbf{emp}\}\ \textbf{set\_global}\ i\ \{[] \mid \mathbf{emp} \wedge g_i = x\}}\ [\text{set\_global}]$$

$$\begin{array}{l}
\{[] \mid l_1 = 2 \wedge \mathsf{emp}\} \\
(\textbf{get\_local}\ 1) \\
\{[l_1] \mid l_1 = 2 \wedge \mathsf{emp}\} \\
\{[2] \mid l_1 = 2 \wedge \mathsf{emp}\} \\
(\textbf{i32.const}\ 3) \\
\{[2,3] \mid l_1 = 2 \wedge \mathsf{emp}\} \\
(\textbf{i32.add}) \\
\{[5] \mid l_1 = 2 \wedge \mathsf{emp}\}
\end{array}$$

Note: The (isDeclaredLocal i) and (isDeclaredGlobal i) predicates are an internal detail of the meta-theory ensuring that l_i and g_i do not refer to local/global variables that are not declared by the module. They always hold for any well-typed WebAssembly program.

Figure 7 Proof Rules: Variable Management Instructions (left); Simple Proof Sketch (right).

the top of the stack. On the other hand, (**set_global** i) requires one value from the value stack in the pre-condition, and in the post-condition has consumed it, and guarantees that g_i, the i-th global variable, holds this value.

In Figure 7 (right), we give a proof sketch of a simple WebAssembly program that uses basic and variable management instructions, illustrating how stack assertions behave. We start from the pre-condition $\{[] \mid l_1 = 2 \wedge \mathsf{emp}\}$, which tells us that the stack and the heap are empty and that the first local variable, l_1, equals 2. Executing (**get_local** 1) adds l_1 to the stack, which we can immediately replace with 2 due to our pure knowledge that $l_1 = 2$. The second line of the program pushes the constant 3 onto the stack (the top of the stack is on the *right-hand side* of the assertion). Finally, the two values are added together, and the resulting stack holds a single value, 5.

Memory Management Instructions. Proof rules for instructions that interact with the WebAssembly memory are given in Figure 8. The (t.**load**) and (t.**store**) proof rules are similar to standard separation heap rules, except that they are annotated with the type of the value in the heap, which determines the number of bytes that this value occupies, and also a static offset, which is added to the given address.

As discussed, the (**mem.size**) and (**mem.grow**) instructions allow WebAssembly to alter the memory size. The "permission" to observe the memory size is encoded using the $\textbf{size}(\tau)$ assertion, which states that the memory is currently τ pages long. This permission, however, does not imply permission to access in-bounds locations; the logic still requires $x \mapsto_t n$ to be held in order to access the location x, even if x is known to be in-bounds because **size** is held. Growing the memory using the (**mem.grow**) instruction confers ownership of all newly created locations, and leaves the index of the first newly allocated location on the stack.

Control Flow Instructions. The proof rules for WebAssembly control constructs are given in Figure 9. These rules illustrate how the labels (L) and return (R) fields of the context are used in practice. In particular, L contains a list of assertions, and the i-th assertion describes the state that has to hold if we break out of i enclosing contexts. Similarly, the R assertion describes the state that has to hold if we execute a function return.

In line with this, the precondition of (**br** i) in the [br] rule equals the i-th assertions of L. On the other hand, its post-condition is arbitrary, which is justified by the fact that any code following a **br** instruction in the same block of code cannot be reached due to the structured control flow of WebAssembly. Analogously, the precondition of a (**return**) statement in the [return] rule equals the return field of the context, and its post-condition is arbitrary. Observe the clear analogy between the role of *labs* and *ret* in the semantics and the role of L and R in the proof rules for **br** and **return**, respectively.

The main aspect of the [block] and [loop] rules is how they interact with the context. Concretely, in the [block] rule, the labels field is extended with the post-condition of the block, whereas in the [loop] rule, it is extended with its pre-condition. Bearing in mind the [br] rule, this precisely captures the WebAssembly control flow: when we break to a block, we exit the block, and when we break to a loop, we continue with the next iteration and the pre-condition of the loop acts as its invariant.

This approach is inspired by the proof rule for "structured" **goto** statements of Clint and Hoare [7], as WebAssembly's **block** and **br** opcodes replicate the structural conditions imposed by [7] on the use of **goto**. Note also that the explicit type annotations of [block] and [loop], combined with the guarantees of the WebAssembly type system, allow the rules to precisely fix the size of the stack in both the pre- and post-condition.

Next, the [if] rule branches depending on the value that is on the top of the stack. If this value is non-zero, the **then** branch is taken, and the **else** branch otherwise. As is commonplace, the post-conditions of the two **if** branches have to match.

The [br_if] rule is a conditional break. If the break is taken, the value on the top of the stack is popped, and known to be non-zero, and the instruction functions identically to **br**. The post-condition represents the case where the break is not taken: the value on the top of the stack is popped, and known to be 0.

Finally, the **br_table** instruction acts like the switch statement of modern languages, breaking to the appropriate label depending on the value on the top of the stack.

Structural Proof Rules. Structural proof rules, shown in Figure 10 and demonstrated in practice throughout §4, are needed to compose proofs together. The [seq] rule for program concatenation is inherited from standard separation logic, whereas the others are either new or require adjustment for Wasm Logic.

$$\frac{}{\Gamma \vdash \{[\tau_1] \mid (\tau_1 + \mathit{off}) \mapsto_t \tau_2\}\ t.\textsf{load}\ \mathit{off}\ \{[\tau_2] \mid (\tau_1 + \mathit{off}) \mapsto_t \tau_2\}}\ [\text{load}]$$

$$\frac{}{\Gamma \vdash \{[\tau_1, \tau_2] \mid (\tau_1 + \mathit{off}) \mapsto_t -\}\ t.\textsf{store}\ \mathit{off}\ \{[] \mid (\tau_1 + \mathit{off}) \mapsto_t \tau_2\}}\ [\text{store}]$$

$$\frac{}{\Gamma \vdash \{[] \mid \mathbf{size}(\tau)\}\ \textsf{mem.size}\ \{[\tau] \mid \mathbf{size}(\tau)\}}\ [\text{mem.size}]$$

$$\frac{}{\Gamma \vdash \{[\tau_1] \mid \mathbf{size}(\tau_2)\}\ \textsf{mem.grow}\ \left\{ \exists v.\ [v] \middle| \begin{array}{l} \left(\begin{array}{l} \circledast_{\tau_2 \leqslant i/64k < (\tau_2+\tau_1)}\ i \mapsto 0 * \mathbf{size}(\tau_2 + \tau_1) \\ \wedge\ v = \tau_2\ \wedge\ ((\tau_2 + \tau_1) \leqslant 2^{16})_{\mathbb{N}} \end{array} \right) \\ \vee\ (\mathbf{size}(\tau_2)\ \wedge\ v = -1) \end{array} \right\}}\ [\text{mem.grow}]$$

Figure 8 Proof Rules: Memory Management Instructions.

$$\frac{L!i = P}{F, A, L, R \vdash \{P\}\ \textbf{br}\ i\ \{Q\}}\,[\text{br}] \qquad \frac{}{F, A, L, R \vdash \{R\}\ \textbf{return}\ \{Q\}}\,[\text{return}]$$

$$\frac{Q_m\ ; \Gamma \vdash \{P_n\}\ e^*\ \{Q_m\}}{\Gamma \vdash \{P_n\}\ \textbf{block}\ t^n\ \to t^m\ e^*\ \textbf{end}\ \{Q_m\}}\,[\text{block}] \qquad \frac{P_n\ ; \Gamma \vdash \{P_n\}\ e^*\ \{Q_m\}}{\Gamma \vdash \{P_n\}\ \textbf{loop}\ t^n\ \to t^m\ e^*\ \textbf{end}\ \{Q_m\}}\,[\text{loop}]$$

$$\frac{\begin{array}{c}\Gamma \vdash \{S \mid H \wedge \tau \neq 0_{i32}\}\ \textbf{block}\ \mathit{tf}\ e_1^*\ \textbf{end}\ \{Q\} \\ \Gamma \vdash \{S \mid H \wedge \tau = 0_{i32}\}\ \textbf{block}\ \mathit{tf}\ e_2^*\ \textbf{end}\ \{Q\}\end{array}}{\Gamma \vdash \{S :: \tau \mid H\}\ \textbf{if}\ \mathit{tf}\ e_1^*\ \textbf{else}\ e_2^*\ \ \textbf{end}\ \{Q\}}\,[\text{if}] \qquad \frac{\Gamma \vdash \{S \mid H \wedge \tau \neq 0_{i32}\}\ \textbf{br}\ i\ \{Q\}}{\Gamma \vdash \{S :: \tau \mid H\}\ \textbf{br\_if}\ i\ \{S \mid H \wedge \tau = 0_{i32}\}}\,[\text{br\_if}]$$

$$\frac{\begin{array}{c}\forall k.\, 0 \leqslant k < \mathsf{llen}(i^*) \to \Gamma \vdash \{S \mid H \wedge \tau = k_{\mathsf{i32}}\}\ \textbf{br}\ (i^*!k)\ \{Q\} \\ \Gamma \vdash \{S \mid H \wedge \neg(0 \leqslant \tau < \mathsf{llen}(i^*))_{\mathsf{i32}}\}\ \textbf{br}\ i\ \{Q\}\end{array}}{\Gamma \vdash \{S :: \tau \mid H\}\ \textbf{br\_table}\ i^*\ i\ \{Q\}}\,[\text{br\_table}]$$

Figure 9 Proof Rules: Control Flow Instructions.

$$\frac{\Gamma \vdash \{P\}\ e_1^*\ \{Q\} \quad \Gamma \vdash \{Q\}\ e_2^*\ \{R\}}{\Gamma \vdash \{P\}\ e_1^*\ e_2^*\ \{R\}}\,[\text{seq}] \qquad \frac{F, A, L, R^? \vdash \{P\}\ e^*\ \{Q\}}{F, A, (map\ (\exists x.)\ L), (\exists x.\ R)^? \vdash \{\exists x.\ P\}\ e^*\ \{\exists x.\ Q\}}\,[\text{exists}]$$

$$\frac{F, A, L, R^? \vdash \{P\}\ e^*\ \{Q\} \quad fv(H) \cap mv(e^*) = \varnothing}{F, A, (map\ (\otimes\ H)\ L), (R \otimes H)^? \vdash \{P \otimes H\}\ e^*\ \{Q \otimes H\}}\,[\text{frame}]$$

$$\frac{\begin{array}{c}F, A, L', R'_{n'} \vdash \{P'\}\ e^*\ \{Q'\} \qquad P \Rightarrow P' \qquad Q' \Rightarrow Q \qquad \mathsf{llen}(L) = \mathsf{llen}(L') \\ \forall i < \mathsf{llen}(L). \exists L_n\ L'_{n'}.\ L!i = L_n \wedge L'!i = L'_{n'} \wedge n' \leqslant n \wedge L'_{n'} \Rightarrow L_n \quad n' \leqslant n \wedge R'_{n'} \Rightarrow R_n\end{array}}{F, A, L, R_n \vdash \{P\}\ e^*\ \{Q\}}\,[\text{consequence}]$$

$$\frac{\Gamma \vdash \{\exists \overrightarrow{x}.\ (S_p \mid H)\}\ e^*\ \{\exists \overrightarrow{y}.\ (S_q \mid H')\} \qquad fv(S_k) \cap (mv(e^*) \cup \overrightarrow{x} \cup \overrightarrow{y}) = \varnothing}{\Gamma \vdash \{\exists \overrightarrow{x}.\ (S_k; S_p \mid H)\}\ e^*\ \{\exists \overrightarrow{y}.\ (S_k; S_q \mid H')\}}\,[\text{extension}]$$

$$\frac{F, A, L, R^? \vdash \{P\}\ e^*\ \{Q\}}{F, A, (L; L_f), R \vdash \{P\}\ e^*\ \{Q\}}\,[\text{context}]$$

Note: $mv(e^*)$ denotes the set of local and global variables modified by the execution of e^*.

Figure 10 Proof Rules: Structural.

The existential elimination rule, [exists], has to eliminate the existential from all assertions in L and also the R. If we were only to eliminate the existential from the pre- and post-condition, as is standard, the rule would be unsound, as we could derive the following:

$$\frac{-, -, [([] \mid l_0 = k)], - \vdash \{[] \mid l_0 = k\}\ (\textbf{br}\ 0)\ \{Q\}}{-, -, [([] \mid l_0 = k)], - \vdash \{\exists k'.\ [] \mid l_0 = k'\}\ (\textbf{br}\ 0)\ \{\exists x.\ Q\}}\,[\text{unsound exists}]$$

which does not correspond to the intended meaning of the context, as the pre-condition of the break no longer implies its matching assertion in L. For similar reasons, the [frame] rule must frame off from all assertions in L and also the R. As shown in §4, we can derive simpler proof rules for straight-line code that do not require irrelevant manipulation of L and R.

$$\frac{\begin{array}{c} \mathit{func} = \textsf{\textbf{func}}\ t^n \to t^m\ \textsf{\textbf{local}}\ t^k\ e^* \quad S_n = [x_0..x_{n-1}] \quad \forall i.\ l_i \notin fv(S_n) \cup fv(H) \cup fv(Q_m) \\ (F, A, [Q_m], Q_m) \vdash \{[] \mid H \wedge \bigwedge_{0 \leqslant i < n}(l_i = x_i) \wedge \bigwedge_{n \leqslant i < n+k}(l_i = 0)\}\ e^*\ \{Q_m\} \end{array}}{F, A, L, R \vdash \{S_n \mid H\}\ \textsf{\textbf{callcl}}\ \mathit{func}\ \{Q_m\}}\ [\text{function}]$$

$$\frac{\{P\}\ \textsf{\textbf{call}}\ i\ \{Q\} \in A(\Gamma) \quad i < \textsf{llen}(F(\Gamma))}{\Gamma \vdash \{P\}\ \textsf{\textbf{call}}\ i\ \{Q\}}\ [\text{call}]$$

$$\frac{\forall(\{P\}\ e^*\ \{Q\}) \in \mathit{specs}.\ \Gamma \vdash \{P\}\ e^*\ \{Q\}}{\Gamma \Vdash \mathit{specs}}\ [\text{specsI}] \qquad \frac{\Gamma \Vdash \mathit{specs} \quad (\{P\}\ e^*\ \{Q\}) \in \mathit{specs}}{\Gamma \vdash \{P\}\ e^*\ \{Q\}}\ [\text{specsE}]$$

$$\frac{\begin{array}{l} \forall \mathit{spec} \in \mathit{specs}.\ \mathit{spec} = \{\_\}\ \textsf{\textbf{call}}\ \_\ \{\_\} \\ F, \mathit{specs}, [], [] \Vdash \{\ \{P\}\ \textsf{\textbf{callcl}}\ (F!j)\ \{Q\} \mid \{P\}\ \textsf{\textbf{call}}\ j\ \{Q\} \in \mathit{specs}\ \} \end{array}}{F, [], [], [] \Vdash \mathit{specs}}\ [\text{module}]$$

Figure 11 Proof Rules: Function-Related Instructions, Modules.

In addition to the standard strengthening of the pre-condition and weakening of the post-condition, the [consequence] rule allows us to weaken the assertions in L and also the R. This weakening comes with a side condition that we are not allowed to increase the number of elements on the corresponding stack, which comes from the intuition that breaking out carrying n values does not necessarily imply that we can break out with $n + 1$ values. The [consequence] rule uses the entailment relation of Wasm Logic, denoted by $P \Rightarrow Q$ and defined in the standard way in §5, Figure 16.

The two new rules introduced for Wasm Logic are [extension] and [context]. The [extension] rule is the analog of [frame] for stacks, and it allows us to arbitrarily extend the "bottom" of the stack. This, in turn, enables the proof rules of Figures 6, 7, and 8 to be generalised to arbitrary stacks, with the rules modifying only the head. The [context] rule allows us to remove unneeded assertions from L and also, potentially, R. This rule is sound because the triple encodes that e^*, when executed, will only jump to targets in L, so it is trivially correct for L to be further enlarged.

Function-Related Instructions, Modules. The proof rules for function-related instructions and modules are given in Fig. 11. We give a unified semantics to function calls in WebAssembly through the auxiliary **callcl** instruction and the corresponding [function] rule, which we now explain in detail. First, when inside a function body, if we execute (**br** 0) at top-level or (**return**) anywhere, the function terminates. For this reason, the context from which we start proving a function body has the labels and the return field set to the post-condition of the function Q_m. Next, as previously described, the function arguments are taken from the stack. Therefore, we require the length of the stack to match the number of function parameters, n, given in the function definition. Next, the n arguments themselves are transferred into the first n local variables (l_0 through l_{n-1}), whereas the remaining declared local variables (l_n through l_{n+k}) are set to 0. Finally, as local variables are declared per-function, we forbid function pre- and post-conditions from talking about local variables altogether in order to avoid name clashes. Note that, as with [block] and [loop], the function's explicit type annotation allows us to precisely fix the stack size of both the pre- and post-condition.

At the top level, we have rules for proving specifications for sets of mutually recursive functions. We follow the strategy described by Oheimb [33] and Nipkow [30]. There, each individual function body is initially proven while assuming the specifications of all other functions (the [function] rule), recursive calls and calls to other functions only use the assumptions (the [call] rule), and from this, it can be concluded that all function specifications are correct without any assumptions (the [module] rule).

4 Using Wasm Logic: A Verified B-Tree Library

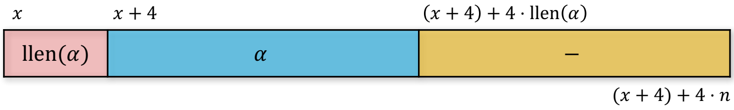
We demonstrate the applicability of Wasm Logic by specifying and verifying a simple WebAssembly B-tree library. B-trees are one of the data structures that we expect to be implemented directly in WebAssembly for efficiency reasons. In particular, a B-tree node commonly occupies an entire page of secondary storage (for example, a hard drive) and WebAssembly memory is allocated in pages. Our B-tree implementation is underpinned by the ordered, bounded array data structure, which we use to demonstrate in detail how Wasm Logic rules can be used in practice (§4.1). We focus on the two non-standard aspects of the logic: stack manipulation and the interplay between structural rules (framing, existential variable elimination, and consequence) and WebAssembly's control flow. We further describe the structure of the B-trees that we implement and present abstract specifications for some of the main B-tree operations (§4.2). The full details of our B-tree implementation are available in the accompanying technical report [47].

Additional Notation (Lists/Sets). We denote: the empty list by [ ]; the list resulting from prepending an element a to a list α by $a{:}\alpha$; concatenation of two lists α and β by $\alpha \cdot \beta$; the length of a list α by $\mathsf{llen}(\alpha)$; the n-th element of a list α by $\alpha!n$; the sublist of a list α starting from index k and containing n elements by $\mathsf{SubList}(\alpha, k, n)$; and the set corresponding to a list α by $\mathsf{ToSet}(\alpha)$. We also denote the number of elements of a set X by $\mathsf{card}(X)$.

4.1 Ordered, Bounded Arrays in WebAssembly

An ordered, bounded array (OBA) is an array whose elements are ordered and which has a fixed upper bound on the number of elements it can contain. We have found OBAs to be an appropriate data structure for representing B-tree nodes, as discussed in detail in [47].

In separation logic, it is commonplace to describe data structures using *abstract predicates* in order to abstract their implementation and simplify the textual representation of the associated proofs.[2] We define the abstract predicate for a 32-bit OBA at address x, with maximum size n and contents α, written $\mathsf{OBA}(x, n, \alpha)$. Informally, the layout of OBAs in memory, illustrated below, is as follows: the first 32-bit cell holds the length of the list α; the next $\mathsf{llen}(\alpha)$ 32-bit cells hold the contents of the list α; and the remaining $(n - \mathsf{llen}(\alpha))$ 32-bit cells constitute over-allocated space.



[2] In some separation logics, abstract predicates are distinct formal entities, but in Wasm Logic they are simply a syntactic shorthand for some particular assertion.

```
$\{[x,k] \mid \mathsf{OBA}(x,n,\alpha) \wedge 0 \leqslant k < \mathsf{llen}(\alpha)\}$
(func OBAGet [i32, i32] → [i32]
  $\{[] \mid \mathsf{OBA}(x,n,\alpha) \wedge 0 \leqslant k < \mathsf{llen}(\alpha) \wedge l_0 = x \wedge l_1 = k\}$
  frame
  |  $\{[] \mid \mathsf{emp}\}$
  |  (get_local 0)
  |  $\{[l_0] \mid \mathsf{emp}\}$
  |  extension
  |  |  $\{[] \mid \mathsf{emp}\}$
  |  |  (get_local 1)
  |  |  $\{[l_1] \mid \mathsf{emp}\}$
  |  $\{[l_0, l_1] \mid \mathsf{emp}\}$
  |  (i32.const 4)
  |  $\{[l_0, l_1, 4] \mid \mathsf{emp}\}$
  |  (i32.mul)(i32.add)
  |  $\{[l_0 + 4 \cdot l_1] \mid \mathsf{emp}\}$
  $\{[l_0 + 4 \cdot l_1] \mid \mathsf{OBA}(x,n,\alpha) \wedge 0 \leqslant k < \mathsf{llen}(\alpha) \wedge l_0 = x \wedge l_1 = k\}$
  $\{[x + 4 \cdot k] \mid \mathsf{OBA}(x,n,\alpha) \wedge 0 \leqslant k < \mathsf{llen}(\alpha)\}$ (by consequence)
  [[ Unfold $\mathsf{OBA}(x,n,\alpha)$ ]]
  $\left\{ \begin{array}{l} [x + 4 \cdot k] \mid (x \mapsto_{i32} \mathsf{llen}(\alpha) * \circledast_{0 \leqslant i < \mathsf{llen}(\alpha)} (x + 4 + 4 \cdot i \mapsto_{i32} \alpha!i) * \circledast_{\mathsf{llen}(\alpha) \leqslant i < n} (x + 4 + 4 \cdot i \mapsto_{i32} -)) \wedge \\ \qquad (\mathsf{Ordered}(\alpha) \wedge \mathsf{llen}(\alpha) \leqslant n \ \wedge (x + 4 + 4 \cdot n \leqslant \mathsf{INT32\_MAX})_{\mathbb{N}}) \wedge 0 \leqslant k < \mathsf{llen}(\alpha) \end{array} \right\}$
  frame
  |  $\{[x + 4 \cdot k] \mid x + 4 + 4 \cdot k \mapsto_{i32} \alpha!k\}$
  |  (i32.load offset=4)
  |  $\{[\alpha!k] \mid x + 4 + 4 \cdot k \mapsto_{i32} \alpha!k\}$
  $\left\{ \begin{array}{l} [\alpha!k] \mid (x \mapsto_{i32} \mathsf{llen}(\alpha) * \circledast_{0 \leqslant i < \mathsf{llen}(\alpha)} (x + 4 + 4 \cdot i \mapsto_{i32} \alpha!i) * \circledast_{\mathsf{llen}(\alpha) \leqslant i < n} (x + 4 + 4 \cdot i \mapsto_{i32} -)) \wedge \\ \qquad (\mathsf{Ordered}(\alpha) \wedge \mathsf{llen}(\alpha) \leqslant n \ \wedge (x + 4 + 4 \cdot n \leqslant \mathsf{INT32\_MAX})_{\mathbb{N}}) \wedge 0 \leqslant k < \mathsf{llen}(\alpha) \end{array} \right\}$
  [[ Fold $\mathsf{OBA}(x,n,\alpha)$ ]]
  $\{[\alpha!k] \mid \mathsf{OBA}(x,n,\alpha) \wedge 0 \leqslant k < \mathsf{llen}(\alpha)\}$
end)
$\{[\alpha!k] \mid \mathsf{OBA}(x,n,\alpha) \wedge 0 \leqslant k < \mathsf{llen}(\alpha)\}$
```

Figure 12 OBAGet: Specification and Verification.

Formally, the definition of the $\mathsf{OBA}(x,n,\alpha)$ predicate is:

$$\begin{array}{l} \mathsf{OBA}(x,n,\alpha) := (x \mapsto_{i32} \mathsf{llen}(\alpha) * \mathsf{Aseg}(x+4,\alpha) * \circledast_{\mathsf{llen}(\alpha) \leqslant i < n} (x + 4 + 4 \cdot i \mapsto_{i32} -)) \wedge \\ \qquad\qquad (\mathsf{Ordered}(\alpha) \wedge \mathsf{llen}(\alpha) \leqslant n \wedge (x + 4 + 4 \cdot n) \leqslant \mathsf{INT32\_MAX})_{\mathbb{N}}), \end{array}$$

where: the predicate $\mathsf{Aseg}(x,\alpha)$ describes the contents as an array segment:

$$\mathsf{Aseg}(x,\alpha) := \circledast_{0 \leqslant i < \mathsf{llen}(\alpha)} (x + 4 \cdot i \mapsto_{i32} \alpha!i);$$

the predicate $\mathsf{Ordered}(\alpha)$ denotes that α is ordered in ascending order:

$$\mathsf{Ordered}(\alpha) := \forall i.\, 0 < i < \mathsf{llen}(\alpha) \Rightarrow \alpha!(i-1) \leqslant \alpha!i;$$

and $\mathsf{INT32\_MAX}$ denotes the maximal positive integer of i32. Additionally, we require that the length of the list be bounded ($\mathsf{llen}(\alpha) \leqslant n$). Finally, since we are working in i32, we have to explicitly prevent overflow by stating that $(x + 4 + 4 \cdot n \leqslant \mathsf{INT32\_MAX})_{\mathbb{N}}$.

Straight-Line Code: OBAGet. We demonstrate the basics of proof sketches in Wasm Logic using the example of the $\mathsf{OBAGet}(x,k)$ function, specified and verified in Figure 12. OBAGet takes two parameters: x, denoting the memory address at which the OBA starts; and k, denoting the (non-negative) index of the OBA element to be retrieved. Assuming that k does not exceed the current OBA length, the function returns the k-th element of the OBA.

This example illustrates the following aspects of Wasm Logic: the interaction between function parameters, the stack, and the local variables; basic stack and heap manipulation; basic use of the frame, extension, and consequence rules; and predicate unfolding and folding.

In Wasm, function inputs are taken from and function outputs are put onto the stack, as specified in the pre- and post-conditions. When verifying the function body, the values of the function parameters are introduced as local variables (here, l_0 and l_1), which are propagated throughout the proof and are forgotten in the post-condition (cf. the [function] rule).

When the code being verified is straight-line, i.e. when the labels and the return fields of the context are empty, the [frame] and [consequence] rules can be used as in standard separation logic. On the other hand, the [extension] rule, which manipulates the stack analogously to [frame] manipulating the heap, can always be applied independently of the context (to limit clutter, in Figure 12, we show only one use of the [extension] rule and do not show the context Γ, since it is not relevant for this particular proof).

Predicate unfolding and folding in Wasm Logic is standard. For example, in Figure 12, we have to unfold the OBA predicate and frame off the excess resource in order to isolate the k-th element of the OBA in the heap, perform the lookup according to the [load] rule, and then frame the resource back on and fold the predicate.

Conditionals and Loops: OBAFind. We demonstrate how to reason about WebAssembly conditionals and loops in Wasm Logic using the example of the $\mathsf{OBAFind}(x, e)$ function, specified and verified in Figure 13. $\mathsf{OBAFind}$ takes two parameters: x, denoting the memory address at which the OBA starts; and e, a 32-bit integer. The function returns the index i of the first element of the OBA that is not smaller than e, or $\mathsf{llen}(\alpha)$ if such an element does not exist. The index i effectively tells us the position in the OBA at which either e appears for the first time or would be inserted.

This example addresses, among other things, the following features of Wasm Logic: interaction between conditionals, loops, and the break statement; advanced use of the frame, existential elimination, and consequence rules; and function calls. To focus on these features, we elide previously discussed details, such as predicate management, from the proof sketch.

First, observe how local variables are initialised. The function itself expects two parameters, as given by the type of the function (cf. the [function] rule). These form the first two local variables. The explicitly declared local variables, starting from index 2, are initialised to zero.

The body of the function is a loop that uses the local variable l_2 to iterate over the OBA and find its first element that is not smaller than e. First, the loop checks if l_2 is smaller than the length of the OBA. If it is, the loop terminates (by reaching the loop end), and we know that all of the elements of the OBA are smaller than e. Otherwise, it checks if the l_2-nd element of the OBA is smaller than e. If it is, the loop terminates, and we know that we have found an element not smaller than e in the OBA. Otherwise, l_2 is incremented and the loop restarts (by executing the break instruction).

For the loop construct, we establish the appropriate invariant, $([] \mid P_{inv})$, using the [consequence] rule in the standard way. This invariant essentially states that all of the previously examined elements are smaller than e. Then, following the [loop] rule, we verify the body of the loop while extending the labels field of the context with the invariant. We explicitly state modifications to the context at the point at which they first occur.

As soon as the labels or the return field of the context is not empty, the use of the frame and existential elimination becomes more involved. For example, when framing off, we have to frame off not only from the current state, but also from all of the labels, as well as from the return assertion. We illustrate this in Figure 13, using the first instruction of the loop body, (**get_local** 2), where we have to frame off P_{inv} both from the state and the labels of the context in order to apply the [get_local] rule.

```
$\{[x, e] \mid \mathsf{OBA}(x, n, \alpha)\}$
(func OBAFind [i32, i32] → [i32]
(locals i32)
  $\{[] \mid \mathsf{OBA}(x, n, \alpha) \wedge l_0 = x \wedge l_1 = e \wedge l_2 = 0\}$
  $P_{inv} : \mathsf{OBA}(x, n, \alpha) \wedge l_0 = x \wedge l_1 = e \wedge 0 \leqslant l_2 \leqslant \mathsf{llen}(\alpha) \wedge (\forall j.\, 0 \leqslant j < l_2 \Rightarrow \alpha!j < e)$
  $\{[] \mid P_{inv}\}$ (by consequence)
  (loop
  $([] \mid P_{inv}) \vdash$
    $\{[] \mid P_{inv}\}$
    frame | $([] \mid \mathsf{emp}) \vdash \{[] \mid \mathsf{emp}\}$
          | (get_local 2)
          | $([] \mid \mathsf{emp}) \vdash \{[l_2] \mid \mathsf{emp}\}$
    $\{[l_2] \mid P_{inv}\}$
    (get_local 0) (i32.load)
    $\{[l_2, \mathsf{llen}(\alpha)] \mid P_{inv}\}$
    (i32.lt)
    $C_1 : (v = 0 \Rightarrow l_2 = \mathsf{llen}(\alpha)) \wedge (v \neq 0 \Rightarrow l_2 < \mathsf{llen}(\alpha))$
    $\{\exists v.\, [v] \mid P_{inv} \wedge C_1\}$
    $(\exists v.\, [] \mid P_{inv}) \vdash \{\exists v.\, [v] \mid P_{inv} \wedge C_1\}$ (by consequence)
           | $([] \mid P_{inv}) \vdash \{[v] \mid P_{inv} \wedge C_1\}$
           | (if
           | $([] \mid P_{inv} \wedge C_2), ([] \mid P_{inv}) \vdash$
           |   $\{[] \mid P_{inv} \wedge l_2 < \mathsf{llen}(\alpha)\}$
           |   (get_local 0) (get_local 2)
           |   $\{[x, l_2] \mid P_{inv} \wedge l_2 < \mathsf{llen}(\alpha)\}$
           |   (S2) | $\vdash \{[x, l_2] \mid \mathsf{OBA}(x, n, \alpha) \wedge 0 \leqslant l_2 < \mathsf{llen}(\alpha)\}$
           |        | (callOBAGet)
           |        | $\vdash \{[\alpha!l_2] \mid \mathsf{OBA}(x, n, \alpha) \wedge 0 \leqslant l_2 < \mathsf{llen}(\alpha)\}$
           |   $([] \mid P_{inv} \wedge C_2), ([] \mid P_{inv}) \vdash \{[\alpha!l_2] \mid P_{inv} \wedge l_2 < \mathsf{llen}(\alpha)\}$
           |   (get_local 1) (i32.lt)
           |   $\{\exists v.\, [v] \mid P_{inv} \wedge l_2 < \mathsf{llen}(\alpha) \wedge (v = 0 \Rightarrow \alpha!l_2 \geqslant e) \wedge (v \neq 0 \Rightarrow \alpha!l_2 < e)\}$
    exists |   (if
           |   $C_2 : ((l_2 < \mathsf{llen}(\alpha) \wedge \alpha!l_2 \geqslant e) \vee l_2 = \mathsf{llen}(\alpha))$
           |   $([] \mid P_{inv} \wedge C_2), ([] \mid P_{inv} \wedge C_2), ([] \mid P_{inv}) \vdash$
           |     $\{[] \mid P_{inv} \wedge l_2 < \mathsf{llen}(\alpha) \wedge \alpha!l_2 < e\}$
           |     (get_local 2) (i32.const 1) (i32.add)
           |     $\{[l_2 + 1] \mid P_{inv} \wedge l_2 < \mathsf{llen}(\alpha) \wedge \alpha!l_2 < e\}$
           |     (set_local 2)
           |     $\{[] \mid \mathsf{OBA}(x, n, \alpha) \wedge l_0 = x \wedge l_1 = e \wedge l_2{-}1 < \mathsf{llen}(\alpha) \wedge$
           |         $(\forall j.\, 0 \leqslant j < l_2{-}1 \Rightarrow \alpha!j < e) \wedge \alpha!(l_2{-}1) < e\}$
           |     $\{[] \mid \mathsf{OBA}(x, n, \alpha) \wedge l_0 = x \wedge l_1 = e \wedge (\forall j.\, 0 \leqslant j < l_2 \Rightarrow \alpha!j < e) \wedge l_2 \leqslant \mathsf{llen}(\alpha)\}$
           |     $\{[] \mid P_{inv}\}$
           |     (br 2)
           |     $\{[] \mid P_{inv} \wedge C_2\}$
           |   end)
           |   $\{[] \mid P_{inv} \wedge C_2\}$
           | end)
           | $\{[] \mid P_{inv} \wedge C_2\}$
    $(\exists v.\, [] \mid P_{inv}) \vdash \{\exists v.\, [] \mid P_{inv} \wedge C_2\}$
    $([] \mid P_{inv}) \vdash \{[] \mid P_{inv} \wedge C_2\}$ (by consequence)
  end)
  $\{[] \mid P_{inv} \wedge C_2\}$
  (get_local 2)
  $\{[l_2] \mid \mathsf{OBA}(x, n, \alpha) \wedge l_0 = x \wedge l_1 = e \wedge 0 \leqslant l_2 \leqslant \mathsf{llen}(\alpha) \wedge (\forall j.\, 0 \leqslant j < l_2 \Rightarrow \alpha!j < e) \wedge C_2\}$
  $\{\exists i.\, [i] \mid \mathsf{OBA}(x, n, \alpha) \wedge l_0 = x \wedge l_1 = e \wedge l_2 = i \wedge 0 \leqslant i \leqslant \mathsf{llen}(\alpha) \wedge$
           $(\forall j.\, 0 \leqslant j < i \Rightarrow \alpha!j < e) \wedge (\forall j.\, i \leqslant j < \mathsf{llen}(\alpha) \Rightarrow e \leqslant \alpha!j)\}$
end)
$\{\exists i.\, [i] \mid \mathsf{OBA}(x, n, \alpha) \wedge 0 \leqslant i \leqslant \mathsf{llen}(\alpha) \wedge (\forall j.\, 0 \leqslant j < i \Rightarrow \alpha!j < e) \wedge (\forall j.\, i \leqslant j < \mathsf{llen}(\alpha) \Rightarrow e \leqslant \alpha!j)\}$
```

Figure 13 OBAFind: Specification and Verification.

In the general case, however, the label assertions, the return assertion, and the state need not match in resource, meaning that the [frame] rule may be unable to manipulate the label/return context. In practice, we have identified two strategies for handling this issue: **(S1)** specialising "falsey" labels/return via the [consequence] rule; or **(S2)** adjusting the context via the [context] rule.

We illustrate the first strategy using the following derivation tree:

$$\dfrac{\dfrac{-,-,[(S_1 \mid \bot),(S_2,\bot)],(S_R,\bot) \vdash \{\, S_P \mid P \,\}\ \mathrm{e}^*\ \{\, S_Q \mid Q \,\}}{\begin{array}{l}-,-,[(S_1 \mid \bot * F),(S_2 \mid \bot * F)],(S_R \mid \bot * F) \vdash \\ \qquad\qquad \{\, S_P \mid P * F \,\}\ \mathrm{e}^*\ \{\, S_Q \mid Q * F \,\}\end{array}}\,[\text{frame}] \quad \begin{array}{l}(S_1 \mid \bot * F) \Rightarrow (S_1 \mid H_1)\\ (S_2 \mid \bot * F) \Rightarrow (S_2 \mid H_2)\\ (S_R \mid \bot * F) \Rightarrow (S_R \mid H_R)\end{array}}{-,-,[(S_1 \mid H_1),(S_2 \mid H_2)],(S_R \mid H_R) \vdash \{\, S_P \mid P * F \,\}\ \mathrm{e}^*\ \{\, S_Q \mid Q * F \,\}}\,[\text{cons}]$$

This strategy takes advantage of the fact that if e^* never actually executes (for example) (**br** n), then $L!n$ can have a $\bot$ component, allowing the manufacturing of any frame through application of the [consequence] rule.

An example of the second strategy works as follows:

$$\dfrac{\dfrac{-,-,[],\texttt{None} \vdash \{\, S_P \mid P \,\}\ \mathrm{e}^*\ \{\, S_Q \mid Q \,\}}{-,-,[],\texttt{None} \vdash \{\, S_P \mid P * F \,\}\ \mathrm{e}^*\ \{\, S_Q \mid Q * F \,\}}\,[\text{frame}]}{-,-,[L_1,L_2],R \vdash \{\, S_P \mid P * F \,\}\ \mathrm{e}^*\ \{\, S_Q \mid Q * F \,\}}\,[\text{context}]$$

Here, we use the [context] rule to temporarily remove all of the labels and the return, allowing us to frame off only from the state. This strategy be seen in action immediately before the function call to OBAGet in Figure 13.

Both strategies can normally be applied before any non-break, non-return instruction, although the second strategy is preferred. However, there are occasions where the first strategy must be used. For example, if e^* executes (**br** 1), then $L!0$ can no longer be removed by [context]. However, it can still be falsified, allowing the first approach.

Existential elimination is another fundamental separation logic rule that needs to consider the context in Wasm Logic and can only be applied if all of the labels, the return, and the state have the same leading existential variable(s). This requirement can normally be established via the [consequence] rule and can be used regardless of the context and the position in the code. For example, consider the following part of the proof derivation for the first **if** statement of OBAFind (cf. Figure 13 for more details):

$$\dfrac{\dfrac{-,-,[([] \mid P_{inv})],\texttt{None} \vdash \{\, [v] \mid P_{inv} \wedge C_1 \,\}\ (\textbf{if} \ldots \textbf{end})\ \{\, [] \mid P_{inv} \wedge C_2 \,\}}{-,-,[(\exists v.\, [] \mid P_{inv})],\texttt{None} \vdash \{\, \exists v.\, [v] \mid P_{inv} \wedge C_1 \,\}\ (\textbf{if} \ldots \textbf{end})\ \{\, \exists v.\, [] \mid P_{inv} \wedge C_2 \,\}}\,[\text{exists}]}{-,-,[([] \mid P_{inv})],\texttt{None} \vdash \{\, \exists v.\, [v] \mid P_{inv} \wedge C_1 \,\}\ (\textbf{if} \ldots \textbf{end})\ \{\, [] \mid P_{inv} \wedge C_2 \,\}}\,[\text{cons}]$$

Here, we use [consequence] to add the existential v directly to the label (possible because v is not featured in P_{inv}) and remove it from the obtained post-condition (possible because v is not featured in R_2). In cases where this direct approach would lead to variable capture, we would have an additional first step of renaming the existentials appropriately.

In the first **if** statement of OBAFind, we also encounter a call to the OBAGet function. In Wasm Logic, function calls are handled in the standard way, meaning that frame and consequence are used first to isolate the appropriate pre-condition from the current state and then to massage the obtained post-condition into a desired form. For simplicity, in the code we call the functions by name, rather than by index.

Finally, we comment on the treatment of break statements, using the example of the (**br** 2) statement seen in OBAFind. Given the [br] rule, the pre-condition of that break statement must match the loop invariant $([] \mid P_{inv})$, which we establish. The post-condition,

$$\{[x, e] \mid \mathsf{OBA}_{\mathsf{nd}}(x, n, \alpha) \wedge \mathsf{llen}(\alpha) < n\}$$
$$(\textbf{func}\ \mathrm{OBAInsert}\ [\mathsf{i32}, \mathsf{i32}] \to []\ \ldots\ \textbf{end})$$
$$\left\{\begin{array}{l}\exists \alpha'.\, [] \mid \mathsf{OBA}_{\mathsf{nd}}(x, n, \alpha') \wedge \\ \qquad\qquad \mathsf{ToSet}(\alpha') = \mathsf{ToSet}(\alpha) \cup \{e\}\end{array}\right\}$$

$$\{[x, e] \mid \mathsf{OBA}_{\mathsf{nd}}(x, n, \alpha)\}$$
$$(\textbf{func}\ \mathrm{OBADelete}\ [\mathsf{i32}, \mathsf{i32}] \to []\ \ldots\ \textbf{end})$$
$$\left\{\begin{array}{l}\exists \alpha'.\, [] \mid \mathsf{OBA}_{\mathsf{nd}}(x, n, \alpha') \wedge \\ \qquad\qquad \mathsf{ToSet}(\alpha') = \mathsf{ToSet}(\alpha) \backslash \{e\}\end{array}\right\}$$

$$\{[t] \mid \textbf{size}(0) \wedge 2 \leqslant t \leqslant 4095\}$$
$$(\textbf{func}\ \mathsf{BTreeCreate}\ [\mathsf{i32}] \to [] \ldots \textbf{end})$$
$$\{[] \mid \mathsf{BTree}(t, \varnothing) \wedge 2 \leqslant t \leqslant 4095\}$$

$$\{[k] \mid \mathsf{BTree}(t, \kappa)\}$$
$$(\textbf{func}\ \mathsf{BTreeSearch}\ [\mathsf{i32}] \to [\mathsf{i32}] \ldots \textbf{end})$$
$$\left\{\begin{array}{l}\exists b.\, [b] \mid \mathsf{BTree}(t, \kappa) \wedge \\ \qquad\qquad (k \in \kappa \Rightarrow b = 1) \wedge (k \notin \kappa \Rightarrow b = 0)\end{array}\right\}$$

$$\{[k] \mid \mathsf{BTree}(t, \kappa)\}$$
$$(\textbf{func}\ \mathsf{BTreeInsert}\ [\mathsf{i32}] \to [\mathsf{i32}] \ldots \textbf{end})$$
$$\{[] \mid \mathsf{BTree}(t, \kappa \cup \{k\})\}$$

Figure 14 Specifications of: OBAInsert/OBADelete (left); B-Tree operations (right).

however, is left free in the [br] rule, and has to be chosen correctly so that the subsequent derivation makes sense. Observe that, due to the design of WebAssembly, any code found between a break statement and the end of the block of code in which it is found is dead code. In our case, this means that we never reach the exit of that **if** branch – instead, we unconditionally jump to the head of the main loop. The only way to reach the end of that **if** statement is if the test of that **if** yields zero, in which case our state would be $([] \mid P_{inv} \wedge C_2)$. Now, since the [if] rule requires the final states from both branches to be the same, we can choose precisely $([] \mid P_{inv} \wedge C_2)$ to be the post-condition of the break statement. More generally, a safe option is to always choose the post-condition of a break statement to be $([] \mid \bot)$, and from there derive any required assertion using the [consequence] rule.

Additional OBA Functions. In order to support basic B-tree operations, we also need to be able to insert/delete elements into/from an OBA. Moreover, as B-tree keys are unique (cf. §4.2), we strengthen the OBA predicate to enforce non-duplication of elements:

$$\mathsf{OBA}_{\mathsf{nd}}(x, n, \alpha) := \mathsf{OBA}(x, n, \alpha) \wedge \mathsf{llen}(\alpha) = \mathsf{card}(\mathsf{ToSet}(\alpha)).$$

Note that the previously presented OBA functions, OBAGet and OBAFind, can also be used with an $\mathsf{OBA}_{\mathsf{nd}}$. We give the specifications of OBAInsert and OBADelete in Figure 14 (left). Their corresponding proof sketches are available in [47].

4.2 B-Trees in WebAssembly

B-trees are self-balancing tree data structures that allow search, sequential access, insertion, and deletion in logarithmic time. They generalise binary search trees in that a node of a B-tree can have more than two children. B-trees are particularly well-suited for storage systems that manipulate large blocks of data, such as hard drives, and are commonly used in databases and file systems [8].

Every node x of a B-tree contains: an indicator denoting whether or not it is a leaf, λ; the number of keys that it holds, n; and the n keys themselves, $\kappa_1, \ldots \kappa_n$. Additionally, each non-leaf node contains $n + 1$ pointers to its children, $\pi_1, \ldots, \pi_{n+1}$.

The number of keys that a B-tree node may have is bounded. These bounds are expressed in terms of a fixed integer $t \geqslant 2$, called *the branching factor* of the B-tree. In particular, every node except the root must have at least $t - 1$ keys, and every node must have *at most* $2t - 1$ keys. Moreover, if a B-tree is non-empty, the root must have at least one key. Finally, all of the leaves of the B-tree have the same depth.

The keys of a B-tree are ordered, in the sense that the keys of every node are ordered (for us, in ascending order), and that every key of a non-leaf node is greater than all of the keys of its left child and smaller than all of the keys of its right child.

As an illustrative example, in Figure 15 we show a B-tree with branching factor $t = 2$ that contains all prime numbers between 1 and 100. It has 25 keys distributed over 12 nodes, with every node having at least $t - 1 = 1$ and at most $2t - 1 = 3$ keys.

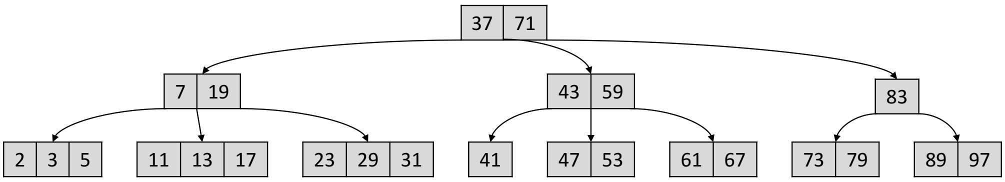


Figure 15 Prime numbers from 1 to 100 in a B-tree of branching factor two, with the λ and n parameters of the nodes elided.

Onward, we describe the layout of a B-tree in WebAssembly memory, define the associated predicates, and show the specifications for B-tree creation, search, and insertion, implemented based on the algorithms and auxiliary functions in [8]. The implementations are available, together with their accompanying proof sketches, in full in [47].

B-Tree Metadata Page. The first page of memory is reserved for keeping track of information about the state of the module. For example, one aspect of module state are the addresses of "free" pages where nodes can be allocated, and another is the root node address.

We first define what it means to be a page in memory with (non-negative integer) index n:

$$\mathsf{Page}(n) := \underset{n \cdot 64k \leqslant i < (n+1) \cdot 64k}{\circledast} (i \mapsto_{i32} -) \wedge 0 \leqslant n \wedge ((n+1) \cdot 64k \leqslant \mathsf{INT32\_MAX})_{\mathbb{N}}$$

Next, we define the predicate capturing the free pages, $\mathsf{Free}(\varphi)$, which stores the list of free pages, φ, in an $\mathsf{OBA}_{\mathsf{nd}}$, and confers ownership of all of the pages in φ. The $\mathsf{OBA}_{\mathsf{nd}}$ length $(64k/4 - 3 = 16381)$ is chosen to ensure that it can never overflow over the bounds of the metadata page, taking into account the two first elements of the page as well as the length of the array itself that is stored in the $\mathsf{OBA}_{\mathsf{nd}}$.

$$\mathsf{Free}(\varphi) := \mathsf{OBA}_{\mathsf{nd}}(8, 16381, \varphi) \underset{0 \leqslant i < \mathsf{llen}(\varphi)}{\circledast} (\mathsf{Page}(\varphi ! i));$$

The full metadata predicate, $\mathsf{Meta}(t, r, l, \varphi)$, describes the metadata page layout: t denotes the branching factor of the B-tree; r denotes the address of its root; μ denotes the current memory size in pages; and φ denotes the list of free pages.

$$\mathsf{Meta}(t, r, \mu, \varphi) := 0 \mapsto_{i32} t * 4 \mapsto_{i32} r * \mathbf{size}(\mu) * \mathsf{Free}(\varphi).$$

B-Tree Nodes. We next show the definition of the abstract predicate $\mathsf{Node}(x, \lambda, \kappa, \pi)$, which captures a B-tree node at page x, with leaf indicator λ, keys κ, and pointers π. A B-tree node takes up an entire WebAssembly page in memory, which can hold 16384 32-bit integers. The first 32-bit integer of the page is the leaf indicator (non-zero means non-leaf); the next 8191 32-bit integers hold information about the node keys; and the last 8192 32-bit integers hold information about the node pointers. The associated predicates are defined as follows:

$$\begin{aligned}\mathsf{Keys}(x,\kappa) &:= \mathsf{OBA}_{\mathsf{nd}}(x\cdot 64k+4, 8090, \kappa);\\ \mathsf{Ptrs}(x,\pi) &:= \mathsf{BA}(x\cdot 64k+32k, 8091, \pi);\\ \mathsf{Node}(x,\lambda,\kappa,\pi) &:= x\cdot 64k \mapsto_{i32} \lambda * \mathsf{Keys}(x,\kappa) * \mathsf{Ptrs}(x,\pi).\end{aligned}$$

Note that, since the pointers need not be ordered, we describe them using use a simpler bounded array predicate, $\mathsf{BA}(x,n,\alpha)$, whose definition is the same as that of the OBA predicate given in §4.1, but without the ordering requirement. Recall also that the OBAs and BAs come with a leading 32-bit integer capturing their length, meaning that the maximum number of keys/pointers our B-tree node can hold is 8090/8091 and that the maximal branching factor of our B-trees is 4095.

B-Tree Definition and Operations. Finally, we define an abstract predicate, $\mathsf{BTree}(t,\kappa)$, capturing a WebAssembly B-Tree with branching factor t and set of keys κ:

$$\mathsf{BTree}(t,\kappa) \triangleq \exists r,\mu,\varphi,\lambda,\phi.\, \mathsf{Meta}(t,r,\mu,\varphi) * \mathsf{BTreeRec}^{t,r,\mu}(r,\kappa,\lambda,\phi).$$

Due to lack of space, the full definition of the $\mathsf{BTreeRec}$ predicate is shown and explained in detail in [47]. Informally, $\mathsf{BTreeRec}^{t,r,\mu}(r',\kappa,\lambda,\phi)$ captures a subtree of a B-tree with branching factor t, root r, in a memory of size μ. This subtree has root r' and set of keys κ. Additionally, the B-tree node at r' is a leaf iff $\lambda \neq 0$ and is full iff $\phi \neq 0$.

In Figure 14 (right), we give the specifications of WebAssembly functions for basic B-tree operations: creation; search; and insertion. The specifications are abstract, in that they do not reveal any detail of the underlying implementations.

5 Soundness

The semantic interpretation of our triple and the accompanying soundness proof are informed by the approaches of de Bruin [9] and Oheimb [33]. The former gives us a semantics for **goto** which we use as the foundation for WebAssembly's **br** and **return** instructions. The latter gives us a strategy for handling mutual recursion.

Interpretation is defined against an *abstract variable store*, $\rho \in \mathcal{Sto}$. Abstract variable stores are finite partial mappings from variables to constants: $\mathcal{Sto} \equiv \mathcal{Var} \rightharpoonup \mathcal{Const}$.

Defining interpretation for terms and stack assertions is straightforward. On the other hand, interpretation of heap assertions is more involved. In traditional separation logic [35], ownership and existence of memory locations are conflated to simplify the soundness proof. This, however, cannot be done for WebAssembly: in the concrete WebAssembly linear memory, the existence of the addressable location $x+1$ implies that the addressable location x also exists. However, asserting ownership of location $x+1$ should not imply ownership of x.

To address this, we define a two-stage interpretation of heap assertions. We first define their interpretation into a set of abstract heaps, $\mathcal{AHeap}$. An abstract heap, $h \in \mathcal{AHeap}$, is a map from locations to bytes that additionally keeps track of the memory size, which may be fixed by ownership of the **size** resource. The **size** resource can be thought of as tracking the state of memory allocation, with ownership of **size** implying permission to perform allocations through **mem.grow**, similarly to the "free set" resource of [34]. Each abstract heap that is a member of the assertion interpretation represents a possible set of owned locations. Our separation algebra is defined over abstract heaps, as shown in Figure 16.

Interpretation of terms

$$[\![\cdot]\!] :: \mathcal{T}erm \Rightarrow \mathcal{S}to \Rightarrow \mathcal{C}onst$$

$$\begin{aligned}
[\![c]\!](\rho) &\triangleq c\\
[\![\nu]\!](\rho) &\triangleq \rho(\nu)\\
[\![f(\tau_1,\ldots,\tau_n)]\!](\rho) &\triangleq f([\![\tau_1]\!](\rho),\ldots,[\![\tau_n]\!](\rho))
\end{aligned}$$

Interpretation of stack assertions

$$[\![\cdot]\!] :: \mathcal{T}erm \text{ list} \Rightarrow \mathcal{S}to \Rightarrow \mathcal{C}onst \text{ list}$$

$$\begin{aligned}
[\![\ [\,]\ ]\!](\rho) &\triangleq [\,]\\
[\![\ \mathcal{S} :: \tau\ ]\!](\rho) &\triangleq [\![\ \mathcal{S}\ ]\!](\rho) :: [\![\tau]\!](\rho)
\end{aligned}$$

Abstract heaps

$$\mathit{size} ::= \bullet \mid \mathit{i32}$$

$$\begin{aligned}
\mathcal{AH}eap &::= (\mathit{i32} \rightharpoonup \mathit{byte}) \times \mathit{size}\\
(h_m,\bullet)\ \dot{\uplus}\ (h'_m,\bullet) &\triangleq (h_m \uplus h'_m,\bullet)\\
(h_m,\bullet)\ \dot{\uplus}\ (h'_m,n) &\triangleq (h_m \uplus h'_m,n)\\
(h_m,n)\ \dot{\uplus}\ (h'_m,\bullet) &\triangleq (h_m \uplus h'_m,n)
\end{aligned}$$

Note: the two last cases require that $\forall i \in \mathit{dom}(h_m) \uplus \mathit{dom}(h'_m).\ i < n * 64k$

Interpretation of pure/heap assertions

$$[\![\cdot]\!] :: \mathcal{A}_{ph} \Rightarrow \mathcal{S}to \Rightarrow \mathcal{AH}eap \text{ set}$$

$$\begin{aligned}
[\![\bot]\!](\rho) &\triangleq \varnothing\\
[\![\tau_1 = \tau_2]\!](\rho) &\triangleq \{\ h \mid [\![\tau_1]\!](\rho) = [\![\tau_2]\!](\rho)\ \}\\
[\![\tau_1 \mapsto \tau_2]\!](\rho) &\triangleq \{\ ([\![\tau_1]\!](\rho) \mapsto [\![\tau_2]\!](\rho),\bullet)\ \}\\
[\![\tau_1 \wedge \tau_2]\!](\rho) &\triangleq [\![\tau_1]\!](\rho) \cap [\![\tau_2]\!](\rho)\\
[\![\neg H]\!](\rho) &\triangleq ([\![H]\!](\rho))^c\\
[\![\exists x.\ H]\!](\rho) &\triangleq \{\ h \mid \exists c.\ h \in [\![H]\!](\rho[x \mapsto c])\ \}\\
[\![p(\tau_1,\ldots,\tau_n)]\!](\rho) &\triangleq \{\ h \mid p(\ [\![\tau_1]\!](\rho),\ldots,[\![\tau_n]\!](\rho)\ )\ \}\\
[\![H * H']\!](\rho) &\triangleq \{\ h_1\ \dot{\uplus}\ h_2 \mid h_1 \in [\![H]\!](\rho),\ h_2 \in [\![H']\!](\rho)\ \}\\
[\![\mathbf{size}(\tau)]\!](\rho) &\triangleq \{\ (\varnothing,[\![\tau]\!](\rho))\ \}
\end{aligned}$$

Interpretation of assertions

$$[\![\cdot]\!] :: \mathcal{A} \Rightarrow \mathcal{S}to \Rightarrow (\mathcal{C}onst \text{ list} \times \mathcal{AH}eap) \text{ set}$$

$$\begin{aligned}
[\![\ \mathcal{S} \mid H\ ]\!](\rho) &\triangleq \{(v^*,h) \mid v^* = [\![\mathcal{S}]\!](\rho), h \in [\![H]\!](\rho)\}\\
[\![\ \exists x.\ P\ ]\!](\rho) &\triangleq \{(v^*,h) \mid \exists x.\ (v^*,h) \in [\![P]\!](\rho[x \mapsto c])\}
\end{aligned}$$

Entailment

$$P \Rightarrow Q \triangleq \forall \rho.\ [\![P]\!](\rho) \subseteq [\![Q]\!](\rho)$$

Figure 16 Interpretations of Terms and Assertions.

Before describing the second, *reification* stage, we recall the definition of *instances* and *WebAssembly stores* as defined in the official WebAssembly specification [16] (the table fields are elided as they are only used by **call_indirect**):

$$\begin{array}{llll}
s ::= \{ & \text{funcs:} & \mathit{func\ list} & \\
 & \text{mems:} & \mathit{mem\ list} & \\
 & \text{globs:} & \mathit{glob\ list}\ \} &
\end{array}
\qquad
\begin{array}{lll}
\mathit{inst} ::= \{ & \text{faddrs:} & \mathit{nat\ list}\\
 & \text{maddr:} & \mathit{nat\ option}\\
 & \text{gaddrs:} & \mathit{nat\ list}\ \}
\end{array}
\qquad
\begin{array}{rcl}
\mathit{locs} & ::= & \mathit{Const\ list}\\
\mathit{labs} & ::= & \mathit{nat\ list}\\
\mathit{ret} & ::= & \mathit{nat\ option}
\end{array}$$

The reification stage further relates abstract heaps to WebAssembly stores, giving the concrete WebAssembly memories that are consistent with the **size** resource, such that all owned locations exist. Store reification is defined between a WebAssembly store, instance, abstract heap, abstract variable store, and function list, as follows:

$$\frac{\begin{array}{c}
\forall i.\ F!i = \text{funcs}(s)!((\text{faddrs}(\mathit{inst}))!i)\\
\forall (i,c) \in \text{fst}(h).\ c = (\text{mems}(s)!(\text{maddr }\mathit{inst}))!i\\
\text{snd}(h) \neq \bullet \Longrightarrow \mathit{pages}((\text{mems}(s)!(\text{maddr }\mathit{inst}))) = \text{snd}(h)\\
\forall (g_i,c) \in \rho.\ c = \text{globs}(s)!((\text{gaddrs}(\mathit{inst}))!i)
\end{array}}{\mathit{reifies}_{sto}(s,\ \mathit{inst},\ h,\ \rho,\ F)}\ \text{rei}_{sto}$$

We also define reification for local variables, labels, and returns:

$$\frac{\forall (l_i,v) \in \rho.\ v = \mathit{locs}!i}{\mathit{reifies}_{loc}(\mathit{locs},\ \rho)}\ \text{rei}_{loc}
\qquad
\frac{\forall i.\ (L!i = P_n) \Longleftrightarrow (\mathit{labs}!i = n)}{\mathit{reifies}_{lab}(\mathit{labs},\ L)}\ \text{rei}_{lab}
\qquad
\frac{(R = R_n) \Longleftrightarrow (\mathit{ret} = n)}{\mathit{reifies}_{ret}(\mathit{ret},\ R)}\ \text{rei}_{ret}$$

Semantic Interpretation. We define the semantic interpretation of Wasm Logic triples in Figure 17. We say that a triple (s, locs, v^*) satisfies an assertion P if its members can be reified from a member of the interpretation of P. The judgement $F, L, R \vDash \{P\}\ e^*\ \{Q\}$ means,

$$
\begin{aligned}
&F, L, R \vDash \{P\}\ e^*\ \{Q\} \triangleq \forall s, \mathit{locs}, v^*, \mathit{labs}, \mathit{labs}^f, v^{f*}, h, h^f, \rho, \mathit{ret}, s', \mathit{locs}', \mathit{res}.\ (v^*, h) \in [\![P]\!](\rho)\ \wedge \\
&\quad \mathit{reifies}_s(s, \mathit{inst}, h \uplus h^f, \rho, F) \wedge \mathit{reifies}_{loc}(\mathit{locs}, \rho) \wedge \mathit{reifies}_{lab}(\mathit{labs}, L) \wedge \mathit{reifies}_{ret}(\mathit{ret}, R)\ \wedge \\
&\qquad (s, \mathit{locs}, v_e^{f*} v_e^* e^*) \Downarrow_{\mathit{inst}}^{(\mathit{labs};\mathit{labs}^f),\mathit{ret}}\ (s', \mathit{locs}', \mathit{res}) \Longrightarrow \\
&\qquad\quad \mathit{res} \neq \textsf{Trap}\ \wedge \\
&\qquad\quad \exists h', \rho'.\ \mathit{reifies}_s(s',\ \mathit{inst},\ h' \uplus h^f,\ \rho',\ F) \wedge \mathit{reifies}_{loc}(\mathit{locs}',\ \rho')\ \wedge \\
&\qquad\qquad (\mathit{res} = \textsf{Normal}\ v^* \Rightarrow \exists v'^*.\ v^* = v^{f*} v'^*\ \wedge (v'^*, h') \in [\![Q]\!](\rho'))\ \wedge \\
&\qquad\qquad (\mathit{res} = \textsf{Break}\ i\ v^* \Rightarrow (v^*, h') \in [\![L!i]\!](\rho'))\ \wedge \\
&\qquad\qquad (\mathit{res} = \textsf{Return}\ v^* \Rightarrow (v^*, h') \in [\![R]\!](\rho'))
\end{aligned}
$$

$$F, L, R \Vvdash \mathit{specs} \triangleq (\forall(\{P\}\ e^*\ \{Q\}) \in \mathit{specs}.\ F, L, R \vDash \{P\}\ e^*\ \{Q\})$$

$$F, A, L, R \Vvdash \mathit{specs} \triangleq (F, [], \epsilon \Vvdash A \Rightarrow F, L, R \Vvdash \mathit{specs})$$

Figure 17 Semantic interpretation of the specification triple.

intuitively, that for all triples $(s, \mathit{locs}, v_e^*)$ that satisfy P, executing $(s, \mathit{locs}, (v_e^{f*})(v_e^*)e^*)$ to completion will result in a triple $(s', \mathit{locs}', \mathit{res})$ with the following properties: if res is of the form **Normal** v^*, then $(s', \mathit{locs}', v^*)$ satisfies Q; if res is of the form **Break** i v^*, then $(s', \mathit{locs}', v^*)$ satisfies $L!i$; if res is of the form **Return** v^*, then $(s', \mathit{locs}', v^*)$ satisfies R.

Note that framing is featured in three places in the definition: in the heap (h^f); in the stack (v^{f*}); and in the labels (labs^f). The heap frame is treated in the standard way. The stack frame remains in the case of a **Normal** result, but is discarded in case of the **Break** and **Return** results automatically, by WebAssembly's semantics. Finally, the labels frame encodes that the full label context during reduction may be arbitrarily large, but that only the initial labels *labs* will be targeted by the **br** instructions present in e^*.

Soundness. We now state our soundness result, fully mechanised in Isabelle/HOL.

▶ **Theorem 2** (inference_rules_sound).

$$\Gamma \Vdash \mathit{specs} \Longrightarrow \Gamma \Vvdash \mathit{specs}$$

6 Related Work

WebAssembly's official specification is given as a pen-and-paper formal semantics [16, 36], a large core of which has been mechanised in Isabelle [46]. Our mechanised soundness results build on this existing mechanisation. CT-Wasm [48] is a proposed cryptographic extension to WebAssembly's type system that protects against side-channel and information flow leaks. Aside from this, research on WebAssembly has focussed mainly on dynamic analysis. Wasabi [23] is a general purpose framework for dynamic analysis. Other work has focussed on taint tracking and binary instrumentation [14, 41]; and the detection of unauthorised WebAssembly-based cryptocurrency miners [45, 27].

Control Flow. Our proof rules for Wasm Logic's break/continue-to-block-style semi-structured control flow take inspiration from the program logic for "structured **goto**" proposed by Clint and Hoare [7] and first proven sound by de Bruin [9]. These works use a traditional Hoare Logic based on first-order logic; we have adapted their approach to our Wasm Logic. In doing so, we have observed that the existential elimination and consequence rules of Hoare logic, and the frame rule of separation logic, require modification, as detailed in §3.2.

Huisman and Jacobs [19] describe an early Hoare logic for Java, and their treatment of Java's **break** and **continue** statements in their operational semantics is similar to our use of the **Break** and **Return** execution results. However, their specifications must explicitly track in the post-condition that a statement terminates via **break** or **continue**, leading to unwieldy proof rules for loops, since separate specifications must be proven for each possible kind of termination of the loop body.

It is common for program logics which handle unstructured control flow, such as **goto** or continuations, to include a context of target assumptions in the semantics of the triple [3, 9, 42, 38]. Separation logics for such languages require a "higher-order frame rule", which distributes the frame across all such assumptions [20, 5, 51, 32, 22]. Similarly, our adaptions to the "structured **goto**" approach result in rules akin to a higher-order frame rule, despite the first-order nature of our logic.

Stack-Based Logics. Two existing program logics are defined over languages which are close to WebAssembly in their typed treatments of the stack: Benton [3], and Bannwart and Müller [1]. However, unlike Wasm Logic, these works does not propose a structured assertion syntax for the stack, instead using unstructured assertions about the values of individual stack positions. This means that assertions must be re-written with a *shift* operation whenever the shape of the stack changes due to the execution of an instruction, and irrelevant portions of the assertions cannot be framed off during local proofs without keeping track of the necessary resulting shift. Saabas and Uustalu [38] give a program logic for a low-level stack-based language with no heap. Their stack assertion is related to ours in that it has a list structure, but their proof rules rely on a global style of term substitution, and their discussion of compositionality does not appear to extend to generalising existing specifications to larger stacks. This means that one cannot conduct local proofs over just the portion of the stack that is changing in the program fragment, which we permit thanks to our [extension] rule. There has been other previous work on program logics for low-level, assembly-like languages, often incorporating a stack [29, 4, 10, 28, 2, 20]. These languages do not have type system restrictions on the stack that are as strong as WebAssembly's, and must therefore find other, less structured ways to represent the stack formally.

7 Conclusions and Future Work

We have presented Wasm Logic, a sound program logic for first-order, encapsulated WebAssembly, and proven the soundness result in Isabelle/HOL. Using Wasm Logic, we have specified and verified a simple WebAssembly B-tree library, giving abstract specifications independent of the underlying implementation.

In designing Wasm Logic, we have found the properties of WebAssembly's type system helpful for streamlining the assertions of Wasm Logic. The restrictions placed on the runtime behaviour of the WebAssembly stack by the type system are mirrored in the structured nature of our logic's stack assertions. To account for WebAssembly's uncommon control flow, we have adapted the standard separation logic triple and proof rules, inspired by the early approach of Clint and Hoare [7] for "structured **goto**".

We plan to extend Wasm Logic to handle programs made up of multiple WebAssembly modules composed together. To do this, we must extend Wasm Logic with the ability to reason about multiple, disjoint memories. Moreover, we would need to account for the JavaScript "glue code", mandatory for module interoperability. This is part of our broader goal of integrating JavaScript and WebAssembly reasoning. To achieve this, however, we will need to support some higher-order reasoning, as WebAssembly modules and functions are first-class entities in JavaScript. We also plan to extend Wasm Logic to be able to reason about higher-order pure WebAssembly code and the **call_indirect** instruction. For both of these goals, we will refer to existing work on higher-order separation logics [44, 21]. Although WebAssembly's higher-order constructs are not entirely standard, we believe that it is possible to map WebAssembly's use of the *table* as a higher-order store to the more traditional program states of other higher-order logics, and hence take direct inspiration from their proof rules and soundness approaches. Again, we would also need to account for the JavaScript component required to mutate the table.

Our long-term goal is to be able to reason, in a single formalism, about integrated JavaScript/WebAssembly programs as they will appear on the Web. We ultimately hope to integrate our work on Wasm Logic with existing work on program analysis for JavaScript [15, 12, 13] to provide a combined proof system, as well as a verification tool.

We expect WebAssembly to be extended with threads and concurrency primitives in the near future [40]. Because there is no sharing of stacks in the WebAssembly threads proposal, we believe that many of our proof rules will be fully transferrable to a hypothetical concurrent separation logic for WebAssembly with threads, although proof rules for the (now shared) heap will need revising, as will the semantic interpretation. For this, we will take inspiration from various modern concurrent separation logics [6, 43, 39].

References

1 Fabian Bannwart and Peter Müller. A Program Logic for Bytecode. *Electron. Notes Theor. Comput. Sci.*, 141(1):255–273, December 2005. doi:10.1016/j.entcs.2005.02.026.

2 Björn Bartels and Nils Jähnig. Mechanized, Compositional Verification of Low-Level Code. In Julia M. Badger and Kristin Yvonne Rozier, editors, *NASA Formal Methods*, pages 98–112, Cham, 2014. Springer International Publishing.

3 Nick Benton. A Typed, Compositional Logic for a Stack-based Abstract Machine. In *Proceedings of the Third Asian Conference on Programming Languages and Systems*, APLAS'05, pages 364–380, Berlin, Heidelberg, 2005. Springer-Verlag. doi:10.1007/11575467_24.

4 Lennart Beringer and Martin Hofmann. A Bytecode Logic for JML and Types. In *Proceedings of the 4th Asian Conference on Programming Languages and Systems*, APLAS'06, pages 389–405, Berlin, Heidelberg, 2006. Springer-Verlag. doi:10.1007/11924661_24.

5 Lars Birkedal and Hongseok Yang. Relational Parametricity and Separation Logic. In *Proceedings of the 10th International Conference on Foundations of Software Science and Computational Structures*, FOSSACS'07, pages 93–107, Berlin, Heidelberg, 2007. Springer-Verlag. URL: http://dl.acm.org/citation.cfm?id=1760037.1760047.

6 Stephen Brookes and Peter W. O'Hearn. Concurrent Separation Logic. *ACM SIGLOG News*, 3(3):47–65, August 2016. doi:10.1145/2984450.2984457.

7 M. Clint and C. A. R. Hoare. Program proving: Jumps and functions. *Acta Informatica*, 1(3):214–224, September 1972. doi:10.1007/BF00288686.

8 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

9 Arie de Bruin. Goto statements: semantics and deduction systems. *Acta Informatica*, 15(4):385–424, August 1981. doi:10.1007/BF00264536.

10 Y. Dong, S. Wang, L. Zhang, and P. Yang. Modular Certification of Low-Level Intermediate Representation Programs. In *2009 33rd Annual IEEE International Computer Software and Applications Conference*, volume 1, pages 563–570, July 2009. doi:10.1109/COMPSAC.2009.81.

11 Jonas Echterhoff. On the future of Web publishing in Unity, 2014. URL: https://blogs.unity3d.com/2014/04/29/on-the-future-of-web-publishing-in-unity/.

12 José Fragoso Santos, Petar Maksimović, Daiva Naudžiūnienė, Thomas Wood, and Philippa Gardner. JaVerT: JavaScript Verification Toolchain. *Proc. ACM Program. Lang.*, 2(POPL):50:1–50:33, December 2017. doi:10.1145/3158138.

13 José Fragoso Santos, Petar Maksimović, Gabriela Sampaio, and Philippa Gardner. JaVerT 2.0: Compositional Symbolic Execution for JavaScript. *Proc. ACM Program. Lang.*, 3(POPL):66:1–66:31, January 2019. doi:10.1145/3290379.

14 William Fu, Raymond Lin, and Daniel Inge. TaintAssembly: Taint-Based Information Flow Control Tracking for WebAssembly, 2018. arXiv:arXiv:1802.01050.

15 Philippa Anne Gardner, Sergio Maffeis, and Gareth David Smith. Towards a Program Logic for JavaScript. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 31–44, New York, NY, USA, 2012. ACM. doi:10.1145/2103656.2103663.

16 Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the Web Up to Speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 185–200, New York, NY, USA, 2017. ACM. doi:10.1145/3062341.3062363.

17 David Herman, Luke Wagner, and Alon Zakai. asm.js, 2014. URL: http://asmjs.org/spec/latest.

18 C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Commun. ACM*, 12(10):576–580, October 1969. doi:10.1145/363235.363259.

19 Marieke Huisman and Bart Jacobs. Java Program Verification via a Hoare Logic with Abrupt Termination. In Tom Maibaum, editor, *Fundamental Approaches to Software Engineering*, pages 284–303, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

20 Jonas B. Jensen, Nick Benton, and Andrew Kennedy. High-level Separation Logic for Low-level Code. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 301–314, New York, NY, USA, 2013. ACM. doi:10.1145/2429069.2429105.

21 Robbert Krebbers, Ralf Jung, Aleš Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. The Essence of Higher-Order Concurrent Separation Logic. In *Proceedings of the 26th European Symposium on Programming Languages and Systems - Volume 10201*, pages 696–723, New York, NY, USA, 2017. Springer-Verlag New York, Inc. doi:10.1007/978-3-662-54434-1_26.

22 Neelakantan R. Krishnaswami. *Verifying Higher-Order Imperative Programs with Higher-Order Separation Logic.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA., July 2011.

23 Daniel Lehmann and Michael Pradel. Wasabi: A Framework for Dynamically Analyzing WebAssembly. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 1045–1058, New York, NY, USA, 2019. ACM. doi:10.1145/3297858.3304068.

24 Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. The Java Virtual Machine Specification, 2013. URL: https://docs.oracle.com/javase/specs/jvms/se7/jvms7.pdf.

25 Mozilla. Mozilla and Epic Preview Unreal Engine 4 Running in Firefox, 2014. URL: https://blog.mozilla.org/blog/2014/03/12/mozilla-and-epic-preview-unreal-engine-4-running-in-firefox/.

26 Peter Müller and Martin Nordio. Proof-transforming Compilation of Programs with Abrupt Termination. In *Proceedings of the 2007 Conference on Specification and Verification of Component-based Systems: 6th Joint Meeting of the European Conference on Software Engineering and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, SAVCBS '07, pages 39–46, New York, NY, USA, 2007. ACM. doi:10.1145/1292316.1292321.

27 Marius Musch, Christian Wressnegger, Martin Johns, and Konrad Rieck. Web-based Cryptojacking in the Wild, 2018. arXiv:arXiv:1808.09474.

28 Magnus O. Myreen, Anthony C. J. Fox, and Michael J. C. Gordon. Hoare Logic for ARM Machine Code. In *Proceedings of the 2007 International Conference on Fundamentals of Software Engineering*, FSEN'07, pages 272–286, Berlin, Heidelberg, 2007. Springer-Verlag. URL: `http://dl.acm.org/citation.cfm?id=1775223.1775241`.

29 Magnus O. Myreen and Michael J. C. Gordon. Hoare Logic for Realistically Modelled Machine Code. In *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'07, pages 568–582, Berlin, Heidelberg, 2007. Springer-Verlag. URL: `http://dl.acm.org/citation.cfm?id=1763507.1763565`.

30 Tobias Nipkow. Hoare Logics for Recursive Procedures and Unbounded Nondeterminism. In Julian Bradfield, editor, *Computer Science Logic*, pages 103–119, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

31 Martin Nordio, Peter Müller, and Bertrand Meyer. Proof-Transforming Compilation of Eiffel Programs. In *Objects, Components, Models and Patterns, 46th International Conference, TOOLS EUROPE 2008, Zurich, Switzerland, June 30 - July 4, 2008. Proceedings*, pages 316–335, 2008. `doi:10.1007/978-3-540-69824-1_18`.

32 Peter W. O'Hearn, Hongseok Yang, and John C. Reynolds. Separation and Information Hiding. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, pages 268–280, New York, NY, USA, 2004. ACM. `doi:10.1145/964001.964024`.

33 David von Oheimb. Hoare Logic for Mutual Recursion and Local Variables. In *Proceedings of the 19th Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 168–180, London, UK, UK, 1999. Springer-Verlag. URL: `http://dl.acm.org/citation.cfm?id=646837.708364`.

34 Mohammad Raza and Philippa Gardner. Footprints in Local Reasoning. In Roberto Amadio, editor, *Foundations of Software Science and Computational Structures*, pages 201–215, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

35 John C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, LICS '02, pages 55–74, Washington, DC, USA, 2002. IEEE Computer Society. URL: `http://dl.acm.org/citation.cfm?id=645683.664578`.

36 Andreas Rossberg, Ben L. Titzer, Andreas Haas, Derek L. Schuff, Dan Gohman, Luke Wagner, Alon Zakai, J. F. Bastien, and Michael Holman. Bringing the Web Up to Speed with WebAssembly. *Commun. ACM*, 61(12):107–115, November 2018. `doi:10.1145/3282510`.

37 Ando Saabas and Tarmo Uustalu. A Compositional Natural Semantics and Hoare Logic for Low-Level Languages. *Electron. Notes Theor. Comput. Sci.*, 156(1):151–168, May 2006. `doi:10.1016/j.entcs.2005.09.031`.

38 Ando Saabas and Tarmo Uustalu. Compositional Type Systems for Stack-based Low-level Languages. In *Proceedings of the Twelfth Computing: The Australasian Theory Symposium - Volume 51*, CATS '06, pages 27–39, Darlinghurst, Australia, Australia, 2006. Australian Computer Society, Inc. URL: `http://dl.acm.org/citation.cfm?id=2523791.2523798`.

39 Filip Sieczkowski, Kasper Svendsen, Lars Birkedal, and Jean Pichon-Pharabod. A Separation Logic for Fictional Sequential Consistency. In *Programming Languages and Systems*, ESOP '15, pages 736–761, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

40 Ben Smith. Threading proposal for WebAssembly, 2018. URL: `https://github.com/WebAssembly/threads`.

41 Aron Szanto, Timothy Tamm, and Artidoro Pagnoni. Taint Tracking for WebAssembly, 2018. `arXiv:arXiv:1807.08349`.

42 Gang Tan and Andrew W. Appel. A Compositional Logic for Control Flow. In *Proceedings of the 7th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI'06, pages 80–94, Berlin, Heidelberg, 2006. Springer-Verlag. `doi:10.1007/11609773_6`.

43 Viktor Vafeiadis and Chinmay Narayan. Relaxed Separation Logic: A Program Logic for C11 Concurrency. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object*

Oriented Programming Systems Languages &#38; Applications, OOPSLA '13, pages 867–884, New York, NY, USA, 2013. ACM. doi:10.1145/2509136.2509532.

44 Carsten Varming and Lars Birkedal. Higher-Order Separation Logic in Isabelle/HOLCF. *Electron. Notes Theor. Comput. Sci.*, 218:371–389, October 2008. doi:10.1016/j.entcs.2008.10.022.

45 Wenhao Wang, Benjamin Ferrell, Xiaoyang Xu, Kevin W. Hamlen, and Shuang Hao. SEISMIC: SEcure In-lined Script Monitors for Interrupting Cryptojacks. In Javier Lopez, Jianying Zhou, and Miguel Soriano, editors, *Computer Security*, pages 122–142, Cham, 2018. Springer International Publishing.

46 Conrad Watt. Mechanising and Verifying the WebAssembly Specification. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2018, pages 53–65, New York, NY, USA, 2018. ACM. doi:10.1145/3167082.

47 Conrad Watt, Petar Maksimović, Neelakantan R. Krishnaswami, and Philippa Gardner. A Program Logic for First-Order Encapsulated WebAssembly, 2018. arXiv:1811.03479.

48 Conrad Watt, John Renner, Natalie Popescu, Sunjay Cauligi, and Deian Stefan. CT-wasm: Type-driven Secure Cryptography for the Web Ecosystem. *Proc. ACM Program. Lang.*, 3(POPL):77:1–77:29, January 2019. doi:10.1145/3290390.

49 WebAssembly Community Group. Roadmap, 2018. URL: https://webassembly.org/roadmap/.

50 WebAssembly Community Group. WebAssembly Specifications, 2018. URL: https://webassembly.github.io/spec/.

51 Hongseok Yang. Semantics of Separation-Logic Typing and Higher-Order Frame Rules. In *Proceedings of the 20th Annual IEEE Symposium on Logic in Computer Science*, LICS '05, pages 260–269, Washington, DC, USA, 2005. IEEE Computer Society. doi:10.1109/LICS.2005.47.

52 Bennet Yee, David Sehr, Gregory Dardyk, J Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native Client: A sandbox for portable, untrusted x86 native code. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2009.

53 Alon Zakai. Emscripten: An LLVM-to-JavaScript Compiler. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, OOPSLA '11, pages 301–312, New York, NY, USA, 2011. ACM. doi:10.1145/2048147.2048224.

Garbage-Free Abstract Interpretation Through Abstract Reference Counting

Noah Van Es
Software Languages Lab, Vrije Universiteit Brussel, Belgium
noah.van.es@vub.be

Quentin Stiévenart
Software Languages Lab, Vrije Universiteit Brussel, Belgium
quentin.stievenart@vub.be

Coen De Roover
Software Languages Lab, Vrije Universiteit Brussel, Belgium
coen.de.roover@vub.be

Abstract

Abstract garbage collection is the application of garbage collection to an abstract interpreter. Existing work has shown that abstract garbage collection can improve both the interpreter's precision and performance. Current approaches rely on heuristics to decide when to apply abstract garbage collection. Garbage will build up and impact precision and performance when the collection is applied infrequently, while too frequent applications will bring about their own performance overhead. A balance between these tradeoffs is often difficult to strike.

We propose a new approach to cope with the buildup of garbage in the results of an abstract interpreter. Our approach is able to eliminate all garbage, therefore obtaining the maximum precision and performance benefits of abstract garbage collection. At the same time, our approach does not require frequent heap traversals, and therefore adds little to the interpreters's running time. The core of our approach uses reference counting to detect and eliminate garbage as soon as it arises. However, reference counting cannot deal with cycles, and we show that cycles are much more common in an abstract interpreter than in its concrete counterpart. To alleviate this problem, our approach detects cycles and employs reference counting at the level of strongly connected components. While this technique in general works for any system that uses reference counting, we argue that it works particularly well for an abstract interpreter. In fact, we show formally that for the continuation store, where most of the cycles occur, the cycle detection technique only requires $\mathcal{O}(1)$ amortized operations per continuation push.

We present our approach formally, and provide a proof-of-concept implementation in the Scala-AM framework. We empirically show our approach achieves both the optimal precision and significantly better performance compared to existing approaches to abstract garbage collection.

2012 ACM Subject Classification Theory of computation → Program analysis

Keywords and phrases abstract interpretation, abstract garbage collection, reference counting

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2019.10

Supplement Material ECOOP 2019 Artifact Evaluation approved artifact available at `https://dx.doi.org/10.4230/DARTS.5.2.7`

Funding *Noah Van Es*: Funded by a PhD Fellowship of the Research Foundation – Flanders (FWO).

1 Introduction

Garbage collection (GC) is a well-known approach to automatic memory management that reclaims resources by removing items from the heap that are no longer needed. In general, there are two main approaches to garbage collection [2, 31]: tracing GC, such as the mark-and-sweep and stop-and-copy algorithms, and reference counting. Implementers of language runtimes often deem tracing GC superior. Regular reference counting cannot deal with cyclic structures in the heap, and may result in a greater overhead in terms of both memory usage and performance [50].

An abstract interpreter [12, 40] soundly over-approximates the behaviour of a concrete interpreter. The approximation has to be chosen such that the abstract interpreter terminates on all programs, but still accounts for every behaviour the program may exhibit when executed by a concrete interpreter. It is not straightforward to design an efficient abstract interpreter [11, 27, 29]. Over-approximating too much renders its results less precise, while over-approximating too little slows its convergence down. Next to their abstract value lattice, abstract interpreters feature several configuration parameters to strike this balance, such as context-sensitivity [51, 44], widening [13] and different control-flow abstractions [30, 21], the interplay of which impacts performance and precision in less than predictable manners.

1.1 Motivating Abstract Garbage Collection

Garbage collection appears to be an exception to this rule. Incorporating garbage collection in an abstract interpreter can greatly improve both its precision and performance [42, 43]. Garbage in the heap can disrupt the interpreter's behaviour. Several means of guaranteeing termination involve bounding the set of memory locations available to the interpreter. The interpreter may therefore bring garbage "back to life" whenever its allocator returns an already-occupied location. The resulting imprecision propagates to the abstract state space explored by the interpreter, causing it to explore spurious states. Garbage collection can keep the interpreter's memory, and hence the state space it explores, free of garbage and its negative effects. It has been shown to reduce the explored state space by orders of magnitude [43, 47], resulting in equally large improvements in precision and performance.

Precision Loss Illustrated. The Scheme program in Listing 1 illustrates how surviving garbage can cause an abstract interpreter to lose precision.

Listing 1 Motivating abstract garbage collection.

```
1 (let [(double (lambda (x) (+ x x)))
2       (square (lambda (y) (* y y)))
3       (apply-fn (lambda (f n) (f n)))]
4   (apply-fn double 3)
5   (apply-fn square 4))
```

Clearly, a concrete interpreter evaluates this program to `16`. As mentioned above, an abstract interpreter will sometimes reuse the same memory location for different allocations. Monovariant allocation schemes such as 0CFA [51], for instance, reuse the same memory location for all allocations of the same lexical variable [20]. Values that end up in the same memory location are joined together, for instance by computing the union of all these values.

We denote the memory location used for a variable `v` by `@v`, and use the notation `@v` $\mapsto$ `a` to indicate that the heap at memory location `@v` contains the abstract value `a`. After the first function call on line 4, the interpreter's memory will contain `@f` $\mapsto$ `{double}` and `@n` $\mapsto$ `{3}`. The contents of `@f` and `@n` can be considered garbage after this function call, but is kept in memory without abstract garbage collection. This means that the second call of `apply-fn` on line 5 will have to join its argument values with the ones of the previous call that still reside at locations `@f` and `@n`, resulting in `@f` $\mapsto$ `{double,square}` and `@n` $\mapsto$ `{3,4}`.

The garbage left at these locations is now "alive" again, and will cause further losses in precision. Due to the imprecision at location `@f`, the abstract interpreter will consider `double` as well as `square` as potential targets for the function call, causing *spurious paths* in the control flow that affect both precision and performance. For instance, the contents of `@n` will propagate to both `@x` ↦ `{3,4}` and `@y` ↦ `{3,4}`. The abstract interpreter effectively considers every possible combination of operator and operands, and evaluates this program to `{6,7,8,9,12,16}`. This is a major, but sound, over-approximation of the actual result.

Now consider that abstract GC had been applied immediately after the call on line 4, but just before the call on line 5. This application would have removed all garbage values from memory locations `@f`, `@n` and `@x`, as these locations are out of scope at this program point. The second function call would then have been explored under the precise mappings `@f` ↦ `{square}` and `@n` ↦ `{4}`, causing the abstract interpreter to evaluate this program to `{16}`.

1.2 Problem: Application Policies for Abstract Garbage Collection

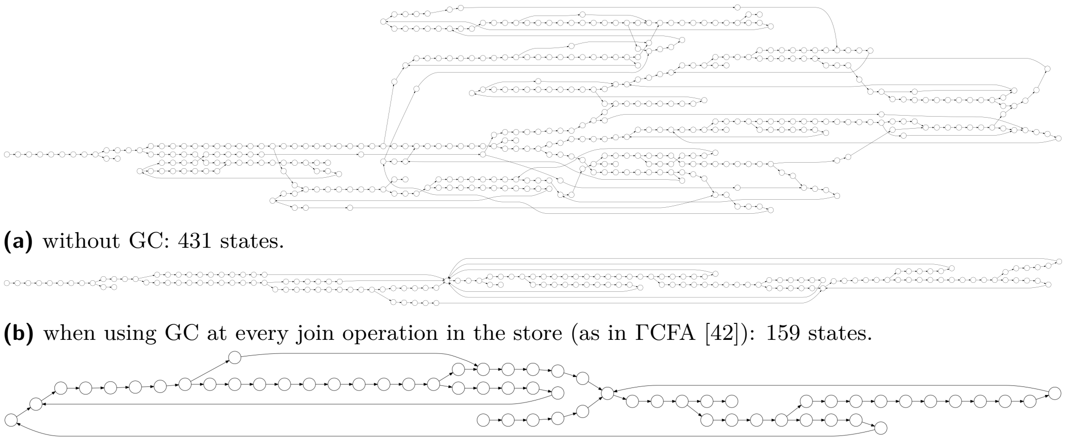
The objective of abstract GC is not to reclaim memory space, but to prevent potential precision loss due to the reuse of memory locations in the future that contain garbage values. Abstract GC therefore ought to be applied preemptively. The question is only *when*.

In the example of Listing 1, it was crucial to apply abstract GC just between the function calls on line 4 and 5 to avoid the precision loss. Of course, the abstract interpreter is not privy to this knowledge when it arrives to that program point. Nevertheless, the precision loss cannot be recovered at any later point through abstract GC.

Survey of Existing Policies. A simple, but surprisingly effective policy that is often used in practice [43, 46, 49, 14, 16] is to apply abstract GC at every evaluation step of the abstract interpreter. Doing so ensures that all garbage is eliminated immediately, rendering the abstract interpreter effectively "garbage-free" [23]. A garbage-free interpreter only explores states that do not contain any garbage, which implies that no precision is lost to garbage values at all. However, realizing this property comes at a cost: frequent GC applications can become performance-detrimental, in particular for programs with large heaps [34, 16].

Other policies apply GC less frequently to avoid this overhead, which does result in some precision loss from non-collected garbage. For instance, the TAJS static analysis [26] applies abstract GC upon every function exit. This policy requires fewer GC applications, and function exits are good GC application candidates as garbage tends to be left behind when exiting a function's scope. The policy of ΓCFA [42] (and later, LFA [39]), which pioneered the use of abstract GC, is to apply abstract GC when the address allocator returns a memory location that is already in use. As such, it triggers abstract GC whenever values need to be joined in the store, so that by design a value can never be joined with garbage. While the ΓCFA policy avoids a common form of garbage-induced imprecision, it still applies GC frequently. The explored state space can moreover not be considered "garbage-free" as some of the states remain polluted and/or spurious. In fact, in a follow-up paper [43], the authors changed their policy to an application of GC at every step for the sake of garbage-freeness.

Precision Impact of Policies. Figure 1 illustrates the impact of common abstract GC policies on the state space explored by an abstract interpreter for a slightly more complicated Scheme program, included in Appendix C. For this example, the abstract interpreter uses a monovariant allocation policy and an abstract value lattice that approximates a value by the set of all its possible types. Policies that apply GC more frequently reduce the state space to be explored, which results in both higher precision and performance, but at the same time also causes a performance overhead. Our evaluation in Section 5 shows that this overhead significantly decreases the throughput of the abstract interpreter. In terms of performance,



(a) without GC: 431 states.

(b) when using GC at every join operation in the store (as in ΓCFA [42]): 159 states.

(c) when using GC at every step (as in [43, 16, 49, 46, 14]): 60 states.

Figure 1 State space explored by the abstract interpreter for different GC policies.

policies that apply GC less frequently (e.g., the ΓCFA one) often prove more efficient in terms of performance despite the larger state space to explore. In terms of precision, applying GC at every step yields the optimal result.

1.3 Approach: Abstract Reference Counting

We aim to realize "garbage-free" abstract interpretation more efficiently, attaining optimal precision without significant compromises to performance. We propose to collect garbage in an abstract interpreter using reference counting, and argue that its evaluation with respect to tracing GC needs to be reconsidered for abstract interpreters.

Reconsidering the Disadvantages of Reference Counting. The overhead of reference counting in the concrete is less outspoken in the abstract. The reason is twofold. First, abstract interpreters typically use fewer memory locations than their concrete counterparts. As such, fewer reference counts need to be maintained. Second, abstract interpreters can tolerate more memory management overhead than concrete interpreters. The fact that reference counting cannot reclaim cyclic garbage, however, remains an issue in the abstract. In fact, we have observed cyclic garbage to be more prevalent in the abstract. Section 4 addresses this problem separately as it directly impedes realizing the "garbage-free" property.

Reconsidering the Advantages of Reference Counting. The main advantage of reference counting stems from its continuous nature. Garbage is collected under reference counting as soon as it arises. As soon as the reference count of a memory location becomes zero, reference counting immediately reclaims that location. Tracing GC approaches hold on to garbage until the next GC application. Unless the application requires minimal GC pauses, this is generally not an issue for concrete interpreters. For abstract interpreters, however, garbage held on to impacts precision – requiring abstract tracing GCs to be applied frequently.

Simple bookkeeping suffices for abstract reference counting to immediately and efficiently eliminate all garbage, precluding the need for GC application policies that seek to balance the imprecision of long-lived garbage with the overhead of collection frequency. Abstract reference counting collects garbage continuously before it can cause a precision loss.

Finally, tracing GC needs to traverse the entire heap, which can get larger for more complex programs. Such expensive traversals increase overhead when applied frequently. The cost of reference counting is mostly insensitive to the size of the heap [2].

Contributions. The contributions of this work are as follows:

- We introduce abstract reference counting as a more efficient approach to abstract garbage collection that renders the explored state space garbage-free. We present this approach formally and prove that in terms of abstract GC, it is not only sound (i.e., it only removes garbage), but also complete (i.e., it removes all garbage).
- We discern sources of garbage cycles during abstract interpretation, and present a novel cycle detection algorithm for abstract reference counting based on these observations. We show that per insertion to the abstract interpreter's continuation store, from which most cyclic garbage stems, the algorithm only requires amortized $\mathcal{O}(1)$ additional operations.
- We provide a proof-of-concept implementation in the Scala-AM framework [54, 53] and an empirical evaluation of our approach using the Gabriel benchmark suite for Scheme programs [18]. The results show that our approach achieves optimal precision and is significantly more performant compared to existing approaches to abstract GC.

We present our work in a minimal setting using the AAM approach [23, 40] to abstract interpretation. Therefore, we elide common optimizations to AAM [27] and do not consider any of its variations (e.g., AAC [30]), although our approach remains applicable in such settings as well. An advantage of AAM is that it is a systematic method for the design of a wide variety of analyses, which is applicable to programs with highly-dynamic behavior (e.g., higher-order programs with mutable state). The performance and precision improvements brought by our work should therefore carry over to such analyses. Moreover, we argue that the concepts behind abstract reference counting are sufficiently general to support its incorporation in other existing analyses that already make use of abstract tracing GC.

Structure of the Paper. We introduce the necessary background in Section 2: an abstract interpreter for ANF-style λ-calculus, into which we have incorporated an abstract tracing GC. Section 3 replaces this abstract tracing GC by abstract reference counting, resulting in a more efficient collection of garbage during abstract interpretation. Section 4 explains why garbage cycles are more common in an abstract interpreter, and proposes a technique to detect these cycles so that reference counting can reclaim all cyclic garbage. We apply this technique to obtain an abstract interpreter that is completely garbage-free. Section 5 presents our experimental results. We compare to existing approaches to abstract GC in terms of precision and performance. We conclude with a discussion of related work in Section 6.

2 Background

Figure 2 defines the syntax of a higher-order language λ_{ANF}, based on the λ-calculus in A-Normal Form [17] (ANF). ANF is a syntactic form that restricts operators and operands to atomic expressions *ae* which can be evaluated immediately without impacting the program state. This simplification can be automated, is purely cosmetic, and without loss of generality.

We start with the formal definition of the small-step operational semantics for λ_{ANF} (Section 2.1). Following the Abstracting Abstract Machines (AAM) approach [23, 40], we derive an abstract interpreter from these semantics, and discuss the impact of abstract tracing GC on the state space explored by the interpreter (Section 2.2).

$$
\begin{array}{rlll}
e \in \mathit{Exp} ::= & \mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2 & [\textsc{Let}] & f, \mathit{ae} \in \mathit{Atom} ::= v \mid \mathit{lam} \\
\mid & f\ \mathit{ae} & [\textsc{Call}] & \mathit{lam} \in \mathit{Lam} ::= \lambda x.e \\
\mid & \mathit{ae} & [\textsc{Return}] & x, v \in \mathit{Var} \text{ (a set of identifiers)}
\end{array}
$$

Figure 2 Grammar of the minimalist higher-order language λ_{ANF}.

2.1 Concrete Interpretation of λ_{ANF}

The concrete interpreter for λ_{ANF} is formulated as an abstract machine using small-step operational semantics. We systematically describe the design of this abstract machine.

State Space. Figure 3 describes the state space Σ for the concrete interpreter. A state ς consists of an expression e under evaluation, an environment ρ mapping variables to addresses, a store σ mapping addresses to values (in case of λ_{ANF}, the only values are closures), a continuation store σ_k to model the "stack", which maps continuation addresses to continuations, and the current continuation address a_k (pointing to "the top of the stack"). A continuation κ consists of the variable address to which the resulting value should be bound and the next expression, environment, and continuation address to continue the evaluation. For a concrete interpreter, addresses come from an infinite set (e.g., $\mathsf{Addr} = \mathsf{KAddr} = \mathbb{N}$).

$$
\begin{array}{rlrl}
\varsigma \in \Sigma = & \mathsf{Exp} \times \mathsf{Env} \times \mathsf{Store} & \mathit{clo} \in \mathsf{Clo} = & \mathit{Lam} \times \mathsf{Env} \\
& \times\, \mathsf{KStore} \times \mathsf{KAddr} & \kappa \in \mathsf{Kont} = & \mathsf{Addr} \times \mathit{Exp} \times \mathsf{Env} \times \mathsf{KAddr} \\
\rho \in \mathsf{Env} = & \mathit{Var} \rightharpoonup \mathsf{Addr} & l \in \mathsf{Loc} = & \mathsf{Addr} \cup \mathsf{KAddr} \\
\sigma \in \mathsf{Store} = & \mathsf{Addr} \rightharpoonup \mathsf{Clo} & a \in & \mathsf{Addr} \text{ (an infinite set)} \\
\sigma_k \in \mathsf{KStore} = & \mathsf{KAddr} \rightharpoonup \mathsf{Kont} & a_k \in & \mathsf{KAddr} \text{ (an infinite set)}
\end{array}
$$

Figure 3 State space of the concrete interpreter for λ_{ANF}.

For a state ς, we implicitly assume subscripted notations for its components so that $\varsigma = \langle e_\varsigma, \rho_\varsigma, \sigma_\varsigma, \sigma_{k_\varsigma}, a_{k_\varsigma} \rangle$. For (partial) maps, $[]$ denotes the empty map, while the notation $m[a \mapsto b]$ extends the map m so that $m[a \mapsto b](a) = b$ and $m[a \mapsto b](x) = m(x)$ for $x \neq a$.

Evaluation Rules. Atomic expressions can be evaluated in a single step, without making any modifications to the store. Therefore, we first introduce an auxiliary function $\mathcal{A}$: $\mathit{Atom} \times \mathsf{Env} \times \mathsf{Store} \to \mathit{Clo}$ to evaluate atomic expressions:

$$
\mathcal{A}(v, \rho, \sigma) = \sigma(\rho(v)) \qquad\qquad \mathcal{A}(\mathit{lam}, \rho, \sigma) = \langle \mathit{lam}, \rho \rangle
$$

Figure 4 shows the small-step transition relation $(\to)$ for λ_{ANF}. We assume that the value allocation function alloc and the continuation allocation function alloc_k always return a *fresh* address so that $\mathsf{alloc}(x, \varsigma) \notin \mathit{dom}(\sigma_\varsigma)$ and $\mathsf{alloc}_k(e, \varsigma) \notin \mathit{dom}(\sigma_{k_\varsigma})$.

We write $\varsigma_a \xrightarrow{n} \varsigma_b$ when $\varsigma_a = \varsigma_0 \to \varsigma_1 \to \ldots \to \varsigma_n = \varsigma_b$. Similarly, we define $(\xrightarrow{*})$ as the symmetric, transitive closure of $(\to)$. Note that for a concrete interpreter, this transition relation is deterministic, i.e. if $\varsigma_0 \xrightarrow{n} \varsigma_1$ and $\varsigma_0 \xrightarrow{n} \varsigma_2$ then $\varsigma_1 = \varsigma_2$.

Garbage Collection. We now add a tracing garbage collector to this concrete interpreter. We use notation and definitions similar to those used by Might and Shivers [42]. First, we define a family of auxiliary functions $\mathcal{T}_X : X \to \mathcal{P}(\mathsf{Loc})$ that return all addresses that are referenced directly by some state, environment, closure or continuation:

$$\frac{a = \mathsf{alloc}(x, \varsigma) \qquad {a_k}' = \mathsf{alloc}_k(e_1, \varsigma) \qquad \rho' = \rho[x \mapsto a]}{\underbrace{\langle \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2, \rho, \sigma, \sigma_k, a_k\rangle}_{\varsigma} \to \langle e_1, \rho, \sigma, \sigma_k[{a_k}' \mapsto \langle a, e_2, \rho', a_k\rangle], {a_k}'\rangle} \quad (\textsc{E-Let})$$

$$\frac{\mathcal{A}(f, \rho, \sigma) = \langle \lambda x.e', \rho'\rangle \qquad \mathcal{A}(ae, \rho, \sigma) = v \qquad a = \mathsf{alloc}(x, \varsigma)}{\underbrace{\langle f\ ae, \rho, \sigma, \sigma_k, a_k\rangle}_{\varsigma} \to \langle e', \rho'[x \mapsto a], \sigma[a \mapsto v], \sigma_k, a_k\rangle} \quad (\textsc{E-Call})$$

$$\frac{\mathcal{A}(ae, \rho, \sigma) = v \qquad \sigma_k(a_k) = \langle a', e', \rho', {a_k}'\rangle}{\langle ae, \rho, \sigma, \sigma_k, a_k\rangle \to \langle e', \rho', \sigma[a' \mapsto v], \sigma_k, {a_k}'\rangle} \quad (\textsc{E-Return})$$

Figure 4 Transition rules of the concrete interpreter for λ_{ANF}.

$$\mathcal{T}_\Sigma(\langle e, \rho, \sigma, \sigma_k, a_k\rangle) = \mathcal{T}_{\mathsf{Env}}(\rho) \cup \{a_k\} \qquad \mathcal{T}_{\mathsf{Env}}(\rho) = \mathsf{range}(\rho)$$

$$\mathcal{T}_{\mathsf{Clo}}(\langle \lambda x.e, \rho\rangle) = \mathcal{T}_{\mathsf{Env}}(\rho) \qquad \mathcal{T}_{\mathsf{Kont}}(\langle a, e, \rho, a_k\rangle) = \mathcal{T}_{\mathsf{Env}}(\rho) \cup \{a, a_k\}$$

Next, we introduce the adjacency relation between addresses, $(\rightsquigarrow_\varsigma) : \mathsf{Loc} \times \mathsf{Loc}$, where intuitively $\widehat{l} \rightsquigarrow_\varsigma \widehat{l'}$ means that there is a reference from address $\widehat{l}$ to address $\widehat{l'}$:

$$\frac{l \in \mathcal{T}_{\mathsf{Clo}}(\sigma_\varsigma(a))}{a \rightsquigarrow_\varsigma l} \qquad \frac{l \in \mathcal{T}_{\mathsf{Kont}}(\sigma_{k_\varsigma}(a_k))}{a_k \rightsquigarrow_\varsigma l}$$

and use the notation $(\overset{*}{\rightsquigarrow}_\varsigma)$ for its reflexive transitive closure. The function $\mathcal{R} : \Sigma \to \mathcal{P}(\mathsf{Loc})$, which computes all reachable addresses of a state, is then defined as follows:

$$\mathcal{R}(\varsigma) = \{l' \in \mathsf{Loc} \mid l \in \mathcal{T}_\Sigma(\varsigma) \land l \overset{*}{\rightsquigarrow}_\varsigma l'\}.$$

Finally, the function $\Gamma : \Sigma \to \Sigma$ applies garbage collection to a state by restricting the domain of the store and the continuation store to the addresses that are still reachable. The function Γ can be seen as a tracing garbage collector, since it defines garbage directly in terms of what is no longer reachable [2, 31]. Domain restriction is denoted by a vertical bar, so that $f|_X(x) = f(x)$ for $x \in X$ and $f|_X(x)$ is undefined for $x \notin X$.

$$\Gamma(\varsigma) = \langle e_\varsigma, \rho_\varsigma, \sigma_\varsigma|_{\mathcal{R}(\varsigma)}, \sigma_{k_\varsigma}|_{\mathcal{R}(\varsigma)}, a_k\rangle$$

To keep the abstract machine's transition relation simple and deterministic, we incorporate GC into the semantics by defining a new relation $(\to_\Gamma)$ as the composition of $(\to)$ and Γ:

$$(\to_\Gamma) = \Gamma \circ (\to)$$

This means that $(\to_\Gamma)$ first takes an evaluation step, followed by a garbage collection. This "GC at every step" strategy makes the interpreter *garbage-free*: during evaluation, for every state ς, we have that $\mathcal{R}(\varsigma) = dom(\sigma_\varsigma) \cup dom(\sigma_{k_\varsigma})$. However, for a concrete interpreter, this property does not really have an impact on the result of the evaluation. Theorem 1 states this formally: for a given state ς, taking n transition steps using $(\to_\Gamma)$ leads to the same state, *modulo GC*, as taking n transition steps using $(\to)$.

▶ **Theorem 1.** *If* $\varsigma_0 \xrightarrow{n}_\Gamma \varsigma_1$ *and* $\varsigma_0 \xrightarrow{n} \varsigma_2$*, then* $\varsigma_1 = \Gamma(\varsigma_2)$.

Proof. This follows directly from Might and Shivers [42] and is detailed in Appendix A. ◀

This confirms that concrete garbage collection does not affect the evaluation of a program. For a practical implementation, however, concrete garbage collection can still be useful when the size of the σ and σ_k components become too large to fit in computer memory.

Program Semantics. Using the transition relation $(\rightarrow_\Gamma)$, we can define a function $\mathsf{eval} : \mathit{Exp} \rightarrow \mathcal{P}(\Sigma)$ which computes all the states reachable by the concrete interpreter, starting from the initial state of the program:

$$\mathsf{eval}(e) = \{\varsigma \in \Sigma \mid \langle e, [], [], [], a_{\mathsf{halt}}\rangle \xrightarrow{*}_{\Gamma} \varsigma\}$$

where a_{halt} is a special address in the set KAddr. By computing $\mathsf{eval}(e)$, we obtain the *collecting semantics*[1] of the program e. Unfortunately, when e does not terminate, the concrete interpreter may explore an infinite amount of states in Σ. Hence, for a concrete interpreter, the set $\mathsf{eval}(e)$ is potentially infinite, and therefore not always computable.

2.2 Abstract Interpretation of λ_{ANF}

We now systematically turn this concrete interpreter for λ_{ANF} into an abstract interpreter, highlighting the important changes that we need to make in *gray*.

State Space. The main issue with the concrete interpreter in Section 2.1 is that the state space Σ is infinite, and hence the set $\mathsf{eval}(e)$ is not always computable for any program e. To solve this issue, the AAM approach [23, 40] replaces the infinite sets Addr and KAddr with finite sets $\widehat{\mathsf{Addr}}$ and $\widehat{\mathsf{KAddr}}$, respectively. One can easily verify that this suffices to keep the state space finite. Figure 5 shows the *abstract* state space $\widehat{\Sigma}$.

$$\begin{array}{rlcrl}
\widehat{\varsigma} \in \widehat{\Sigma} = & \mathsf{Exp} \times \widehat{\mathsf{Env}} \times \widehat{\mathsf{Store}} & \qquad & \widehat{\mathit{clo}} \in \widehat{\mathsf{Clo}} = & \mathit{Lam} \times \widehat{\mathsf{Env}} \\
& \times\, \widehat{\mathsf{KStore}} \times \widehat{\mathsf{KAddr}} & & \widehat{\kappa} \in \widehat{\mathsf{Kont}} = & \widehat{\mathsf{Addr}} \times \mathit{Exp} \times \widehat{\mathsf{Env}} \times \widehat{\mathsf{KAddr}} \\
\widehat{\rho} \in \widehat{\mathsf{Env}} = & \mathit{Var} \rightharpoonup \widehat{\mathsf{Addr}} & & \widehat{l} \in \widehat{\mathsf{Loc}} = & \widehat{\mathsf{Addr}} \cup \widehat{\mathsf{KAddr}} \\
\widehat{\sigma} \in \widehat{\mathsf{Store}} = & \widehat{\mathsf{Addr}} \rightarrow \mathcal{P}(\widehat{\mathsf{Clo}}) & & \widehat{a} \in & \widehat{\mathsf{Addr}} \text{ (a finite set)} \\
\widehat{\sigma}_k \in \widehat{\mathsf{KStore}} = & \widehat{\mathsf{KAddr}} \rightarrow \mathcal{P}(\widehat{\mathsf{Kont}}) & & \widehat{a}_k \in & \widehat{\mathsf{KAddr}} \text{ (a finite set)}
\end{array}$$

Figure 5 State space of the abstract interpreter for λ_{ANF}.

As the abstract interpreter can only use a finite number of addresses, it may need to reuse the same address for multiple allocations. Values that end up at the same address need to be *joined* together to obtain a sound, but finite approximation. Hence, both the store and continuation store now map addresses to a *set* of closures and a *set* of continuations, respectively. We introduce the *join operator* $\sqcup$ defined as follows:

$$(\widehat{\sigma}_1 \sqcup \widehat{\sigma}_2)(\widehat{a}) = \widehat{\sigma}_1(\widehat{a}) \cup \widehat{\sigma}_2(\widehat{a}) \qquad (\widehat{\sigma_{k1}} \sqcup \widehat{\sigma_{k2}})(\widehat{a}_k) = \widehat{\sigma_{k1}}(\widehat{a}_k) \cup \widehat{\sigma_{k2}}(\widehat{a}_k) \qquad \bot_{\widehat{\sigma}} = \bot_{\widehat{\sigma}_k} = \lambda \widehat{l}.\emptyset$$

Evaluation Rules. The atomic evaluation function $\widehat{\mathcal{A}} : \mathit{Atom} \times \widehat{\mathsf{Env}} \times \widehat{\mathsf{Store}} \rightarrow \mathcal{P}(\widehat{\mathsf{Clo}})$ for the abstract interpreter evaluates atomic expressions to a *set* of closures:

$$\widehat{\mathcal{A}}(v, \widehat{\rho}, \widehat{\sigma}) = \widehat{\sigma}(\widehat{\rho}(v)) \qquad\qquad \widehat{\mathcal{A}}(\mathit{lam}, \widehat{\rho}, \widehat{\sigma}) = \{\, \langle \mathit{lam}, \widehat{\rho}\rangle \,\}$$

[1] not to be confused with *garbage collection*

We leave the choice of sets $\widehat{\mathsf{Addr}}$ and $\widehat{\mathsf{KAddr}}$, as well as the allocation functions $\widehat{\mathsf{alloc}}$ and $\widehat{\mathsf{alloc}}_k$ open as configuration parameters of the abstract interpreter (resulting in a particular *allocation policy*). Any allocation policy yields a sound and decidable analysis [23] (as long as the sets $\widehat{\mathsf{Addr}}$ and $\widehat{\mathsf{KAddr}}$ are chosen to be finite). However, the choice is not arbitrary, as the allocation policy decides how often (determined by the size of $\widehat{\mathsf{Addr}}$ and $\widehat{\mathsf{KAddr}}$) and when (determined by $\widehat{\mathsf{alloc}}$ and $\widehat{\mathsf{alloc}}_k$) addresses need to be reused. This choice therefore affects the precision and *polyvariance* [20] of the abstract interpreter[2]. For example, the following allocation policy results in monovariant analysis [51] which reuses the same address for all allocations of the same variable:

$$\widehat{\mathsf{Addr}} = \mathit{Var} \qquad \widehat{\mathsf{KAddr}} = \mathit{Exp} \qquad \widehat{\mathsf{alloc}}(x, \widehat{\varsigma}) = x \qquad \widehat{\mathsf{alloc}}_k(e, \widehat{\varsigma}) = e$$

Figure 6 shows the abstract transition relation ($\widehat{\rightarrow}$) for λ_{ANF}. Note that, unlike in the concrete interpreter for λ_{ANF}, the transition relation is no longer deterministic.

$$\frac{\widehat{a} = \widehat{\mathsf{alloc}}(x, \widehat{\varsigma}) \qquad \widehat{a}'_k = \widehat{\mathsf{alloc}}_k(e_1, \widehat{\varsigma}) \qquad \widehat{\rho}' = \widehat{\rho}[x \mapsto \widehat{a}]}{\underbrace{\langle \mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2, \widehat{\rho}, \widehat{\sigma}, \widehat{\sigma}_k, \widehat{a}_k \rangle}_{\hat{\varsigma}} \widehat{\rightarrow} \langle e_1, \widehat{\rho}, \widehat{\sigma}, \widehat{\sigma}_k \sqcup [\widehat{a}'_k \mapsto \{ \langle \widehat{a}, e_2, \widehat{\rho}', \widehat{a}_k \rangle \}], \widehat{a}'_k \rangle} \quad \text{(E-Let)}$$

$$\frac{\widehat{\mathcal{A}}(f, \widehat{\rho}, \widehat{\sigma}) \ni \langle \lambda x.e', \widehat{\rho}' \rangle \qquad \widehat{\mathcal{A}}(\mathrm{ae}, \widehat{\rho}, \widehat{\sigma}) = \widehat{v} \qquad \widehat{a} = \widehat{\mathsf{alloc}}(x, \widehat{\varsigma})}{\underbrace{\langle f\ \mathrm{ae}, \widehat{\rho}, \widehat{\sigma}, \widehat{\sigma}_k, \widehat{a}_k \rangle}_{\hat{\varsigma}} \widehat{\rightarrow} \langle e', \widehat{\rho}'[x \mapsto \widehat{a}], \widehat{\sigma} \sqcup [\widehat{a} \mapsto \widehat{v}], \widehat{\sigma}_k, \widehat{a}_k \rangle} \quad \text{(E-Call)}$$

$$\frac{\widehat{\mathcal{A}}(\mathrm{ae}, \widehat{\rho}, \widehat{\sigma}) = \widehat{v} \qquad \widehat{\sigma}_k(\widehat{a}_k) \ni \langle \widehat{a}', e', \widehat{\rho}', \widehat{a}'_k \rangle}{\langle \mathrm{ae}, \widehat{\rho}, \widehat{\sigma}, \widehat{\sigma}_k, \widehat{a}_k \rangle \widehat{\rightarrow} \langle e', \widehat{\rho}', \widehat{\sigma} \sqcup [\widehat{a}' \mapsto \widehat{v}], \widehat{\sigma}_k, \widehat{a}'_k \rangle} \quad \text{(E-Return)}$$

Figure 6 Transition rules of the abstract interpreter for λ_{ANF}.

Garbage Collection. Just as in the concrete interpreter, we can perform garbage collection in the abstract interpreter (known as *abstract garbage collection*) by removing all addresses from $\widehat{\sigma}_{\hat{\varsigma}}$ and $\widehat{\sigma_{k\hat{\varsigma}}}$ that are no longer reachable from $\hat{\varsigma}$. The definitions remain mostly unchanged:

$$\widehat{\mathcal{T}}_{\widehat{\Sigma}}(\langle e, \widehat{\rho}, \widehat{\sigma}, \widehat{\sigma}_k, \widehat{a}_k \rangle) = \widehat{\mathcal{T}}_{\widehat{\mathsf{Env}}}(\widehat{\rho}) \cup \{\widehat{a}_k\} \qquad \widehat{\mathcal{T}}_{\widehat{\mathsf{Env}}}(\widehat{\rho}) = \mathsf{range}(\widehat{\rho})$$

$$\widehat{\mathcal{T}}_{\widehat{\mathsf{Clo}}}(\langle \lambda x.e, \widehat{\rho} \rangle) = \widehat{\mathcal{T}}_{\widehat{\mathsf{Env}}}(\widehat{\rho}) \qquad \widehat{\mathcal{T}}_{\widehat{\mathsf{Kont}}}(\langle a, e, \widehat{\rho}, \widehat{a}_k \rangle) = \widehat{\mathcal{T}}_{\widehat{\mathsf{Env}}}(\widehat{\rho}) \cup \{\widehat{a}, \widehat{a}_k\}$$

$$\widehat{\mathcal{T}}_{\mathcal{P}(\widehat{\mathsf{Clo}})}(\widehat{\mathsf{V}}) = \bigcup_{\widehat{clo} \in \widehat{\mathsf{V}}} \widehat{\mathcal{T}}_{\widehat{\mathsf{Clo}}}(\widehat{clo}) \qquad \widehat{\mathcal{T}}_{\mathcal{P}(\widehat{\mathsf{Kont}})}(\widehat{\mathsf{K}}) = \bigcup_{\widehat{\kappa} \in \widehat{\mathsf{K}}} \widehat{\mathcal{T}}_{\widehat{\mathsf{Kont}}}(\widehat{\kappa})$$

Similarly, the adjacency relation $(\widehat{\rightsquigarrow}_{\hat{\varsigma}}) : \widehat{\mathsf{Loc}} \times \widehat{\mathsf{Loc}}$ is adapted for abstract addresses:

$$\frac{\widehat{l} \in \widehat{\mathcal{T}}_{\mathcal{P}(\widehat{\mathsf{Clo}})}(\widehat{\sigma}_{\hat{\varsigma}}(\widehat{a}))}{\widehat{a} \mathrel{\widehat{\rightsquigarrow}_{\hat{\varsigma}}} \widehat{l}} \qquad \frac{\widehat{l} \in \widehat{\mathcal{T}}_{\mathcal{P}(\widehat{\mathsf{Kont}})}(\widehat{\sigma_{k\hat{\varsigma}}}(\widehat{a}_k))}{\widehat{a}_k \mathrel{\widehat{\rightsquigarrow}_{\hat{\varsigma}}} \widehat{l}}$$

[2] For the sake of simplicity, our abstract interpreter does not include a timestamp component (as in [23]), which could be used to express more complex allocation policies such as k-CFA with $k \geq 1$.

All reachable addresses can then be computed using $\widehat{\mathcal{R}} : \widehat{\Sigma} \to \mathcal{P}(\widehat{\mathsf{Loc}})$:

$$\widehat{\mathcal{R}}(\widehat{\varsigma}) = \{\widehat{l'} \in \widehat{\mathsf{Loc}} \mid \widehat{l} \in \widehat{\mathcal{T}}_{\widehat{\Sigma}}(\widehat{\varsigma}) \wedge \widehat{l} \overset{*}{\widehat{\rightsquigarrow}}_{\widehat{\varsigma}} \widehat{l'}\}.$$

and the abstract garbage collection function $\widehat{\Gamma} : \widehat{\Sigma} \to \widehat{\Sigma}$ is defined as follows:

$$\widehat{\Gamma}(\widehat{\varsigma}) = \langle e_{\widehat{\varsigma}}, \widehat{\rho}_{\widehat{\varsigma}}, \widehat{\sigma}_{\widehat{\varsigma}}|_{\widehat{\mathcal{R}}(\widehat{\varsigma})}, \widehat{\sigma_k}_{\widehat{\varsigma}}|_{\widehat{\mathcal{R}}(\widehat{\varsigma})}, \widehat{a_k}_{\widehat{\varsigma}}\rangle$$

where $f|_X(x) = f(x)$ for $x \in X$ and $f|_X(x) = \emptyset$ for $x \not\in X$. The new abstract transition relation $(\widehat{\rightarrow}_{\widehat{\Gamma}})$ can again be defined as a composition of $\widehat{\Gamma}$ and $(\widehat{\rightarrow})$:

$$(\widehat{\rightarrow}_{\widehat{\Gamma}}) = \widehat{\Gamma} \circ (\widehat{\rightarrow})$$

Applying garbage collection at every evaluation step ensures that we end up with a *garbage-free* abstract interpreter. More precisely, if we define $\widehat{\mathcal{S}} : \widehat{\Sigma} \to \mathcal{P}(\widehat{\mathsf{Loc}})$, so that $\widehat{\mathcal{S}}(\widehat{\varsigma})$ is the set of addresses bound in the stores of $\widehat{\varsigma}$, as:

$$\widehat{\mathcal{S}}(\widehat{\varsigma}) = \{\widehat{a} \in \widehat{\mathsf{Addr}} \mid \widehat{\sigma}_{\widehat{\varsigma}}(\widehat{a}) \neq \emptyset\} \cup \{\widehat{a}_k \in \widehat{\mathsf{KAddr}} \mid \widehat{\sigma_k}_{\widehat{\varsigma}}(\widehat{a}_k) \neq \emptyset\}$$

Then we define garbage-free as follows:

▶ **Definition 2** (Garbage-free). *A state $\widehat{\varsigma}$ is* garbage-free *iff $\widehat{\mathcal{R}}(\widehat{\varsigma}) = \widehat{\mathcal{S}}(\widehat{\varsigma})$, or equivalently: $\widehat{\Gamma}(\widehat{\varsigma}) = \widehat{\varsigma}$. A transition relation $(\widehat{\rightarrow})$ is* garbage-free *iff it preserves garbage-freeness. That is, $(\widehat{\rightarrow})$ is garbage-free iff for every garbage-free state $\widehat{\varsigma}$ where $\widehat{\varsigma} \,\widehat{\rightarrow}\, \widehat{\varsigma}'$, $\widehat{\varsigma}'$ is garbage-free.*

▶ **Theorem 3.** $(\widehat{\rightarrow}_{\widehat{\Gamma}})$ *is garbage-free*

Proof. By definition, $\widehat{\Gamma}(\widehat{\varsigma})$ is always a garbage-free state, and $(\widehat{\rightarrow}_{\widehat{\Gamma}})$ applies $\widehat{\Gamma}$ to the resulting state. Therefore, every resulting state of $(\widehat{\rightarrow}_{\widehat{\Gamma}})$ is garbage-free, hence $(\widehat{\rightarrow}_{\widehat{\Gamma}})$ is garbage-free. ◀

By design, every state produced by $\widehat{\varsigma}$ produced by $(\widehat{\rightarrow}_{\widehat{\Gamma}})$ is garbage-free (Theorem 3). Note however, that applying garbage collection at every step, as in $(\widehat{\rightarrow}_{\widehat{\Gamma}})$, may not be practical, since computing the set $\widehat{\mathcal{R}}(\widehat{\varsigma})$ at every evaluation step causes a significant performance overhead. In a practical implementation, one may choose to apply abstract garbage collection at less regular intervals; however, in doing so the abstract interpreter is no longer guaranteed to be garbage-free (i.e., in general, $\widehat{\mathcal{R}}(\widehat{\varsigma}) \subseteq \widehat{\mathcal{S}}(\widehat{\varsigma})$).

Theorem 1 previously showed that this garbage-free property does not really matter for a concrete interpreter. This is no longer the case for the abstract interpreter: as a counter-example, consider a monovariant analysis of the program presented in Listing 1. If the abstract interpreter uses the transition relation $(\widehat{\rightarrow})$, then after the function call on line 4 we have that $\widehat{\sigma}_{\widehat{\varsigma}}(f) = \{\widehat{clo}_{\texttt{double}}\}$, where $\widehat{clo}_{\texttt{double}}$ is the closure of the `double` function that f was bound to. Note that at that program point, $f \not\in \widehat{\mathcal{R}}(\widehat{\varsigma})$, but since $(\widehat{\rightarrow})$ does not perform abstract GC, the binding is not removed from $\widehat{\sigma}$. Therefore, at the function call of line 5, when f needs to be bound to the closure $\widehat{clo}_{\texttt{square}}$ of the `square` function, the values are joined together using $\sqcup$ so that $\widehat{\sigma}_{\widehat{\varsigma}'}(f) = \{\widehat{clo}_{\texttt{double}}, \widehat{clo}_{\texttt{square}}\}$. Intuitively, this means that the abstract interpreter does not know exactly which of these two closures f is bound to, and to soundly over-approximate any possible execution behaviour, it needs to consider both options. This imprecision therefore introduces spurious control flow into the abstract interpreter, which implies that it will explore more states than necessary. Also note that at that point, $f \in \widehat{\mathcal{R}}(\widehat{\varsigma}')$, so applying $\widehat{\Gamma}$ to collect garbage is no longer able to recover this loss of precision in $\widehat{\varsigma}'$. In contrast, if we keep the abstract interpreter garbage-free by using the $(\widehat{\rightarrow}_{\widehat{\Gamma}})$

transition relation, we can avoid this precision loss. After the function call at line 4, since $f \notin \widehat{\mathcal{R}}(\hat{\varsigma})$, and since all states produced by $(\widehat{\rightarrow}_{\widehat{\Gamma}})$ are garbage-free, we have that $f \notin \widehat{\mathcal{S}}(\hat{\varsigma})$, which implies that $\hat{\sigma}_{\hat{\varsigma}}(f) = \emptyset$. Therefore, for the next call to `apply-fn`, $\hat{\sigma}_{\hat{\varsigma}'}(f) = \{\widehat{clo}_{\texttt{square}}\}$, so that the abstract interpreter knows precisely which closure is bound to f.

In general, an abstract interpreter loses precision whenever two non-empty sets are joined in the store (or continuation store) using $\sqcup$ (which happens when the abstract interpreter reuses an address that is already allocated). As common wisdom puts it: *"merging [i.e., join] is the enemy of precision"* [29], and as such it should only be used sparingly. This is exactly what abstract garbage collection achieves by emptying all sets at addresses that are no longer reachable, so that these sets can no longer be merged with in the future.

Program Semantics. The *abstract* collecting semantics of a program can now be defined using the function $\widehat{\mathsf{eval}} : Exp \rightarrow \mathcal{P}(\widehat{\Sigma})$ which computes all the states reachable by the abstract interpreter, starting from the initial state of the program:

$$\widehat{\mathsf{eval}}(e) = \{\hat{\varsigma} \in \widehat{\Sigma} \mid \langle e, [], \bot_{\hat{\sigma}}, \bot_{\hat{\sigma}_k}, \hat{a}_{\mathsf{halt}} \rangle \overset{*}{\widehat{\rightarrow}}_{\widehat{\Gamma}} \hat{\varsigma}\}$$

where $\hat{a}_{\mathsf{halt}}$ is a special address in the set $\widehat{\mathsf{KAddr}}$. As $\widehat{\Sigma}$ is finite, for any program e it is guaranteed that $\widehat{\mathsf{eval}}(e)$ is finite and therefore computable. We can reason over the behaviour of e by reasoning over $\widehat{\mathsf{eval}}(e)$ to obtain a sound and decidable program analysis.

The definition of $\widehat{\mathsf{eval}}$ reveals another benefit of abstract garbage collection. Garbage-free abstract interpretation reduces the state space that needs to be explored (i.e., the size of $\widehat{\mathsf{eval}}(e)$), which improves both its precision and performance while still producing a sound over-approximation of the program's concrete semantics. That is, a garbage-free abstract interpreter only explores a smaller subset of $\widehat{\Sigma}$ where $\widehat{\Gamma}(\hat{\varsigma}) = \hat{\varsigma}$. In contrast, an abstract interpreter that is not garbage-free may explore equivalent states multiple times, i.e., it may explore $\hat{\varsigma}_1, \hat{\varsigma}_2, ..., \hat{\varsigma}_k$ where $\hat{\varsigma}_1 \neq \hat{\varsigma}_2 \neq ... \neq \hat{\varsigma}_k$, but $\widehat{\Gamma}(\hat{\varsigma}_1) = \widehat{\Gamma}(\hat{\varsigma}_2) = ... = \widehat{\Gamma}(\hat{\varsigma}_k)$. This can be seen clearly in Figure 1, where performing a GC before every join as in ΓCFA [43] results in 159 states, while performing a GC for every state results in 60 states.

3 Abstract Reference Counting

Section 2.2 detailed the prototypical design of an abstract interpreter that is garbage-free thanks to abstract GC [42]. The benefits of being garbage-free include substantial improvements to both the precision and performance of the abstract interpreter. The cost of realizing this property through abstract tracing GC is the need for an application policy that collects garbage at every evaluation step, which results in its own performance overhead.

In this section, we develop a more efficient design for garbage-free abstract interpreters. Instead of computing $\widehat{\mathcal{R}}(\hat{\varsigma})$ at every step (as in tracing garbage collection), the main idea is to keep track of all references to every address $\hat{l}$ in $\hat{\varsigma}$ (as in reference counting), from which it is possible to determine whether $\hat{l} \in \widehat{\mathcal{R}}(\hat{\varsigma})$. When there are no more references to $\hat{l}$, we have that $\hat{l} \notin \widehat{\mathcal{R}}(\hat{\varsigma})$, and the memory location can be reclaimed. If the abstract interpreter consistently collects all addresses without references, and there are no cyclic references in memory[3], then for every address $\hat{l}$ that still has references, we have that $\hat{l} \in \widehat{\mathcal{R}}(\hat{\varsigma})$.

[3] which means that $\forall \hat{l}_1, \hat{l}_2 \in \widehat{\mathsf{Loc}}, \neg(\hat{l}_1 \overset{*}{\widehat{\rightsquigarrow}} \hat{l}_2 \wedge \hat{l}_2 \overset{*}{\widehat{\rightsquigarrow}} \hat{l}_1)$.

As the abstract interpreter performs abstract garbage collection through reference counting, we refer to it as *abstract reference counting.* We augment the abstract interpreter of Section 2.2 with abstract reference counting and show that, in the absence of cycles, it is equivalent to a garbage-free abstract interpreter using abstract tracing GC with the transition relation $(\widehat{\rightarrow}_{\widehat{\Gamma}})$. Section 4 discusses our solution to the problem of cyclic garbage.

3.1 Abstract Interpretation with Reference Counting using $(\widehat{\rightarrow}_{\mathsf{arc}})$

$$\widehat{\varsigma} \in \widehat{\Sigma}_{\mathsf{arc}} = \mathsf{Exp} \times \widehat{\mathsf{Env}} \times \widehat{\mathsf{Store}} \times \widehat{\mathsf{KStore}} \times \widehat{\mathsf{KAddr}} \times \widehat{\mathsf{Refs}} \qquad \widehat{\phi} \in \widehat{\mathsf{Refs}} = \widehat{\mathsf{Loc}} \to \mathcal{P}(\widehat{\mathsf{Loc}})$$

Figure 7 State space of the abstract interpreter with reference counting for λ_{ANF}. Other components preserve their definition given in Figure 5.

Figure 7 shows the updated abstract state space. Every state $\widehat{\varsigma}$ has been augmented with an additional component $\widehat{\phi}_{\widehat{\varsigma}}$ that keeps track of the references between addresses. Defined deterministically in terms of $\widehat{\varsigma}$'s original components, this addition does not increase the complexity of the state space explored by the abstract interpreter:

$$\widehat{\phi}_{\widehat{\varsigma}}(\widehat{l}) = \{\widehat{l}' \in \widehat{\mathsf{Loc}} \mid \widehat{l}' \mathrel{\widehat{\rightsquigarrow}_{\widehat{\varsigma}}} \widehat{l}\}$$

That is, $\widehat{\phi}(\widehat{l})$ contains all addresses that refer directly to $\widehat{l}$. The actual *reference count* of $\widehat{l}$ is obtained as $|\widehat{\phi}(\widehat{l})|$. Mapping the addresses in $\widehat{\phi}$ to a set of addresses ($\mathcal{P}(\widehat{\mathsf{Loc}})$) rather than an actual reference count $(\mathbb{N})^4$ renders our formalization more concise as well as amenable to the incorporation of cycle detection (cf. Section 4). We introduce the following definitions:

$$(\widehat{\phi}_1 \sqcup \widehat{\phi}_2)(\widehat{l}) = \widehat{\phi}_1(\widehat{l}) \cup \widehat{\phi}_2(\widehat{l}) \qquad\qquad \bot_{\widehat{\phi}} = \lambda \widehat{l}.\emptyset$$

We introduce the new transition relation $(\widehat{\rightarrow}_{\mathsf{arc}})$ as a composition of an auxiliary transition relation $(\widehat{\rightarrow}_0)$ and a function $\widehat{\mathsf{collect}}$. The auxiliary relation $(\widehat{\rightarrow}_0)$ extends the transition relation $(\widehat{\rightarrow})$ of Figure 6 by updating the references of addresses (in $\widehat{\phi}$) for every update to the store (or continuation store). The function $\widehat{\mathsf{collect}}$ is responsible for collecting garbage as reference counts become zero (i.e. $|\widehat{\phi}(\widehat{l})| = 0$ for some $\widehat{l}$) after every transition step of $(\widehat{\rightarrow}_0)$.

$$\frac{\widehat{\varsigma} \mathrel{\widehat{\rightarrow}_0} \widehat{\varsigma}' \qquad \widehat{\varsigma}'' = \widehat{\mathsf{collect}}(\widehat{\varsigma}, \widehat{\varsigma}')}{\widehat{\varsigma} \mathrel{\widehat{\rightarrow}_{\mathsf{arc}}} \widehat{\varsigma}''}$$

Figure 8 shows the definition of the auxiliary transition $(\widehat{\rightarrow}_0)$. Whenever the updated transition rules need to insert a value $\widehat{v}$ (or continuation $\widehat{\kappa}$) at address $\widehat{l}$, they now add $\widehat{l}$ to the set of references $\widehat{\phi}(\widehat{l}')$ for every address $\widehat{l}'$ that is directly reachable from the new value $\widehat{v}$ (or continuation $\widehat{\kappa}$) that $\widehat{l}$ is bound to. These addresses can be computed using $\widehat{\mathcal{T}}_{\mathcal{P}(\widehat{\mathsf{Clo}})}(\widehat{v})$ (or $\widehat{\mathcal{T}}_{\widehat{\mathsf{Kont}}}(\widehat{\kappa})$), where $\widehat{\mathcal{T}}$ was defined in Section 2.2. While $(\widehat{\rightarrow}_0)$ maintains $\widehat{\phi}$ as intended, it does not yet collect any garbage. Indeed, none of the transition rules in $(\widehat{\rightarrow}_0)$ remove any references to or from addresses, and the components $\widehat{\sigma}$, $\widehat{\sigma}_k$ and $\widehat{\phi}$ only grow monotonically.

[4] Note that it is not necessary to maintain a finite approximation such as $\widehat{\mathbb{N}} = \{0, 1, \infty\}$ for this reference count. The key insight is to count the "abstract" references to an abstract address, not the concrete references to the corresponding concrete address(es) that are approximated by that abstract address. That is, our approach performs abstract interpretation *with* reference counting, not an abstract interpretation *of* reference counting.

$$\frac{\widehat{a} = \widehat{\mathsf{alloc}}(x, \widehat{\varsigma}) \qquad \widehat{a}'_k = \widehat{\mathsf{alloc}}_k(e_1, \widehat{\varsigma}) \qquad \widehat{\rho}' = \widehat{\rho}[x \mapsto \widehat{a}] \\ \widehat{\kappa} = \langle \widehat{a}, e_2, \widehat{\rho}', \widehat{a}_k \rangle \qquad \widehat{\phi}' = \widehat{\phi} \sqcup \bigsqcup_{\widehat{l} \in \widehat{\mathcal{T}}_{\widehat{\mathsf{Kont}}}(\widehat{\kappa})} [\widehat{l} \mapsto \{\widehat{a}'_k\}]}{\underbrace{\langle \mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2, \widehat{\rho}, \widehat{\sigma}, \widehat{\sigma}_k, \widehat{a}_k, \widehat{\phi} \rangle}_{\hat{\varsigma}} \;\widehat{\rightarrow}_{\mathsf{0}}\; \langle e_1, \widehat{\rho}, \widehat{\sigma}, \widehat{\sigma}_k \sqcup [\widehat{a}'_k \mapsto \widehat{\kappa}], \widehat{a}'_k, \widehat{\phi}' \rangle} \quad (\text{E-Let})$$

$$\frac{\widehat{a} = \widehat{\mathsf{alloc}}(x, \widehat{\varsigma}) \qquad \widehat{\mathcal{A}}(f, \widehat{\rho}, \widehat{\sigma}) \ni \langle \lambda x.e', \widehat{\rho}' \rangle \\ \widehat{\mathcal{A}}(ae, \widehat{\rho}, \widehat{\sigma}) = \widehat{v} \qquad \widehat{\phi}' = \widehat{\phi} \sqcup \bigsqcup_{\widehat{l} \in \widehat{\mathcal{T}}_{\mathcal{P}(\widehat{\mathsf{Clo}})}(\widehat{v})} [\widehat{l} \mapsto \{\widehat{a}\}]}{\underbrace{\langle f\ ae, \widehat{\rho}, \widehat{\sigma}, \widehat{\sigma}_k, \widehat{a}_k, \widehat{\phi} \rangle}_{\hat{\varsigma}} \;\widehat{\rightarrow}_{\mathsf{0}}\; \langle e', \widehat{\rho}'[x \mapsto \widehat{a}], \widehat{\sigma} \sqcup [\widehat{a} \mapsto \widehat{v}], \widehat{\sigma}_k, \widehat{a}_k, \widehat{\phi}' \rangle} \quad (\text{E-Call})$$

$$\frac{\widehat{\mathcal{A}}(ae, \widehat{\rho}, \widehat{\sigma}) = \widehat{v} \qquad \widehat{\sigma}_k(\widehat{a}_k) \ni \langle \widehat{a}, e', \widehat{\rho}', \widehat{a}'_k \rangle \qquad \widehat{\phi}' = \widehat{\phi} \sqcup \bigsqcup_{\widehat{l} \in \widehat{\mathcal{T}}_{\mathcal{P}(\widehat{\mathsf{Clo}})}(\widehat{v})} [\widehat{l} \mapsto \{\widehat{a}\}]}{\langle ae, \widehat{\rho}, \widehat{\sigma}, \widehat{\sigma}_k, \widehat{a}_k, \widehat{\phi} \rangle \;\widehat{\rightarrow}_{\mathsf{0}}\; \langle e', \widehat{\rho}', \widehat{\sigma} \sqcup [\widehat{a} \mapsto \widehat{v}], \widehat{\sigma}_k, \widehat{a}'_k, \widehat{\phi}' \rangle} \quad (\text{E-Return})$$

Figure 8 Auxiliary transition relation ($\widehat{\rightarrow}_{\mathsf{0}}$) for abstract reference counting in λ_{ANF}.

Recall that garbage is defined as all addresses that are not reachable, i.e., all addresses that are not in $\widehat{\mathcal{R}}(\widehat{\varsigma}) = \{\widehat{l}' \in \widehat{\mathsf{Loc}} \mid \widehat{l} \in \widehat{\mathcal{T}}_{\widehat{\Sigma}}(\widehat{\varsigma}) \wedge \widehat{l} \overset{*}{\widehat{\leadsto}}_{\widehat{\varsigma}} \widehat{l}'\}$. We refer to $\widehat{\mathcal{T}}_{\widehat{\Sigma}}(\widehat{\varsigma})$ as the *root set* of addresses for a given state $\widehat{\varsigma}$. Whenever $\widehat{\varsigma}$ transitions to $\widehat{\varsigma}'$, the root set changes from $\widehat{\mathcal{T}}_{\widehat{\Sigma}}(\widehat{\varsigma})$ to $\widehat{\mathcal{T}}_{\widehat{\Sigma}}(\widehat{\varsigma}')$, which creates garbage when an address $\widehat{l}$ that was in $\widehat{\mathcal{T}}_{\widehat{\Sigma}}(\widehat{\varsigma})$ is no longer in $\widehat{\mathcal{T}}_{\widehat{\Sigma}}(\widehat{\varsigma}')$ *and* $\widehat{l}$ is not referenced from any other address (i.e., $|\widehat{\phi}_{\widehat{\varsigma}'}(\widehat{l})| = 0$). When this happens, the abstract interpreter needs to garbage collect $\widehat{l}$ by removing it from the store. Garbage collecting an address $\widehat{l}$ also removes $\widehat{l}$ from $\widehat{\phi}_{\widehat{\varsigma}'}(\widehat{l}')$ for every $\widehat{l}' \in \widehat{\mathsf{Loc}}$ where $\widehat{l} \,\widehat{\leadsto}_{\widehat{\varsigma}'}\, \widehat{l}'$. This is akin to decrementing the reference count of all addresses $\widehat{l}'$ referenced by $\widehat{l}$. The update can cause other addresses to be garbage collected because they no longer have any references. Note that $\widehat{\phi}$ only takes into account references coming from other addresses; we take care of root references using the function $\widehat{\mathsf{collect}} : \widehat{\Sigma}_{\mathsf{arc}} \times \widehat{\Sigma}_{\mathsf{arc}} \rightarrow \widehat{\Sigma}_{\mathsf{arc}}$, which removes all garbage from $\widehat{\varsigma}'$ that is created due to a change in the root set (from $\widehat{\mathcal{T}}_{\widehat{\Sigma}}(\widehat{\varsigma})$ to $\widehat{\mathcal{T}}_{\widehat{\Sigma}}(\widehat{\varsigma}')$) as $\widehat{\varsigma} \,\widehat{\rightarrow}_{\mathsf{0}}\, \widehat{\varsigma}'$.

$$\widehat{\mathsf{collect}}(\widehat{\varsigma}, \widehat{\varsigma}') = \langle e_{\hat{\varsigma}'}, \widehat{\rho}_{\hat{\varsigma}'}, \widehat{\sigma}_{\hat{\varsigma}'} \setminus \widehat{G},\ \widehat{\sigma_k}_{\hat{\varsigma}'} \setminus \widehat{G},\ \widehat{a_k}_{\hat{\varsigma}'}, \widehat{\phi}' \rangle \text{where } \langle \widehat{\phi}', \widehat{G} \rangle = \widehat{\mathsf{check*}}(\widehat{\mathcal{T}}_{\widehat{\Sigma}}(\widehat{\varsigma}), \emptyset, \widehat{\phi}_{\hat{\varsigma}'})$$

$$\widehat{\mathsf{check*}}(\widehat{C}, \widehat{G}, \widehat{\phi}) = \begin{cases} \langle \widehat{\phi}, \widehat{G} \rangle & \text{if } \widehat{C} = \emptyset \\ \widehat{\mathsf{check}}(\widehat{l}, \widehat{C} \setminus \{\widehat{l}\}, \widehat{G}, \widehat{\phi}) & \text{otherwise, for any } \widehat{l} \in \widehat{C} \end{cases}$$

$$\widehat{\mathsf{check}}(\widehat{l}, \widehat{C}, \widehat{G}, \widehat{\phi}) = \begin{cases} \widehat{\mathsf{dealloc}}(\widehat{l}, \widehat{C}, \widehat{G}, \widehat{\phi}) & \text{if } |\widehat{\phi}(\widehat{l})| = 0 \wedge \widehat{l} \notin \widehat{\mathcal{T}}_{\widehat{\Sigma}}(\widehat{\varsigma}') \\ \widehat{\mathsf{check*}}(\widehat{C}, \widehat{G}, \widehat{\phi}) & \text{otherwise} \end{cases}$$

$$\widehat{\mathsf{dealloc}}(\widehat{l}, \widehat{C}, \widehat{G}, \widehat{\phi}) = \widehat{\mathsf{check*}}(\widehat{C} \cup \widehat{S}, \widehat{G} \cup \{\widehat{l}\}, \widehat{\phi} \ominus \bigsqcup_{\widehat{l}' \in \widehat{S}} [\widehat{l}' \mapsto \{\widehat{l}\}]) \quad \text{where } \widehat{S} = \{\widehat{l}' \mid \widehat{l} \,\widehat{\leadsto}_{\hat{\varsigma}'}\, \widehat{l}'\}$$

We use the notation for set removal to remove elements from a map m, so that $(m \setminus S)(x) = \emptyset$ if $x \in S$ and $(m \setminus S)(x) = m(x)$ if $x \notin S$. We define $(\widehat{\phi}_1 \ominus \widehat{\phi}_2)(\widehat{l}) = \widehat{\phi}_1(\widehat{l}) \setminus \widehat{\phi}_2(\widehat{l})$ to conveniently update $\widehat{\phi}$ when references are removed (due to addresses being garbage collected).

The function $\widehat{\mathsf{collect}}$ checks for every address $\widehat{l} \in \widehat{\mathcal{T}}_{\widehat{\Sigma}}(\widehat{\varsigma})$ whether it is still referenced by at least one other address ($|\widehat{\phi}_{\widehat{\varsigma}'}(\widehat{l})| > 0$) or whether it is still part of the root set ($\widehat{l} \in \widehat{\mathcal{T}}_{\widehat{\Sigma}}(\widehat{\varsigma}')$). If this is not the case, then it should garbage collect $\widehat{l}$ and update $\widehat{\phi}$ to remove $\widehat{l}$ from $\widehat{\phi}(\widehat{l}')$ for every $\widehat{l}'$ where $\widehat{l} \mathrel{\widehat{\rightsquigarrow}_{\widehat{\varsigma}'}} \widehat{l}'$, and recursively apply the same check for every such $\widehat{l}'$. Finally, $\widehat{\mathsf{collect}}(\widehat{\varsigma}, \widehat{\varsigma}')$ returns a new state $\widehat{\varsigma}''$ by removing all garbage from $\widehat{\sigma}_{\widehat{\varsigma}'}$ and $\widehat{\sigma_k}_{\widehat{\varsigma}'}$ and by updating $\widehat{\phi}_{\widehat{\varsigma}'}$ to take into account the removed references.

One can verify from the definitions of $(\widehat{\rightarrow}_0)$ and $\widehat{\mathsf{collect}}$ that the $\widehat{\phi}$-component is always updated correctly. That is, for every state $\widehat{\varsigma} \in \widehat{\Sigma}_{\mathsf{arc}}$, $(\widehat{\rightarrow}_{\mathsf{arc}})$, they maintain the invariant $\widehat{\phi}_{\widehat{\varsigma}}(\widehat{l}) = \{\widehat{l}' \mid \widehat{l}' \mathrel{\widehat{\rightsquigarrow}_{\widehat{\varsigma}}} \widehat{l}\}$, so that we can reason about $\widehat{\phi}(\widehat{l})$ as the set of addresses that refer to $\widehat{l}$.

3.2 Properties of $(\widehat{\rightarrow}_{\mathsf{arc}})$

We now show that – in the absence of cycles – $(\widehat{\rightarrow}_{\mathsf{arc}})$ is garbage-free, which means that abstract reference counting results in a garbage-free abstract interpreter.

First, we show that the garbage collection is *sound*. In terms of abstract garbage collection, soundness means that addresses that are reachable are not removed from the store [42]. Using the definitions of Section 2.2, this means that for every state $\widehat{\varsigma}$ it collects *only* unreachable addresses $\widehat{l} \notin \widehat{\mathcal{R}}(\widehat{\varsigma})$, so that $\widehat{\mathcal{R}}(\widehat{\varsigma}) \subseteq \widehat{\mathcal{S}}(\widehat{\varsigma})$. Intuitively, this implies that the garbage collection is safe, and never removes a binding that is still needed.

▶ **Definition 4** (GC Soundness). *A state $\widehat{\varsigma}$ is sound iff $\widehat{\mathcal{R}}(\widehat{\varsigma}) \subseteq \widehat{\mathcal{S}}(\widehat{\varsigma})$. A transition relation $(\widehat{\rightarrow})$ is sound iff it preserves soundness, i.e., if $\widehat{\varsigma}$ is sound and $\widehat{\varsigma} \mathrel{\widehat{\rightarrow}} \widehat{\varsigma}'$, then $\widehat{\varsigma}'$ is sound.*

▶ **Lemma 5.** *$(\widehat{\rightarrow}_{\mathsf{arc}})$ is sound.*

Proof. The proof is detailed in Appendix A. ◀

We refer to the dual of sound garbage collection as *complete* garbage collection. In the context of abstract GC, completeness means that all addresses that are in the store are still reachable. For every state $\widehat{\varsigma}$, a complete garbage collector collects *all* addresses $\widehat{l} \notin \widehat{\mathcal{R}}(\widehat{\varsigma})$, so that $\widehat{\mathcal{S}}(\widehat{\varsigma}) \subseteq \widehat{\mathcal{R}}(\widehat{\varsigma})$. Intuitively, this implies that the garbage collection never misses any garbage, only keeping non-garbage values in the store at all times. We show that $(\widehat{\rightarrow}_{\mathsf{arc}})$ is GC complete in the absence of cycles.

▶ **Definition 6** (GC Completeness). *A state $\widehat{\varsigma}$ is complete iff $\widehat{\mathcal{S}}(\widehat{\varsigma}) \subseteq \widehat{\mathcal{R}}(\widehat{\varsigma})$. A relation $(\widehat{\rightarrow})$ is complete iff it preserves completeness, i.e. if $\widehat{\varsigma}$ is complete and $\widehat{\varsigma} \mathrel{\widehat{\rightarrow}} \widehat{\varsigma}'$, then $\widehat{\varsigma}'$ is complete.*

▶ **Lemma 7.** *In the absence of cycles, $(\widehat{\rightarrow}_{\mathsf{arc}})$ is complete.*

Proof. The proof is detailed in Appendix A. ◀

Note that combining GC soundness with GC completeness yields the garbage-free property. That is, if a state $\widehat{\varsigma}$ is both sound and complete, we have that $\widehat{\mathcal{R}}(\widehat{\varsigma}) \subseteq \widehat{\mathcal{S}}(\widehat{\varsigma})$ and $\widehat{\mathcal{S}}(\widehat{\varsigma}) \subseteq \widehat{\mathcal{R}}(\widehat{\varsigma})$, hence $\widehat{\mathcal{R}}(\widehat{\varsigma}) = \widehat{\mathcal{S}}(\widehat{\varsigma})$ so that $\widehat{\varsigma}$ by definition is garbage-free. An abstract interpreter that applies abstract tracing GC (as in Section 2.2), but not at every step, is GC sound, but not GC complete, and therefore not garbage-free. Similarly, abstract reference counting in the presence of cycles is still GC sound (Lemma 5), but not GC complete, nor garbage-free.

▶ **Theorem 8.** *In the absence of cycles, $(\widehat{\rightarrow}_{\mathsf{arc}})$ is garbage-free.*

Proof. Follows immediately from Lemma 5 and Lemma 7. ◀

We have now shown that – in the absence of cycles – using $(\widehat{\rightarrow}_{\mathsf{arc}})$ as a transition relation results in a garbage-free abstract interpreter without the need to trigger a full tracing GC at every step. Normally, we would still have to prove other properties of $(\widehat{\rightarrow}_{\mathsf{arc}})$ (such as the soundness of the abstract interpretation); instead, we prove an equivalence with the existing transition relation $(\widehat{\rightarrow}_{\widehat{\Gamma}})$, for which such properties have already been established in related work [23, 42]. First, we introduce a function $\widehat{\mathsf{rc}} : \widehat{\Sigma} \rightarrow \widehat{\Sigma}_{\mathsf{arc}}$ to map states from the abstract interpreter in Section 2.2 to equivalent states in $\widehat{\Sigma}_{\mathsf{arc}}$ with the following definition:

$$\widehat{\mathsf{rc}}(\widehat{\varsigma}) = \langle e_{\hat{\varsigma}}, \widehat{\rho}_{\hat{\varsigma}}, \widehat{\sigma}_{\hat{\varsigma}}, \widehat{\sigma_k}_{\hat{\varsigma}}, \widehat{a}_k, \widehat{\phi} \rangle \qquad \text{where } \widehat{\phi}(\widehat{l}) = \{\widehat{l'} \in \widehat{\mathsf{Loc}} \mid \widehat{l'} \mathrel{\widehat{\rightsquigarrow}_{\hat{\varsigma}}} \widehat{l}\}$$

▶ **Theorem 9.** *In the absence of cycles, the transition relation* $(\widehat{\rightarrow}_{\mathsf{arc}})$ *is equivalent to* $(\widehat{\rightarrow}_{\widehat{\Gamma}})$ *in the sense that:* $\forall \widehat{\varsigma}, \widehat{\varsigma}' \in \widehat{\Sigma},\ \widehat{\varsigma} \mathrel{\widehat{\rightarrow}_{\widehat{\Gamma}}} \widehat{\varsigma}' \iff \widehat{\mathsf{rc}}(\widehat{\varsigma}) \mathrel{\widehat{\rightarrow}_{\mathsf{arc}}} \widehat{\mathsf{rc}}(\widehat{\varsigma}')$.

Proof. The proof is detailed in Appendix A. ◀

As a consequence, we can design a garbage-free abstract interpreter using $(\widehat{\rightarrow}_{\mathsf{arc}})$ by defining

$$\widehat{\mathsf{eval}}_{\mathsf{arc}}(e) = \{\widehat{\varsigma} \in \widehat{\Sigma}_{\mathsf{arc}} \mid \langle e, [], \perp_{\widehat{\sigma}}, \perp_{\widehat{\sigma}_k}, \widehat{a}_{\mathsf{halt}}, \perp_{\widehat{\phi}} \rangle \overset{*}{\widehat{\rightarrow}}_{\mathsf{arc}} \widehat{\varsigma}\}$$

so that for every program e that, in the absence of cycles, $\widehat{\mathsf{eval}}_{\mathsf{arc}}(e) = \left\{\widehat{\mathsf{rc}}(\widehat{\varsigma}) \mid \widehat{\varsigma} \in \widehat{\mathsf{eval}}(e)\right\}$.

4 Reclaiming Garbage Cycles

In the previous section, we showed that the $(\widehat{\rightarrow}_{\mathsf{arc}})$ transition relation is garbage-free under the premise that no states are encountered with cycles in their store or continuation store. The underlying reason for this limitation is that pure reference counting cannot reclaim cycles [50]. However, it is clear that this premise does not hold in general. A concrete interpreter can end up with cycles in the store σ when a program creates cyclic data structures. In this case, abstract interpretation of this program will also result in a cyclic structure in $\widehat{\sigma}$, which will never be reclaimed using $(\widehat{\rightarrow}_{\mathsf{arc}})$. Hence, in general the transition relation $(\widehat{\rightarrow}_{\mathsf{arc}})$ is only GC sound, not GC complete and therefore also not garbage-free.

4.1 Artificial Cycles in $(\widehat{\rightarrow}_{\mathsf{arc}})$

To make matters worse, we can show that cycles are much more common in the abstract. That is, for a given program e, an abstract interpreter may encounter states with cycles in either $\widehat{\sigma}$ (or $\widehat{\sigma}_k$) that would not occur in σ (or σ_k) in states explored by a concrete interpreter. The underlying reason is that an abstract interpreter can only use a finite number of addresses, and therefore is sometimes forced to reuse an address for different allocations.

From a theoretical perspective, this implies that an abstract address ($\in \widehat{\mathsf{Loc}}$) is used as the abstraction for multiple concrete addresses ($\in \mathsf{Loc}$). We assume that abstraction function $\alpha : \mathsf{Loc} \rightarrow \widehat{\mathsf{Loc}}$ maps a concrete address l to the abstract address $\widehat{l}$, while concretization function $\gamma : \widehat{\mathsf{Loc}} \rightarrow \mathcal{P}(\mathsf{Loc})$ returns all concrete addresses that are abstracted to an abstract address. It can be shown from the abstract semantics [23] that if l_1 references l_2 ($l_1 \rightsquigarrow l_2$) under concrete interpretation, then $\alpha(l_1)$ references $\alpha(l_2)$ under abstract interpretation too ($\alpha(l_1) \mathrel{\widehat{\rightsquigarrow}} \alpha(l_2)$). Now consider the linear sequence of concrete references $l_0 \rightsquigarrow l_1 \rightsquigarrow \ldots \rightsquigarrow l_k$ with $l_0 \neq l_1 \neq \ldots \neq l_k$. Its abstraction $\alpha(l_0) \mathrel{\widehat{\rightsquigarrow}} \alpha(l_1) \mathrel{\widehat{\rightsquigarrow}} \ldots \mathrel{\widehat{\rightsquigarrow}} \alpha(l_k)$ becomes cyclic as soon as for some $i \neq j$, $\alpha(l_i) = \alpha(l_j)$. We refer to the cycles that lack a concrete counterpart and stem from the interpreter's abstraction as *artificial cycles.*

In practice, it turns out that artificial cycles are mostly a problem in the continuation store. Consider the E-Let transition rule. This transition rule "pushes" a continuation on the continuation store, so that if $\varsigma \to \varsigma'$, we have that $a_{k_{\varsigma'}} \rightsquigarrow a_{k_{\varsigma}}$. As kalloc always returns a fresh address under concrete interpretation, a sequence of transitions of this rule will give rise to $a_{kn} \rightsquigarrow a_{kn-1} \rightsquigarrow \ldots \rightsquigarrow a_{k0}$ where $a_{kn} \neq a_{kn-1} \neq \ldots \neq a_{k0}$. Under abstract interpretation, in contrast, the same sequence will result in $\alpha(a_{kn}) \mathrel{\widehat{\rightsquigarrow}} \alpha(a_{kn-1}) \mathrel{\widehat{\rightsquigarrow}} \ldots \mathrel{\widehat{\rightsquigarrow}} \alpha(a_{k0})$. A cycle will be created as soon as $\widehat{\mathsf{kalloc}}$ allocates the same address more than once in such a sequence (i.e., once $\alpha(a_{ki}) = \alpha(a_{kj})$ for some $0 \leq i < j \leq n$). This is usually the case when the abstract interpreter visits the same program location multiple times due to, for instance, looping behavior. When the resulting cycle becomes unreachable, none of its constituent addresses will be reclaimed under $(\widehat{\to}_{\mathsf{arc}})$ from the continuation store where they cause imprecisions. While imprecision in the value store $\widehat{\sigma}$ may be tolerable, imprecision in the continuation store $\widehat{\sigma}_k$ has a detrimental impact. The E-Return transition rule, for instance, will generate *spurious* successor states if set $\widehat{\sigma}_k(\widehat{a}_k)$ contains "garbage continuations" due to such imprecision resulting in the exploration of infeasible control flow with an impact on both precision and performance.

4.2 Abstract Reference Counting with Cycle Detection: $(\widehat{\to}_{\mathsf{arc++}})$

We extend $(\widehat{\to}_{\mathsf{arc}})$ so that it remains GC complete in the presence of cycles. Existing techniques for reference counting to reclaim cycles do not immediately help in this setting (cf. Section 6). We therefore propose a domain-specific solution that is tailored towards the artificial cycles that are artefacts of abstraction. First, we make a few observations:

- As discussed above, cycles are the most prevalent in the continuation store. Not reclaiming such cycles impacts precision and performance negatively. Our main priority is therefore the continuation store. While realizing the garbage-free property requires cycle detection in both stores, we show that its application to the continuation store does not have a significant performance overhead (cf. Theorem 10).
- An artificial cycle can only be created when an address is reused, and address reuse often results in a cycle in practice. In a garbage-free abstract interpreter, reusing an address in the continuation store is guaranteed to create a cycle.
- The transition rules in $(\widehat{\to}_0)$ only make the stores grow monotonically; elements are only removed from the store when addresses are garbage collected. As a consequence, cycles never "break up", because if some address that is part of a cycle is collected as garbage, then by the definition of garbage the entire cycle should be collected.

Hence, while it is not obvious how to detect when a cycle becomes a garbage cycle, it is possible to predict the creation of a cycle efficiently. Therefore, our approach explicitly tracks all cycles – or rather *strongly connected components* (SCC) – in $\widehat{\mathcal{S}}(\widehat{\varsigma})$. Such cycles can only grow when two SCCs are merged together, or be removed in their entirety when collected as garbage, enabling maintaining a disjoint-set of the SCCs in $\widehat{\mathcal{S}}(\widehat{\varsigma})$ efficiently. The idea is then to maintain the reference count for every SCC, only counting "external" references coming from other SCCs. When a SCC loses all of its external references, all addresses that are part of that SCC can be garbage collected. As a cycle can by definition never occur between two SCCs, applying reference counting at this level overcomes its main limitation.

Figure 9 depicts the updates to the state space $\widehat{\Sigma}_{\mathsf{arc}}$ that incorporate cycle detection. Newly-added component $\widehat{\pi}$ tracks the partitioning of $\widehat{\mathcal{S}}(\widehat{\varsigma})$ into its SCCs.

$$\begin{aligned} \hat{\varsigma} \in \widehat{\Sigma}_{\mathsf{arc+}} = {} & \mathsf{Exp} \times \widehat{\mathsf{Env}} \times \widehat{\mathsf{Store}} \\ & \times \widehat{\mathsf{KStore}} \times \widehat{\mathsf{KAddr}} \\ & \times \widehat{\mathsf{Refs}} \times \widehat{\mathsf{DS}} \end{aligned} \qquad \begin{aligned} \hat{s}, \widehat{scc} \in \widehat{\mathsf{SCC}} &= \mathcal{P}(\widehat{\mathsf{Loc}}) \\ \hat{\phi} \in \widehat{\mathsf{Refs}} &= \widehat{\mathsf{SCC}} \rightharpoonup \mathcal{P}(\widehat{\mathsf{Loc}}) \\ \hat{\pi} \in \widehat{\mathsf{DS}} &= \mathcal{P}(\widehat{\mathsf{SCC}}) \end{aligned}$$

Figure 9 Updated state space of the abstract interpreter with reference counting to detect cycles. Other components preserve the original definition from Figure 7.

The abstract interpreter needs to maintain disjoint-set $\hat{\pi}$ so that $\bigcup_{\widehat{scc} \in \hat{\pi}} \widehat{scc} = \widehat{\mathsf{Loc}}$ and for any $\hat{s}_1, \hat{s}_2 \in \hat{\pi}$ where $\hat{s}_1 \neq \hat{s}_2$, $\hat{s}_1 \cap \hat{s}_2 = \emptyset$. In what follows, we assume that functions union and find (for which efficient implementations have been proposed [19]) exist, so that:

$$\mathsf{union}(\hat{\pi}, \hat{s}_1, \hat{s}_2) = (\hat{\pi} \setminus \{\hat{s}_1, \hat{s}_2\}) \cup \{\hat{s}_1 \cup \hat{s}_2\} \qquad \mathsf{find}(\hat{\pi}, \hat{l}) = \widehat{scc} \text{ where } \widehat{scc} \in \hat{\pi} \wedge \hat{l} \in \widehat{scc}$$

The original definition of $\hat{\phi}$ from Section 3.1 also changes in that it now tracks all incoming references *for every strongly connected component* from the other SCCs. Note again that the state space does not increase in complexity, since both $\hat{\phi}_{\hat{\varsigma}}$ and $\hat{\pi}_{\hat{\varsigma}}$ are defined deterministically in terms of $\hat{\varsigma}$'s other components:

$$\hat{\phi}_{\hat{\varsigma}}(\widehat{scc}) = \{\hat{l}' \in \widehat{\mathsf{Loc}} \mid \hat{l}' \notin \widehat{scc} \wedge \exists \hat{l} \in \widehat{scc}, \hat{l}' \mathrel{\widehat{\rightsquigarrow}_{\hat{\varsigma}}} \hat{l}\} \text{ for } \widehat{scc} \in \hat{\pi}_{\hat{\varsigma}} \qquad \bot_{\hat{\phi}} = \lambda \widehat{scc}.\, \emptyset$$

$$\forall \hat{l}_1, \hat{l}_2 \in \widehat{\mathsf{Loc}},\ \mathsf{find}(\hat{\pi}_{\hat{\varsigma}}, \hat{l}_1) = \mathsf{find}(\hat{\pi}_{\hat{\varsigma}}, \hat{l}_2) \iff (\hat{l}_1 \mathrel{\overset{*}{\widehat{\rightsquigarrow}}_{\hat{\varsigma}}} \hat{l}_2 \wedge \hat{l}_2 \mathrel{\overset{*}{\widehat{\rightsquigarrow}}_{\hat{\varsigma}}} \hat{l}_1) \qquad \bot_{\hat{\pi}} = \{\{\hat{l}\} \mid \hat{l} \in \widehat{\mathsf{Loc}}\}$$

Transition relation $(\widehat{\rightarrow}_0)$ of Figure 8 needs to be updated to maintain the $\hat{\pi}$ and $\hat{\phi}$ components. Previously in $(\widehat{\rightarrow}_0)$, whenever a reference from address $\hat{l}_{from}$ to a set of addresses $\hat{S}$ was added to the store, $\hat{\phi}$ was updated to $\hat{\phi} \sqcup \bigsqcup_{\hat{l}_{to} \in \hat{S}} [\hat{l}_{to} \mapsto \{\hat{l}_{from}\}]$. For the new auxiliary transition relation $(\widehat{\rightarrow}_0)$, we replace this update with a call to the function $\textsc{extend}(\hat{l}_{from}, \hat{S}, \hat{\phi}, \hat{\pi})$. The full definition of $(\widehat{\rightarrow}_{0++})$ can be found in Appendix B.

The $\textsc{extend}$ function in Algorihm 1 checks if SCCs are merged together when either the store or continuation store is extended. For every new reference from $\hat{l}_{from}$ to $\hat{l}_{to}$ that is added to the store, it calls the function $\textsc{update}$ to detect if a new cycle has been created and to update $\hat{\phi}$ and $\hat{\pi}$ accordingly. The $\textsc{update}$ function initiates a backward search for the SCC of $\hat{l}_{to}$, starting from the SCC of $\hat{l}_{from}$. All strongly connected components that can be traversed in a path from $\hat{l}_{from}$ to $\hat{l}_{to}$ need to be merged together in $\hat{\pi}$ (using union), since such a path implies that a cycle is created as $\hat{l}_{to} \mathrel{\overset{*}{\widehat{\rightsquigarrow}}} \hat{l}_{from}$ *and* $\hat{l}_{from} \mathrel{\widehat{\rightsquigarrow}} \hat{l}_{to}$ (due to the new reference from $\hat{l}_{from}$ to $\hat{l}_{to}$ that was added to the store). During this traversal, $\textsc{update}$ also keeps track of all incoming references to this newly created SCC, so that it can immediately update $\hat{\phi}$ as well. If no such path is found, $\hat{l}_{from}$ and $\hat{l}_{to}$ are not part of the same SCC, so that $\hat{\pi}$ remains unchanged and $\hat{\phi}$ is updated as in $(\widehat{\rightarrow}_0)$. Note that the check $\widehat{scc} \in V$ is only there to prevent redundant work when a SCC has already been visited through a different path; its purpose is *not* to prevent an infinite looping of $\textsc{search}$, which can not happen as there are no cyclic paths between SCCs.

Of course, performing cycle detection through a search comes at a cost, especially when looking at the theoretical worst-case behaviour. In practice, however, the advantages of being able to reclaim garbage cycles appear to outweigh this cost (cf. Section 5.2). Note that the backward search is only initiated when an address is being reused, as for a fresh address $\hat{l}$, there are no incoming references to traverse (i.e., $\hat{\phi}(\mathsf{find}(\hat{\pi}, \hat{l})) = \emptyset$). As address reuse often creates a cycle, incorporating cycle detection usually pays off. In fact, for the continuation store, we can state this more formally in Theorem 10.

Algorithm 1: Cycle detection in $(\widehat{\rightarrow}_{0++})$.

```
function UPDATE($\widehat{l}_{from}$,$\widehat{l}_{to}$,$\widehat{\phi}$,$\widehat{\pi}$):
    $V \leftarrow \emptyset$ ; $I \leftarrow \widehat{\phi}(\mathsf{find}(\widehat{\pi}, \widehat{l}_{to}))$
    function SEARCH($\widehat{l}$):
        $\widehat{scc} \leftarrow \mathsf{find}(\widehat{\pi}, \widehat{l})$
        if $\widehat{scc} = \mathit{find}(\widehat{\pi}, \widehat{l}_{to})$ then return true
        if $\widehat{scc} \in V$ then return false
        $V \leftarrow V \cup \{\widehat{scc}\}$
        $reachable \leftarrow$ false
        $incoming \leftarrow \emptyset$
        foreach $\widehat{l'} \in \widehat{\phi}(\widehat{scc})$ do
            if SEARCH($\widehat{l'}$) then $reachable \leftarrow$ true
            else $incoming \leftarrow incoming \cup \{\widehat{l'}\}$
        end
        if $reachable$ then
            $\widehat{\pi} \leftarrow \mathsf{union}(\widehat{scc}, \mathsf{find}(\widehat{\pi}, \widehat{l}_{to}))$
            $I \leftarrow I \cup incoming$
        end
        return $reachable$
    if SEARCH($\widehat{l}_{from}$) then
        return $\langle \widehat{\phi}|_{\widehat{\pi}} \sqcup [\mathsf{find}(\widehat{\pi}, \widehat{l}_{to}) \mapsto I], \widehat{\pi} \rangle$
    else
        return $\langle \widehat{\phi} \sqcup [\mathsf{find}(\widehat{\pi}, \widehat{l}_{to}) \mapsto \{\widehat{l}_{from}\}], \widehat{\pi} \rangle$
function EXTEND($\widehat{l}_{from}$,$\widehat{S}$,$\widehat{\phi}$,$\widehat{\pi}$):
    foreach $\widehat{l}_{to} \in \widehat{S}$ do
        if $\widehat{l}_{from} \notin \widehat{\phi}(\mathit{find}(\widehat{\pi}, \widehat{l}_{to}))$ then
            $\langle \widehat{\phi}, \widehat{\pi} \rangle \leftarrow$UPDATE($\widehat{l}_{from}$,$\widehat{l}_{to}$,$\widehat{\phi}$,$\widehat{\pi}$)
    end
    return $\langle \widehat{\phi}, \widehat{\pi} \rangle$
```

▶ **Theorem 10.** *In a garbage-free abstract interpreter, extending abstract reference counting with cycle detection for the continuation store only requires amortized $\mathcal{O}(1)$ additional operations per continuation that is added to the continuation store.*

Proof. The proof is detailed in Appendix A. ◀

The intuition here is that every reference that is inserted into $\widehat{\sigma_k}$ can only be traversed once during the backward search of the cycle detection: once traversed, we can show that it is guaranteed to become part of a cycle (cf. proof of Theorem 10 in Appendix A). Since the internal references of a cycle are not traversed by the backward search, it can no longer add to the cost of a future traversal once the SCCs are merged. That is, when a backward search is triggered in the continuation store, we are guaranteed that all traversed edges between SCCs will become part of the same SCC after cycle detection, hence the path is shortened for future traversals. In practical terms, Theorem 10 states that cycle detection can be applied to the continuation store "for free", i.e., without much of an additional cost in performance.

Finally, we can define $(\widehat{\rightarrow}_{\mathsf{arc}++})$ as we did previously for $(\widehat{\rightarrow}_{\mathsf{arc}})$:

$$\frac{\widehat{\varsigma} \; \widehat{\rightarrow}_{0++} \; \widehat{\varsigma}' \qquad \widehat{\varsigma}'' = \widehat{\mathsf{collect}}(\widehat{\varsigma}, \widehat{\varsigma}')}{\widehat{\varsigma} \; \widehat{\rightarrow}_{\mathsf{arc}++} \; \widehat{\varsigma}''}$$

where $\widehat{\mathsf{collect}}$ remains mostly unchanged, save for some trivial modifications to work at the level of SCCs. We refer to Appendix B for the updated definition of $\widehat{\mathsf{collect}}$.

Unlike $(\widehat{\rightarrow}_{\mathsf{arc}})$, we can show that using the transition relation $(\widehat{\rightarrow}_{\mathsf{arc}++})$ results in a garbage-free abstract interpreter, even in the presence of cycles. This is stated by Theorem 11.

▶ **Theorem 11.** $(\widehat{\rightarrow}_{\mathsf{arc}++})$ *is garbage-free.*

Proof. The proofs of Lemma 5 and 7 can be repeated for $(\widehat{\rightarrow}_{\mathsf{arc}++})$, replacing addresses with their strongly connected components where necessary. The premise of Lemma 7 that disallows cycles can then be omitted, since cycles by definition do not occur between SCCs. ◀

Moreover, we can now claim full equivalence of $(\widehat{\rightarrow}_{\widehat{\Gamma}})$ and $(\widehat{\rightarrow}_{\mathsf{arc}++})$. First, define $\widehat{\mathsf{rc}}$ as:

$$\widehat{\mathsf{rc}}(\widehat{\varsigma}) = \langle e_{\hat{\varsigma}}, \widehat{\rho}_{\hat{\varsigma}}, \widehat{\sigma}_{\hat{\varsigma}}, \widehat{\sigma_{k}}_{\hat{\varsigma}}, \widehat{a}_k, \widehat{\phi}, \widehat{\pi} \rangle \text{ where}$$

$$\widehat{\phi}(\widehat{scc}) = \{\widehat{l'} \in \widehat{\mathsf{Loc}} \mid \widehat{l'} \notin \widehat{scc} \land \exists \widehat{l} \in \widehat{scc}, \widehat{l'} \mathrel{\widehat{\rightsquigarrow}_{\hat{\varsigma}}} \widehat{l}\} \text{ for } \widehat{scc} \in \widehat{\pi}$$

$$\forall \widehat{l}_1, \widehat{l}_2 \in \widehat{\mathsf{Loc}},\ \mathsf{find}(\widehat{\pi}, \widehat{l}_1) = \mathsf{find}(\widehat{\pi}, \widehat{l}_2) \iff (\widehat{l}_1 \mathrel{\overset{*}{\widehat{\rightsquigarrow}}_{\hat{\varsigma}}} \widehat{l}_2 \land \widehat{l}_2 \mathrel{\overset{*}{\widehat{\rightsquigarrow}}_{\hat{\varsigma}}} \widehat{l}_1)$$

▶ **Theorem 12.** $(\widehat{\rightarrow}_{\mathsf{arc}})$ *is equivalent to* $(\widehat{\rightarrow}_{\widehat{\Gamma}}) : \forall \widehat{\varsigma}, \widehat{\varsigma}' \in \widehat{\Sigma}, \widehat{\varsigma} \mathrel{\widehat{\rightarrow}_{\widehat{\Gamma}}} \widehat{\varsigma}' \iff \widehat{\mathsf{rc}}(\widehat{\varsigma}) \mathrel{\widehat{\rightarrow}_{\mathsf{arc}++}} \widehat{\mathsf{rc}}(\widehat{\varsigma}')$.

Proof. The proof is analogous to that of Theorem 9. ◀

5 Evaluation

We have already proven the GC soundness and GC completeness of our approach (Lemmas 5 and 7), as well as a theoretical efficiency claim of the cycle detection technique for the continuation store (Theorem 10). We now present the empirical evaluation. Section 5.2 measures the impact of the cycle detection technique on abstract reference counting, while Section 5.3 compares its precision and performance to existing approaches to abstract GC.

5.1 Experimental Setup

We ran all experiments using Scala version 2.12.3 on a server with a 3.5GHz Intel Xeon 2637 processor and 256GB of RAM. We use a similar setup throughout our evaluation.

Implementation. We implemented both abstract reference counting and the existing variants of abstract tracing GC (surveyed in Section 1.2) using the Scala-AM framework for abstract interpretation of Scheme programs [54, 53]. This resulted in 5 abstract interpreters[5], each incorporating the Scheme equivalent of the λ_{ANF} transition relations $(\widehat{\rightarrow})$ and $(\widehat{\rightarrow}_{\widehat{\Gamma}})$ (cf. Section 2), $(\widehat{\rightarrow}_{\mathsf{arc}})$ (cf. Section 3) and $(\widehat{\rightarrow}_{\mathsf{arc}++})$ (cf. Section 4), as well as the additional $(\widehat{\rightarrow}_{\Gamma\mathsf{CFA}})$ and $(\widehat{\rightarrow}_{\mathsf{arc}+})$ relations defined below:

- $(\widehat{\rightarrow}_{\Gamma\mathsf{CFA}})$ adopts the GC policy of ΓCFA [42], which applies abstract GC whenever values need to be joined in the store, so that values are by design never merged with garbage. While this approach is not garbage-free (cf. Section 1.2), it is interesting to examine how this common policy affects the precision and performance of the abstract interpreter.
- $(\widehat{\rightarrow}_{\mathsf{arc}+})$ adopts abstract reference counting, but only applies cycle detection to the continuation store $\widehat{\sigma}_k$. As garbage cycles will remain in value store $\widehat{\sigma}$, this relation is not garbage-free. However, its empirical evaluation is motivated by our observation that garbage cycles are mainly an issue in the continuation store, and by our proof for Theorem 10 that cycle detection should have no major performance overhead there.

[5] Publicly available at `https://github.com/noahvanes/scala-am-abstractgc`

Benchmarks. We evaluate the above implementations using a total of 16 Scheme programs; 11 from the Gabriel benchmark suite [18][6] and 5 from the built-in test suite of Scala-AM. Table 1 lists the size of each Scheme program, with the Gabriel ones depicted in **bold**. For each benchmark program, we configure each abstract interpreter with a monovariant allocator (i.e., 0CFA), and a lattice that abstracts concrete values to the set of their possible types.

Table 1 Lines of code (LOC) for each benchmark. The Gabriel benchmarks are in **bold**.

Benchmark	LOC	Benchmark	LOC	Benchmark	LOC	Benchmark	LOC
boyer	568	**deriv**	39	**takl**	18	gcipd	12
browse	78	**destruc**	37	**puzzle**	144	primtest	35
cpstak	19	**diviter**	8	**triangl**	40	rsa	41
dderiv	83	**divrec**	11	collatz	22	nqueens	34

Measurements. As the implementations differ only in their approach to abstract garbage collection, we follow Earl et al. [16] in the use of the number of explored states as a measure of *precision*. As garbage can cause an abstract interpreter to explore spurious states (cf. Section 1.2), fewer states implies higher precision when all configuration parameters are kept constant. For *performance*, we measure the total running time of the abstract interpreter. Note that running time is impacted by two factors. First, the number of number states the interpreter has to explore. Higher precision can often lead to higher performance. Second, the rate at which the abstract interpreter is able to explore those states. This rate is influenced by the *overhead* caused by abstract garbage collection, which we therefore also measure per state. For the tracing GC approaches ($(\widehat{\rightarrow}_{\widehat{\Gamma}})$ and $(\widehat{\rightarrow}_{\Gamma\mathsf{CFA}})$), this measure corresponds to the time spent in the function $\widehat{\Gamma}$. For approaches based on abstract reference counting ($(\widehat{\rightarrow}_{\mathsf{arc}})$, $(\widehat{\rightarrow}_{\mathsf{arc}+})$ and $(\widehat{\rightarrow}_{\mathsf{arc}++})$), this measure includes the time spent on updating the $\widehat{\phi}$ and $\widehat{\pi}$ components (including cycle detection for $(\widehat{\rightarrow}_{\mathsf{arc}+})$ and $(\widehat{\rightarrow}_{\mathsf{arc}++})$) and in the $\widehat{\mathsf{collect}}$ function. We ran every benchmark 30 times after a warm-up period of 2 minutes. For the time-sensitive measurements, we report the mean of all runs.

5.2 Impact of Cycle Detection

Section 3 introduced transition relation $(\widehat{\rightarrow}_{\mathsf{arc}})$ which performs abstract reference counting, but cannot reclaim cycles. Section 4 extended this relation into $(\widehat{\rightarrow}_{\mathsf{arc}++})$, which can reclaim cyclic garbage by applying cycle detection to both stores. Transition relation $(\widehat{\rightarrow}_{\mathsf{arc}+})$, introduced above, is an in-between: it uses abstract reference counting, but only applies cycle detection to the continuation store. Table 2 compares the corresponding abstract interpreters to evaluate the impact of cycle detection on abstract reference counting.

Impact on precision. The results from Table 2 show that reclaiming garbage cycles from the continuation store only, can already result in important precision improvements. For most benchmarks, $(\widehat{\rightarrow}_{\mathsf{arc}+})$ nearly achieves the same optimal precision as $(\widehat{\rightarrow}_{\mathsf{arc}++})$. For instance, all of the non-Gabriel benchmarks significantly improve precision with cycle detection for the continuation store only, and adding cycle detection to the value store does not increase

[6] We omitted the *ctak* benchmark program due to its use of `call/cc`, which is not yet supported by the abstract interpretation framework.

precision any further. In addition, for the *triangl* benchmark, not reclaiming garbage cycles from the continuation store (using $(\widehat{\rightarrow}_{\mathsf{arc}})$) causes the state space to blow up. However, it is clear that reclaiming them from the continuation store does not always suffice: benchmarks *destruc* and *puzzle* both create cycles in the value store, which are not reclaimed by $(\widehat{\rightarrow}_{\mathsf{arc}})$ nor $(\widehat{\rightarrow}_{\mathsf{arc+}})$. Programs that are written in CPS-style, such as *cpstak*, do not benefit from the extra precision in the continuation store either, as their control flow is encoded in closures that reside in the value store. The garbage-free transition relation $(\widehat{\rightarrow}_{\mathsf{arc++}})$ improves precision more consistently, due to its reclamation of all garbage cycles from both stores. These results underline that cycles cannot be neglected in abstract reference counting, and that a technique for their detection is required to realize the benefits of garbage-free abstract interpretation.

Table 2 Comparison of time taken in milliseconds (t; lower means better performance) and number of states explored (#s; lower means better precision). A time of ∞ means that the benchmark exceeded the time limit of 30 minutes; in this case, we report the number of states explored when the timeout was reached.

	$(\widehat{\rightarrow}_{\mathsf{arc}})$		$(\widehat{\rightarrow}_{\mathsf{arc+}})$		$(\widehat{\rightarrow}_{\mathsf{arc++}})$	
	t	#s	t	#s	t	#s
cpstak	1	94	1	94	1	88
diviter	13	163	14	163	14	162
divrec	15	153	14	146	16	145
destruc	1 208	61 021	1 160	61 021	304	17 344
triangl	∞	>2 186 761	2 044	2 639	1 613	2 639
puzzle	23 950	1 060 585	22 142	1 018 345	2 351	117 572
takl	14 375	862 833	3 581	188 552	2 791	188 549
browse	∞	>4 028 742	∞	>2 586 788	∞	>2 559 221
boyer	∞	>18 544 242	∞	>12 690 008	∞	>12 715 066
deriv	∞	>8 650 787	∞	>28 905 027	∞	>29 513 647
dderiv	∞	>11 561 462	∞	>32 582 390	∞	>32 803 534
collatz	1	60	1	60	1	60
gcipd	1	151	1	91	1	91
primtest	40	5 347	11	1 537	12	1 537
rsa	87	10 646	12	1 538	12	1 538
nqueens	1 207	112 579	319	31 883	338	31 857

Impact on performance. Figure 10 compares the time spent on GC per computed state for the reference-counting transition relations. For most benchmarks, we observe that cycle detection increases the GC overhead slightly. However, this overhead is negligible compared to the corresponding improvements to precision. That is, the overheads in Figure 10 remain comparable for different configurations on each benchmark, while the number of states explored reported in Table 2 decreases significantly when using cycle detection. We observe proportionally large improvements to performance, as the abstract interpreter has fewer states to explore. Again, the garbage-free transition relation $(\widehat{\rightarrow}_{\mathsf{arc++}})$ can most consistently improve performance by completely avoiding any unnecessary computation, with only a negligible increase in overhead. We conclude that the benefits of cycle detection outweigh its cost, as $(\widehat{\rightarrow}_{\mathsf{arc++}})$ is superior to both $(\widehat{\rightarrow}_{\mathsf{arc}})$ and $(\widehat{\rightarrow}_{\mathsf{arc+}})$ in terms of precision and performance.

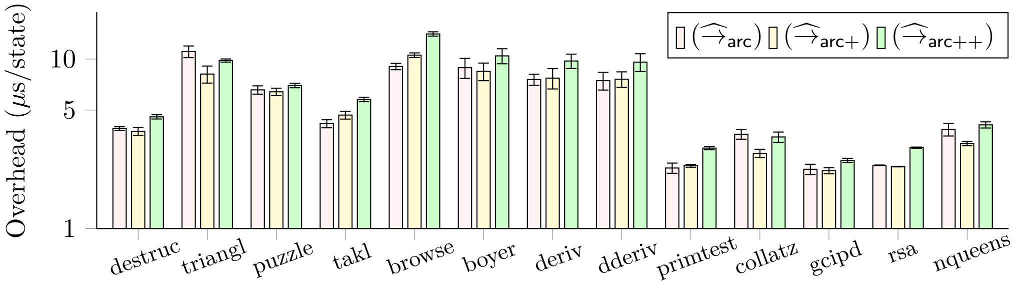


Figure 10 Comparison of overhead measured in time spent on GC per state (lower is better). Error bars indicate the standard deviation of the mean of our measurements (for 30 runs). Note the usage of a logarithmic scale on the y-axis.

Note that Table 2 also lists the number of states that were explored for the benchmarks that did time out. This gives a rough estimate of the interpreters state exploration rate. However, these rates should be interpreted with caution, as some paths in the state space can be significantly cheaper to explore than others. One should therefore only compare the rates of interpreters with the same precision, or look for a pattern of major discrepancies. We do not discern such a pattern among the rates, and deem the differences in precision too outspoken to warrant a direct rate comparison.

5.3 Comparison to Existing Policies for Abstract GC

Table 3 compares $(\widehat{\rightarrow}_{\mathsf{arc++}})$ to other approaches to abstract garbage collection. On one end of the spectrum, there is $(\widehat{\rightarrow})$, which never applies abstract GC. On the other end, there is $(\widehat{\rightarrow}_{\widehat{\Gamma}})$, which applies abstract GC on every evaluation step. An interesting tradeoff is made by $(\widehat{\rightarrow}_{\Gamma\mathsf{CFA}})$, which applies GC before every store join.

Table 3 Comparison of time taken in milliseconds (t; lower means better performance) and number of states explored (#s; lower means better precision) by an abstract interpreter for different approaches to abstract GC. A time of ∞ means that the benchmark exceeded the time limit of 30 minutes; in this case, we report the number of states explored when the timeout was reached.

	$(\widehat{\rightarrow})$		$(\widehat{\rightarrow}_{\widehat{\Gamma}})$		$(\widehat{\rightarrow}_{\Gamma\mathsf{CFA}})$		$(\widehat{\rightarrow}_{\mathsf{arc++}})$	
	t	#s	t	#s	t	#s	t	#s
cpstak	1	120	7	88	2	94	1	88
diviter	14	175	29	162	13	163	14	162
divrec	24	219	28	145	16	148	16	145
destruc	9 671	381 949	2 436	17 344	735	22 444	304	17 344
triangl	∞	>4 826 189	2 018	2 639	1 635	2 767	1 613	2 639
puzzle	∞	>104 243 260	24 704	117 572	14 732	293 167	2 351	117 572
takl	86 830	7 072 820	72 325	377 389	32 356	539 233	2 791	188 549
browse	∞	>6 608 260	∞	>1 928 827	∞	>2 706 491	∞	>2 559 221
boyer	∞	>27 731 639	∞	>232 889	∞	>1 456 963	∞	>12 715 066
deriv	∞	>107 328 548	∞	>5 773 009	∞	>13 262 924	∞	>29 513 647
dderiv	∞	>90 849 930	∞	>3 320 410	∞	>12 954 310	∞	>32 803 534
collatz	1	431	4	60	2	159	1	60
gcipd	3	1 098	8	91	2	147	1	91
primtest	2 249	334 944	166	1 537	75	3 622	12	1 537
rsa	1 840	247 915	176	1 538	63	2 375	12	1 538
nqueens	346 349	37 499 432	4 583	37 847	1 556	52 766	338	31 857

Comparison of precision. Our results confirm the findings of previous work [42, 43, 16, 28]: abstract GC reduces the state space to explore substantially. Looking at the number of explored states in Table 3 reveals that the interpreter without abstract GC using $(\widehat{\rightarrow})$ has the lowest precision for every benchmark compared to all other approaches. The interpreter using $(\widehat{\rightarrow}_{\Gamma\mathsf{CFA}})$ sacrifices some precision by invoking abstract GC less frequently than $(\widehat{\rightarrow}_{\widehat{\Gamma}})$. It explores more states than necessary because it is not garbage-free, which is the most noticeable on more complex programs such as *destruct*, *puzzle* and *takl*.

Our approach, $(\widehat{\rightarrow}_{\mathsf{arc}++})$, achieves the same optimal precision as $(\widehat{\rightarrow}_{\widehat{\Gamma}})$. As we have proven that both transition relations are garbage-free and equivalent, they should explore the same number of states. Note, however, that this is not the case for the *takl* benchmark – an apparent contradiction of Theorem 12. The reason is that the implementation of $(\widehat{\rightarrow}_{\mathsf{arc}++})$ for Scheme is able to collect more garbage than its formalization for λ_{ANF} in Sections 3 and 4. The definition of garbage-free only requires the absence of garbage from the stores *in between* transitions. However, more complex state transitions can create garbage and immediately bring that garbage back to life *within* a single transition. Unlike our formalization for λ_{ANF}, some of the transition rules in our implementation for Scheme both lookup and insert continuations at the same address $\widehat{a}_k$ in a single step. Abstract reference counting can automatically deallocate $\widehat{a}_k$ (and addresses in the same SCC) if no references remain to $\widehat{a}_k$ after the lookup, which possibly prevents a precision loss when $\widehat{a}_k$ (or another address in the same SCC) is reallocated in the same step. Abstract reference counting can trivially reclaim garbage within a single transition due to its continuous nature. Realizing the same effect with abstract tracing GC (i.e., using $(\widehat{\rightarrow}_{\widehat{\Gamma}})$ and $(\widehat{\rightarrow}_{\mathsf{arc}++})$) would require a full GC application both within and after such transitions – increasing its overhead further. We investigated the applicability of this optimization on abstract reference counting for all benchmarks, and found it limited to *takl*, *browse*, *deriv*, *dderiv* and *nqueens*. For the others, $(\widehat{\rightarrow}_{\widehat{\Gamma}})$ and $(\widehat{\rightarrow}_{\mathsf{arc}++})$ exhibit the same precision and explore the same state space.

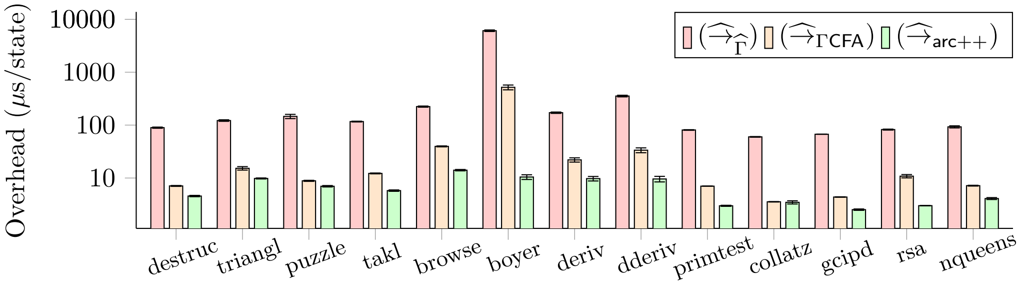


Figure 11 Comparison of overhead measured in time spent on GC per state (lower is better). Error bars indicate the standard deviation of the mean of our measurements (for 30 runs). Note the usage of a logarithmic scale on the y-axis.

Comparison of performance. Figure 11 compares the overhead of abstract GC for $(\widehat{\rightarrow}_{\widehat{\Gamma}})$, $(\widehat{\rightarrow}_{\Gamma\mathsf{CFA}})$ and $(\widehat{\rightarrow}_{\mathsf{arc}++})$. The $(\widehat{\rightarrow})$ interpreter does not suffer such overhead. However, without any abstract GC, its state space becomes polluted (leading to repeated exploration of equivalent states) and explodes (due to infeasible paths). It exhibits the worst performance.

The overhead of abstract GC is an issue for $(\widehat{\rightarrow}_{\widehat{\Gamma}})$. It is clear that a tracing GC application is generally more expensive than a single evaluation step. Applying tracing GC after every step leads to the interpreter's running time being dominated by that of the GC applications. This is outspoken on complex benchmarks that require larger heaps such as *boyer*: since tracing GC needs to traverse the entire heap, the overhead grows with the size of the heap. For the *boyer* benchmark, the interpreter using $(\widehat{\rightarrow}_{\mathsf{arc{+}{+}}})$, which explores the same state space as $(\widehat{\rightarrow}_{\widehat{\Gamma}})$ for this benchmark, explores 12.7M states before timeout, whereas with the excessive GC overhead of $(\widehat{\rightarrow}_{\widehat{\Gamma}})$, it can only explore 233K states with the same time limit. While the garbage-freeness of $(\widehat{\rightarrow}_{\widehat{\Gamma}})$ significantly improves overall performance of $(\widehat{\rightarrow})$ by reducing the state space, it is not as fast as $(\widehat{\rightarrow}_{\mathsf{arc{+}{+}}})$ due to the higher overhead. In Figure 11, for all benchmarks the overhead of $(\widehat{\rightarrow}_{\widehat{\Gamma}})$ differs to that of $(\widehat{\rightarrow}_{\mathsf{arc{+}{+}}})$ by orders of magnitude, which results in significant performance improvements in Table 3 while exploring the same amount of states. The transition relation $(\widehat{\rightarrow}_{\Gamma\mathsf{CFA}})$ avoids this overhead of $(\widehat{\rightarrow}_{\widehat{\Gamma}})$ by applying abstract GC less frequently. As such, its overhead is significantly lower than that of $(\widehat{\rightarrow}_{\widehat{\Gamma}})$. For all our benchmarks, this results in a performance improvement over $(\widehat{\rightarrow}_{\widehat{\Gamma}})$, despite having to explore a larger state space. Note that $(\widehat{\rightarrow}_{\Gamma\mathsf{CFA}})$ is no longer garbage-free and explores more states than necessary. For instance, on the *puzzle* benchmark it almost explores 3 times as many states as $(\widehat{\rightarrow}_{\mathsf{arc{+}{+}}})$. In addition, Figure 11 reveals that the overhead remains considerably higher than that of $(\widehat{\rightarrow})$ and even $(\widehat{\rightarrow}_{\mathsf{arc{+}{+}}})$. This can also be seen in the benchmarks that timeout, where the difference in states explored is significant compared to $(\widehat{\rightarrow})$ and even $(\widehat{\rightarrow}_{\mathsf{arc{+}{+}}})$. In particular, the *boyer* benchmark also shows that it still scales poorly towards larger heap sizes, an inherent downside of tracing GC approaches. Since both overhead and number of states explored are higher than for $(\widehat{\rightarrow}_{\mathsf{arc{+}{+}}})$, the overall performance is worse.

In general, when determining an application policy for abstract tracing GC, sacrificing garbage-freeness for increased performance implies lower precision. Using abstract reference counting with $(\widehat{\rightarrow}_{\mathsf{arc{+}{+}}})$ gives the best of both worlds. Its low overhead is not detrimental to performance: although there is definitely some compared to $(\widehat{\rightarrow})$, it is consistently lower than all other approaches that actually apply abstract GC. Compared to the equivalent transition relation $(\widehat{\rightarrow}_{\widehat{\Gamma}})$ with optimal precision, overhead is reduced by orders of magnitude. The overall performance of the *triangl* benchmark is an exception to the rule: even though the GC overhead is greatly reduced when using $(\widehat{\rightarrow}_{\mathsf{arc{+}{+}}})$, its running time is mostly dominated by expensive evaluation steps, but even in this case none of the other interpreters outperform $(\widehat{\rightarrow}_{\mathsf{arc{+}{+}}})$. Since $(\widehat{\rightarrow}_{\mathsf{arc{+}{+}}})$ achieves the optimal precision, so that overall performance is not impacted by the exploration of spurious states. Hence, our experiments show that $(\widehat{\rightarrow}_{\mathsf{arc{+}{+}}})$ is superior both in terms of precision and performance.

6 Related Work

The focus of our work is a more efficient approach to abstract garbage collection. It relates to previous work on concrete GC, abstract GC, and abstract interpretation in general.

Reference Counting. Our approach is based on the original formulation of reference counting [10], without any common optimizations [50] such as deferred [15] and coalesced [35] reference counting or “lazy freeing” [8]. As these optimizations postpone the reclamation of garbage, they cannot be used for garbage-free abstract interpretation.

We perform abstract reference counting at the level of SCCs, so that garbage cycles can be reclaimed. For concrete reference counting, two major techniques have been proposed to handle cycles: using a back-up tracing garbage collector [56] and trial deletion [9]. We deem both too expensive for frequent applications during abstract interpretation, which is required to guarantee garbage-freeness. Nevertheless, it could be interesting to investigate a variant of abstract reference counting where abstract tracing GC is applied at the evaluation steps that might render a cycle into a garbage cycle. In the concrete, this has been shown [37] to occur when a memory location loses some, but not all of its incoming references. A cheap pre-analysis as in [3] could identify the structures guaranteed to be acyclic beforehand.

Confusingly, the term “abstract reference counting” has been used with different meanings. Hudak [24] formalizes reference counting for a concrete interpreter. After abstraction, he obtains a static analysis that approximates the *concrete* reference count, enabling program optimizations [25, 45] and compile-time GC [32]. The analysis exemplifies static analysis *of* reference counting, rather than static analysis analysis *with* reference counting for the purpose of improving precision and performance. Finally, Jones et al. [31] use the term to abstract over the low-level details concrete implementations differ on.

Tracing Garbage Collection. Our main issue with tracing GC is that it requires frequent heap traversals to keep the abstract interpreter garbage-free. Various techniques exist for concrete interpreters to avoid the correspondingly long GC pauses and to improve their throughput. Incremental algorithms, such as Baker’s incremental stop-and-copy algorithm [33], only remove garbage lazily, and can therefore not be used in our setting. Moreover, they would need to be redesigned for abstract interpreters which do not always allocate a fresh address. Generational algorithms [36] can increase throughput by limiting heap traversals to the young generations only. However, as garbage also arises in older ones, it does not suffice to keep the abstract interpreter garbage-free. A full GC would be required to prevent precision loss for addresses that are re-allocated in the old generation.

Abstract Garbage Collection. The ΓCFA analysis [42, 43] pioneered the concept of abstract garbage collection. The term “garbage-free” was later coined in [23] to describe an abstract interpreter that applies abstract GC at every evaluation step, eliminating all garbage. ΓCFA also incorporated *abstract counting* —not to be confused with abstract reference counting – to count how often an address has been allocated. Abstract counting combines well with abstract GC, as collecting garbage results in more precise (i.e., lower) abstract counts. It can be combined directly with abstract reference counting to the same effect.

We showed that garbage-free abstract interpretation prevents imprecision and improves the interpreter’s performance by limiting exploration to garbage-free states. As such, it improves what is known as *localization* [48]; unnecessary store bindings are removed so that states with irrelevant store differences need not be explored multiple times. For similar purposes, Oh et al. [47] eliminate store bindings in their analysis based on which addresses can be used. The main difference with our and other approaches to abstract GC is the use of a pre-analysis to conservatively approximate which bindings are never used and can safely be removed as garbage. The technique can be combined with reachability-based approaches, such as abstract reference counting, to further improve localization and precision. Might et al. [41] proposed a similar extension to abstract GC, named *conditional* abstract GC, which only keeps bindings in the store that are reachable and satisfy certain conditions. However, it relies on a theorem prover and has to the best of our knowledge not yet been implemented.

In Knauel’s abstract interpretation framework [34], the overhead of tracing GC turned proved performance-detrimental on complex programs. The proposed solution is a *global garbage collector* which collects garbage for multiple states at once with a global root set. Of course, this sacrifices many of the precision benefits of the GC. To our knowledge, no prior work has employed abstract reference counting as a more efficient approach to garbage-free abstract interpretation, despite reference counting being mentioned in the future work of [42]. Bergstrom et al. [6, 7] employ a primitive form of reference counting as a cheaper alternative to full abstract GC. Their control-flow analysis (without explicit store component) tracks for every variable if it has been captured by the lexical environment of another closure; if not, this variable can safely be collected once it goes out of scope. Compared to our approach, this corresponds to abstract reference counting with reference counts from $\widehat{\mathbb{N}} = \{0, 1, \infty\}$. We have shown that such an approximation is unnecessary, as the precise reference count can be maintained without having to increase the complexity of the state space.

Abstract Interpretation. Many abstract interpreters incorporate global store widening [23] to terminate within reasonable time. Unfortunately, previous work [28] has shown that store widening and abstract GC do not combine well: abstract GC can collect fewer addresses, as it needs to account for reachability from multiple states, and store widening can no longer guarantee fast convergence, as the GC violates the monotonicity of the global store. In fact, an often overlooked consequence of abstract GC is that it no longer suffices for lattices to conform to the ascending chain condition; as GC can "descend" into lattices, the analysis is no longer monotonous, and the only way to guarantee termination is to use finite lattices.

Other work [16, 28] has been able to combine abstract GC successfully with pushdown analysis, which substantially increases control-flow precision by properly matching call and return sites. The combination yields "better-than-both-worlds" improvements, although the techniques' inherent incompatibilities need to be overcome. We leave an analogous combination with abstract reference counting open for future work.

Finally, abstract counting can be exploited to enable strong updates [38, 4]. Replacing join operations in the store with overwrites when possible increases precision. Note, however, that such strong updates break the assumption in Section 4 that stores only grow monotonically without GC. To support strong updates, our cycle detection technique therefore needs to be adapted to take into account the possibility that a SCC can break up when a strong update occurs within a single SCC (although this cannot happen in the continuation store).

Incremental Cycle Detection. To reclaim garbage cycles, our approach detects cycles as references are added to the store. We implement such *incremental cycle detection* using a simple backward search, which has suboptimal worst-case performance in theory. More advanced algorithms have been proposed [22, 5] with better asymptotic performance. However, empirical evidence suggests [52] that more complicated algorithms do not always perform better in practice for some situations. Indeed, in Section 4.2 we argued that backward searching usually works well for an abstract interpreter, and we verified this empirically in Section 5.2. Nevertheless, it would be interesting for future work to explore whether other incremental cycle detection algorithms can be more efficient for an abstract interpreter.

7 Conclusion

We have introduced abstract reference counting as a more efficient approach to abstract garbage collection. Existing approaches are based on tracing GC, which is non-continuous in nature, and therefore need to trade-off either precision or performance to strike the balance between the benefits and overhead of abstract GC. Our approach based on reference counting, in contrast, requires but minor bookkeeping yet is provably sound and complete in terms of abstract GC. The result is a garbage-free abstract interpreter that is as precise as one that applies tracing GC at every step – in the absence of cycles.

However, we showed that cycles are a major threat to the garbage-freeness of abstract reference counting. Next to those arising under concrete interpretation, the address allocator of an abstract interpreter can create many artificial cycles. We proposed cycle detection as a solution, maintaining the reference counts at the level of the strongly connected components in the stores. This enables the reclamation of garbage cycles, which is necessary to make abstract reference counting garbage-free.

In terms of precision, our empirical experiments confirm our claim on the garbage-freeness of the abstract interpreter, showing that it achieves the optimal precision for abstract garbage collection. In terms of performance, abstract reference counting – even with cycle detection – greatly reduces overhead by avoiding frequent heap traversals. Without sacrificing precision, it can therefore realize significant performance improvements in abstract interpretation.

References

1 Leif Andersen and Matthew Might. Multi-core Parallelization of Abstracted Abstract Machines. In *Proceedings of the 2013 Workshop on Scheme and Functional Programming, Washington, DC*. Citeseer, 2013.

2 David F. Bacon, Perry Cheng, and V. T. Rajan. A unified theory of garbage collection. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2004, October 24-28, 2004, Vancouver, BC, Canada*, pages 50–68, 2004. doi:10.1145/1028976.1028982.

3 David F. Bacon and V. T. Rajan. Concurrent Cycle Collection in Reference Counted Systems. In *ECOOP 2001 - Object-Oriented Programming, 15th European Conference, Budapest, Hungary, June 18-22, 2001, Proceedings*, pages 207–235, 2001. doi:10.1007/3-540-45337-7_12.

4 Gogul Balakrishnan and Thomas W. Reps. Recency-Abstraction for Heap-Allocated Storage. In *Static Analysis, 13th International Symposium, SAS 2006, Seoul, Korea, August 29-31, 2006, Proceedings*, pages 221–239, 2006. doi:10.1007/11823230_15.

5 Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, and Robert E. Tarjan. A New Approach to Incremental Cycle Detection and Related Problems. *ACM Trans. Algorithms*, 12(2):14:1–14:22, 2016. doi:10.1145/2756553.

6 Lars Bergstrom. Arity raising and control-flow analysis in Manticore. Master's thesis, University of Chicago, 2009.

7 Lars Bergstrom, Matthew Fluet, Matthew Le, John H. Reppy, and Nora Sandler. Practical and effective higher-order optimizations. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*, pages 81–93, 2014. doi:10.1145/2628136.2628153.

8 Hans-Juergen Boehm. The space cost of lazy reference counting. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*, pages 210–219, 2004. doi:10.1145/964001.964019.

9 Thomas W. Christopher. Reference Count Garbage Collection. *Softw., Pract. Exper.*, 14(6):503–507, 1984. doi:10.1002/spe.4380140602.

10 George E. Collins. A method for overlapping and erasure of lists. *Commun. ACM*, 3(12):655–657, 1960. doi:10.1145/367487.367501.

11 Patrick Cousot. The Verification Grand Challenge and Abstract Interpretation. In *Verified Software: Theories, Tools, Experiments, First IFIP TC 2/WG 2.3 Conference, VSTTE 2005, Zurich, Switzerland, October 10-13, 2005, Revised Selected Papers and Discussions*, pages 189–201, 2005. doi:10.1007/978-3-540-69149-5_21.

12 Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, pages 238–252, 1977. doi:10.1145/512950.512973.

13 Patrick Cousot and Radhia Cousot. Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation. In *Programming Language Implementation and Logic Programming, 4th International Symposium, PLILP'92, Leuven, Belgium, August 26-28, 1992, Proceedings*, pages 269–295, 1992. doi:10.1007/3-540-55844-6_142.

14 David Darais, Nicholas Labich, Phuc C. Nguyen, and David Van Horn. Abstracting definitional interpreters (functional pearl). *PACMPL*, 1(ICFP):12:1–12:25, 2017. doi:10.1145/3110256.

15 L. Peter Deutsch and Daniel G. Bobrow. An Efficient, Incremental, Automatic Garbage Collector. *Commun. ACM*, 19(9):522–526, 1976. doi:10.1145/360336.360345.

16 Christopher Earl, Ilya Sergey, Matthew Might, and David Van Horn. Introspective pushdown analysis of higher-order programs. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'12, Copenhagen, Denmark, September 9-15, 2012*, pages 177–188, 2012. doi:10.1145/2364527.2364576.

17 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.

18 Richard P. Gabriel. *Performance and evaluation of LISP systems*, volume 263. MIT press Cambridge, Mass., 1985.

19 Zvi Galil and Giuseppe F. Italiano. Data structures and algorithms for disjoint set union problems. *ACM Computing Surveys (CSUR)*, 23(3):319–344, 1991.

20 Thomas Gilray, Michael D. Adams, and Matthew Might. Allocation characterizes polyvariance: a unified methodology for polyvariant control-flow analysis. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, pages 407–420, 2016. doi:10.1145/2951913.2951936.

21 Thomas Gilray, Steven Lyde, Michael D. Adams, Matthew Might, and David Van Horn. Pushdown control-flow analysis for free. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 691–704, 2016. doi:10.1145/2837614.2837631.

22 Bernhard Haeupler, Telikepalli Kavitha, Rogers Mathew, Siddhartha Sen, and Robert Endre Tarjan. Incremental Cycle Detection, Topological Ordering, and Strong Component Maintenance. *ACM Trans. Algorithms*, 8(1):3:1–3:33, 2012. doi:10.1145/2071379.2071382.

23 David Van Horn and Matthew Might. Abstracting abstract machines. In *Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*, pages 51–62, 2010. doi:10.1145/1863543.1863553.

24 Paul Hudak. A semantic model of reference counting and its abstraction (detailed summary). In *Proceedings of the 1986 ACM conference on LISP and functional programming*, pages 351–363. ACM, 1986.

25 Paul Hudak and Adrienne G. Bloss. The Aggregate Update Problem in Functional Programming Systems. In *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages, New Orleans, Louisiana, USA, January 1985*, pages 300–314, 1985. doi:10.1145/318593.318660.

26 Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type Analysis for JavaScript. In *Static Analysis, 16th International Symposium, SAS 2009, Los Angeles, CA, USA, August 9-11, 2009. Proceedings*, pages 238–255, 2009. doi:10.1007/978-3-642-03237-0_17.

27 J. Ian Johnson, Nicholas Labich, Matthew Might, and David Van Horn. Optimizing abstract abstract machines. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, pages 443–454, 2013. doi:10.1145/2500365.2500604.

28 J. Ian Johnson, Ilya Sergey, Christopher Earl, Matthew Might, and David Van Horn. Pushdown flow analysis with abstract garbage collection. *J. Funct. Program.*, 24(2-3):218–283, 2014.

29 James Ian Johnson. *Automating abstract interpretation of abstract machines*. PhD thesis, Northeastern University, 2015.

30 James Ian Johnson and David Van Horn. Abstracting abstract control. In *DLS'14, Proceedings of the 10th ACM Symposium on Dynamic Languages, part of SLASH 2014, Portland, OR, USA, October 20-24, 2014*, pages 11–22, 2014. doi:10.1145/2661088.2661098.

31 Richard Jones, Antony Hosking, and Eliot Moss. *The garbage collection handbook: the art of automatic memory management*. Chapman and Hall/CRC, 2016.

32 Simon B. Jones and Daniel Le Métayer. Compile-time garbage collection by sharing analysis. In *Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 54–74. ACM, 1989.

33 Henry G. Baker Jr. List Processing in Real Time on a Serial Computer. *Commun. ACM*, 21(4):280–294, 1978. doi:10.1145/359460.359470.

34 Eric Jean Knauel. *A flow analysis framework for realistic scheme programs*. PhD thesis, University of Tübingen, Germany, 2008. URL: http://tobias-lib.ub.uni-tuebingen.de/volltexte/2008/3363/.

35 Yossi Levanoni and Erez Petrank. An on-the-fly reference-counting garbage collector for java. *ACM Trans. Program. Lang. Syst.*, 28(1):1–69, 2006. doi:10.1145/1111596.1111597.

36 Henry Lieberman and Carl Hewitt. A Real-Time Garbage Collector Based on the Lifetimes of Objects. *Commun. ACM*, 26(6):419–429, 1983. doi:10.1145/358141.358147.

37 Alejandro D. Martínez, Rosita Wachenchauzer, and Rafael D. Lins. Cyclic reference counting with local mark-scan. *Information Processing Letters*, 34(1):31–35, 1990.

38 Matthew Might. *Environment Analysis of Higher-Order Languages*. PhD thesis, Georgia Institute of Technology, Atlanta, GA, USA, 2007. URL: http://hdl.handle.net/1853/16289.

39 Matthew Might. Logic-flow analysis of higher-order programs. In *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*, pages 185–198, 2007. doi:10.1145/1190216.1190247.

40 Matthew Might. Abstract Interpreters for Free. In *Static Analysis - 17th International Symposium, SAS 2010, Perpignan, France, September 14-16, 2010. Proceedings*, pages 407–421, 2010. doi:10.1007/978-3-642-15769-1_25.

41 Matthew Might, Benjamin Chambers, and Olin Shivers. Model Checking Via GammaCFA. In *Verification, Model Checking, and Abstract Interpretation, 8th International Conference, VMCAI 2007, Nice, France, January 14-16, 2007, Proceedings*, pages 59–73, 2007. doi:10.1007/978-3-540-69738-1_4.

42 Matthew Might and Olin Shivers. Improving flow analyses via ΓCFA: Abstract garbage collection and counting. In *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming, ICFP 2006, Portland, Oregon, USA, September 16-21, 2006*, pages 13–25, 2006. doi:10.1145/1159803.1159807.

43 Matthew Might and Olin Shivers. Exploiting reachability and cardinality in higher-order flow analysis. *J. Funct. Program.*, 18(5-6):821–864, 2008. doi:10.1017/S0956796808006941.

44 Matthew Might, Yannis Smaragdakis, and David Van Horn. Resolving and exploiting the k-CFA paradox: illuminating functional vs. object-oriented program analysis. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*, pages 305–315, 2010. doi:10.1145/1806596.1806631.

45 Alan Mycroft. *Abstract interpretation and optimising transformations for applicative programs*. PhD thesis, University of Edinburgh, UK, 1982. URL: http://hdl.handle.net/1842/6602.

46 Jens Nicolay, Carlos Noguera, Coen De Roover, and Wolfgang De Meuter. Detecting function purity in JavaScript. In *15th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2015, Bremen, Germany, September 27-28, 2015*, pages 101–110, 2015. doi:10.1109/SCAM.2015.7335406.

47 Hakjoo Oh and Kwangkeun Yi. Access-based abstract memory localization in static analysis. *Sci. Comput. Program.*, 78(9):1701–1727, 2013. doi:10.1016/j.scico.2013.04.002.

48 Noam Rinetzky, Jörg Bauer, Thomas W. Reps, Shmuel Sagiv, and Reinhard Wilhelm. A semantics for procedure local heaps and its abstractions. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, pages 296–309, 2005. doi:10.1145/1040305.1040330.

49 Ilya Sergey, Dominique Devriese, Matthew Might, Jan Midtgaard, David Darais, Dave Clarke, and Frank Piessens. Monadic abstract interpreters. In *ACM SIGPLAN Conference on*

Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013, pages 399–410, 2013. `doi:10.1145/2491956.2491979`.

50 Rifat Shahriyar, Stephen M. Blackburn, and Daniel Frampton. Down for the count? Getting reference counting back in the ring. In *International Symposium on Memory Management, ISMM '12, Beijing, China, June 15-16, 2012*, pages 73–84, 2012. `doi:10.1145/2258996.2259008`.

51 Olin Shivers. Control-Flow Analysis in Scheme. In *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, USA, June 22-24, 1988*, pages 164–174, 1988. `doi:10.1145/53990.54007`.

52 Ragnar Lárus Sigurðsson. Practical performance of incremental topological sorting and cycle detection algorithms. Master's thesis, Chalmers University of Technology, 2016.

53 Quentin Stiévenart, Jens Nicolay, Wolfgang De Meuter, and Coen De Roover. Building a modular static analysis framework in Scala (tool paper). In *Proceedings of the 7th ACM SIGPLAN Symposium on Scala, SCALA@SPLASH 2016, Amsterdam, Netherlands, October 30 - November 4, 2016*, pages 105–109, 2016. `doi:10.1145/2998392.3001579`.

54 Quentin Stiévenart, Maarten Vandercammen, Wolfgang De Meuter, and Coen De Roover. Scala-AM: A Modular Static Analysis Framework. In *16th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2016, Raleigh, NC, USA, October 2-3, 2016*, pages 85–90, 2016. `doi:10.1109/SCAM.2016.14`.

55 Robert Endre Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic Discrete Methods*, 6(2):306–318, 1985.

56 Joseph Weizenbaum. Recovery of reentrant list structures in SLIP. *Commun. ACM*, 12(7):370–372, 1969. `doi:10.1145/363156.363159`.

A Proofs

We detail here the proofs of Theorems 1, 9 and 10, as well as the proofs of Lemmas 5 and 7.

▶ **Theorem 1.** *If* $\varsigma_0 \xrightarrow{n}_\Gamma \varsigma_1$ *and* $\varsigma_0 \xrightarrow{n} \varsigma_2$, *then* $\varsigma_1 = \Gamma(\varsigma_2)$.

Proof. This follows directly from theorems in [42]: both $(\rightarrow)$ and $(\rightarrow_\Gamma) = (\Gamma \circ (\rightarrow))$ are subsets of the non-deterministic transition relation $(\Rrightarrow)$. Hence, by Theorem 6.10 of [42], $\Gamma(\varsigma_1) = \Gamma(\varsigma_2)$ and by Lemma 6.8 of [42] and the definition of $(\rightarrow_\Gamma)$, $\Gamma(\varsigma_1) = \varsigma_1$, so that $\varsigma_1 = \Gamma(\varsigma_2)$. Both proofs of Theorem 6.10 and Lemma 6.8 in [42] are presented for a simple CPS-language, but can be repeated *mutatis mutandis* for λ_{ANF}. ◀

▶ **Lemma 5.** $(\widehat{\rightarrow}_{\mathsf{arc}})$ *is sound.*

Proof. Assuming $\hat{\varsigma}$ is sound and $\hat{\varsigma} \mathrel{\widehat{\rightarrow}_{\mathsf{arc}}} \hat{\varsigma}'$, by inversion we have that $\exists \hat{\varsigma}_0$ so that $\hat{\varsigma} \mathrel{\widehat{\rightarrow}_0} \hat{\varsigma}_0$ and $\hat{\varsigma}' = \widehat{\mathsf{collect}}(\hat{\varsigma}, \hat{\varsigma}_0)$. A trivial case analysis of $(\widehat{\rightarrow}_0)$ shows that $(\widehat{\rightarrow}_0)$, and therefore also $\hat{\varsigma}_0$ are sound. We show that whenever $\widehat{\mathsf{collect}}$ adds an address $\hat{l}$ to $\widehat{G}$, we have that $\hat{l} \notin \widehat{\mathcal{R}}(\hat{\varsigma}_0)$, so that by induction $\forall \hat{l} \in \widehat{G} : \hat{l} \notin \widehat{\mathcal{R}}(\hat{\varsigma}_0)$. It is clear that whenever we add $\hat{l}$ to $\widehat{G}$, every $\hat{l}' \in \widehat{\phi}_{\hat{\varsigma}_0}(\hat{l})$ must have already been deleted from $\widehat{\phi}_{\hat{\varsigma}_0}(\hat{l})$, which implies that $\hat{l}' \in \widehat{G}$. Hence, for every predecessor $\hat{l}' \in \widehat{\phi}_{\hat{\varsigma}_0}(\hat{l})$ we have that $\hat{l}' \notin \widehat{\mathcal{R}}(\hat{\varsigma}_0)$, and since $\hat{l}$ can only deallocated when $\hat{l} \notin \widehat{\mathcal{T}}_{\widehat{\Sigma}}(\hat{\varsigma}_0)$, it must be that $\hat{l} \notin \widehat{\mathcal{R}}(\hat{\varsigma}_0)$. By the soundness of $\hat{\varsigma}_0$, we have that $\widehat{\mathcal{R}}(\hat{\varsigma}_0) \subseteq \widehat{\mathcal{S}}(\hat{\varsigma}_0)$. Since we have just shown that $\widehat{\mathsf{collect}}$ only removes addresses from $\hat{\varsigma}_0$ that are not in $\widehat{\mathcal{R}}(\hat{\varsigma}_0)$, we get $\widehat{\mathcal{R}}(\hat{\varsigma}_0) \subseteq \widehat{\mathcal{S}}(\hat{\varsigma}')$ and $\widehat{\mathcal{R}}(\hat{\varsigma}') = \widehat{\mathcal{R}}(\hat{\varsigma}_0)$, so that $\widehat{\mathcal{R}}(\hat{\varsigma}') \subseteq \widehat{\mathcal{S}}(\hat{\varsigma}')$, hence $\hat{\varsigma}'$ is sound. ◀

For convenience, we define $\widehat{\phi}^*_{\hat{\varsigma}}(\hat{l}) = \{\hat{l}' \mid \hat{l}' \overset{*}{\widehat{\rightsquigarrow}}_{\hat{\varsigma}} \hat{l}\} \setminus \{\hat{l}\}$, so that $\widehat{\phi}^*_{\hat{\varsigma}}(\hat{l})$ is the set of all addresses that can directly or indirectly reach $\hat{l}$ (excluding $\hat{l}$ itself).

▶ **Lemma 7.** *In the absence of cycles,* $(\widehat{\rightarrow}_{\mathsf{arc}})$ *is complete.*

Proof. Assume that $\widehat{\varsigma} \mathrel{\widehat{\rightarrow}_{\mathsf{arc}}} \widehat{\varsigma}'$, where $\widehat{\varsigma} \mathrel{\widehat{\rightarrow}_0} \widehat{\varsigma}_0$ and $\widehat{\varsigma}' = \widehat{\mathsf{collect}}(\widehat{\varsigma}, \widehat{\varsigma}_0)$. We need to prove that $\widehat{\mathcal{S}}(\widehat{\varsigma}') \subseteq \widehat{\mathcal{R}}(\widehat{\varsigma}')$. Let $\widehat{l}$ be an address in $\widehat{\mathcal{S}}(\widehat{\varsigma}')$. If $\widehat{l} \notin \widehat{\mathcal{S}}(\widehat{\varsigma})$, then $\widehat{l}$ was added to the store or continuation store by one of the transition rules in $(\widehat{\rightarrow}_0)$: it is clear that in any of those cases, we also have that $\widehat{l} \in \widehat{\mathcal{R}}(\widehat{\varsigma}')$. If $\widehat{l} \in \widehat{\mathcal{S}}(\widehat{\varsigma})$, then by completeness of $\widehat{\varsigma}$, we have that $\widehat{l} \in \widehat{\mathcal{R}}(\widehat{\varsigma})$. Proceed by induction on $|\widehat{\phi}^*_{\widehat{\varsigma}'}(\widehat{l})|$. For the case $|\widehat{\phi}^*_{\widehat{\varsigma}'}(\widehat{l})| = 0$, by definition of $\widehat{\phi}^*$ and $\widehat{\phi}$ we also have that $|\widehat{\phi}_{\widehat{\varsigma}'}(\widehat{l})| = 0$. If $|\widehat{\phi}_{\widehat{\varsigma}}(\widehat{l})| = 0$, then $\widehat{l}$ was checked by $\widehat{\mathsf{collect}}$ because it must have been that $\widehat{l} \in \widehat{\mathcal{T}}_{\widehat{\Sigma}}(\widehat{\varsigma})$ (since $\widehat{l} \in \widehat{\mathcal{R}}(\widehat{\varsigma})$); if $|\widehat{\phi}_{\widehat{\varsigma}}(\widehat{l})| > 0$, then $\widehat{l}$ was also checked because every address in $\widehat{\phi}_{\widehat{\varsigma}}(\widehat{l})$ got removed, which caused $\widehat{l}$ to be added to $\widehat{C}$. Since in both cases $\widehat{l}$ was not removed after the check (where $\widehat{\phi}(\widehat{l}) = \emptyset$) it must have been that $\widehat{l} \in \widehat{\mathcal{T}}_{\widehat{\Sigma}}(\widehat{\varsigma}')$, and hence $\widehat{l} \in \widehat{\mathcal{R}}(\widehat{\varsigma}')$. For the case $|\widehat{\phi}^*_{\widehat{\varsigma}'}(\widehat{l})| = k > 0$, we have that $|\widehat{\phi}_{\widehat{\varsigma}'}(\widehat{l})| > 0$. Pick any predecessor $\widehat{l}' \in \widehat{\phi}_{\widehat{\varsigma}'}(\widehat{l})$; in the absence of cycles, a predecessor always has fewer ancestors, i.e. $|\widehat{\phi}^*_{\widehat{\varsigma}'}(\widehat{l}')| < k$, so that by the induction hypothesis we get that $\widehat{l}' \in \widehat{\mathcal{R}}(\widehat{\varsigma}')$. Since $\widehat{l}' \in \widehat{\mathcal{R}}(\widehat{\varsigma}')$ and $\widehat{l}' \mathrel{\widehat{\rightsquigarrow}_{\widehat{\varsigma}'}} \widehat{l}$ (by definition of $\widehat{\phi}_{\widehat{\varsigma}'}(\widehat{l})$), we have that $\widehat{l} \in \widehat{\mathcal{R}}(\widehat{\varsigma}')$. Hence, $\widehat{\mathcal{S}}(\widehat{\varsigma}') \subseteq \widehat{\mathcal{R}}(\widehat{\varsigma}')$, i.e. $\widehat{\varsigma}'$ is complete. ◀

▶ **Theorem 9.** *In the absence of cycles, the transition relation, $(\widehat{\rightarrow}_{\mathsf{arc}})$ is equivalent to $(\widehat{\rightarrow}_{\widehat{\Gamma}})$ in the sense that: $\forall \varsigma, \varsigma' \in \widehat{\Sigma}$, $\widehat{\varsigma} \mathrel{\widehat{\rightarrow}_{\widehat{\Gamma}}} \widehat{\varsigma}' \iff \widehat{\mathsf{rc}}(\widehat{\varsigma}) \mathrel{\widehat{\rightarrow}_{\mathsf{arc}}} \widehat{\mathsf{rc}}(\widehat{\varsigma}')$.*

Proof. It is clear that the transition rules of $(\widehat{\rightarrow}_0)$ are isomorphic to those of $(\widehat{\rightarrow})$, and that equivalence holds between both transition relations in that $\forall \widehat{\varsigma}, \widehat{\varsigma}_0 \in \widehat{\Sigma}$, $\widehat{\varsigma} \mathrel{\widehat{\rightarrow}} \widehat{\varsigma}_0 \iff \widehat{\mathsf{rc}}(\widehat{\varsigma}) \mathrel{\widehat{\rightarrow}_0} \widehat{\mathsf{rc}}(\widehat{\varsigma}_0)$. Both $(\widehat{\rightarrow}_{\mathsf{arc}})$ and $(\widehat{\rightarrow}_{\widehat{\Gamma}})$ then remove addresses from $\widehat{\sigma}$ and $\widehat{\sigma}_k$, and since both are garbage-free, they produce the same states (since $\widehat{\mathcal{S}}(\widehat{\varsigma}) = \widehat{\mathcal{R}}(\widehat{\varsigma})$ in both cases). ◀

Note that we can never have a reference from an address $\widehat{a} \in \widehat{\mathsf{Addr}}$ to an address in $\widehat{a}_k \in \widehat{\mathsf{KAddr}}$; This implies that is $\forall \widehat{a}_k \in \widehat{\mathsf{KAddr}}$, $\widehat{a}_k \in \widehat{\mathcal{R}}(\widehat{\varsigma}) \iff \widehat{a_k}_{\widehat{\varsigma}} \mathrel{\overset{*}{\widehat{\rightsquigarrow}}_{\widehat{\varsigma}}} \widehat{a}_k$. Moreover, it implies that a SCC consists either exclusively of addresses in $\widehat{\mathsf{Addr}}$ or addresses in $\widehat{\mathsf{KAddr}}$. We denote $\widehat{\pi}_k$ for the partitioning $\widehat{\pi}$ limited to $\widehat{\mathsf{KAddr}}$, i.e. $\widehat{\pi}_k = \widehat{\pi} \setminus \mathcal{P}(\widehat{\mathsf{Addr}})$.

▶ **Theorem 10.** *In a garbage-free abstract interpreter, extending abstract reference counting with cycle detection for the continuation store only requires amortized $\mathcal{O}(1)$ additional operations per continuation that is added to the continuation store.*

Proof. We use the *potential method* for amortized analysis [55]. Assume a state $\widehat{\varsigma}$ which is garbage-free, i.e. $\widehat{\mathcal{R}}(\widehat{\varsigma}) = \widehat{\mathcal{S}}(\widehat{\varsigma})$. We define its potential Φ as the amount of references between SCCs in the continuation store $\widehat{\sigma}_k$ (, i.e. $\Phi(\widehat{\varsigma}) = \sum_{\widehat{scc_k} \in \widehat{\pi}_{k\widehat{\varsigma}}} |\widehat{\phi}_{\widehat{\varsigma}}(\widehat{scc_k})|$. Cycle detection is only triggered for the continuation store on a transition $\widehat{\varsigma} \mathrel{\widehat{\rightarrow}_{0++}} \widehat{\varsigma}'$ where we insert a new continuation $\widehat{\kappa}$ into the continuation store $\widehat{\sigma_k}_{\widehat{\varsigma}}$ (in the case of λ_{ANF}, this only happens in the transition rule E-Let). Since it is clear that $\Phi(\widehat{\varsigma}) \geq 0$, we can formulate the amortized cost $\widehat{c}$ of such an insertion as $c + (\Phi(\widehat{\varsigma}') - \Phi(\widehat{\varsigma}))$, where c is the actual cost of the insertion (including cycle detection, whose cost is proportional to the traversal of the backward search). If the address $\widehat{a_k}_{\widehat{\varsigma}'}$ is fresh (i.e. $\widehat{a_k}_{\widehat{\varsigma}'} \notin \widehat{\mathcal{S}}(\widehat{\varsigma})$), then the cycle detection requires no traversal, hence $\widehat{c} = 1 + (\Phi(\widehat{\varsigma}') - \Phi(\widehat{\varsigma}))$. We have that $\Phi(\widehat{\varsigma}') = \Phi(\widehat{\varsigma}) + 1$, since the insertion adds $\widehat{a_k}_{\widehat{\varsigma}'}$ to the set $\widehat{\phi}(\mathsf{find}(\widehat{\pi}, \widehat{a_k}_{\widehat{\varsigma}}))$, which results in $\widehat{c} = 2$. If the address $\widehat{a_k}_{\widehat{\varsigma}'}$ is reused (i.e. $\widehat{a_k}_{\widehat{\varsigma}'} \in \widehat{\mathcal{S}}(\widehat{\varsigma})$), then cycle detection will perform a backward search starting from $\widehat{a_k}_{\widehat{\varsigma}'}$. The key insight is that every address $\widehat{l}$ traversed by search will lead to $\widehat{a_k}_{\widehat{\varsigma}}$, hence we are guaranteed to have a cycle. For every such $\widehat{l}$, we have that $\widehat{l} \mathrel{\overset{*}{\widehat{\rightsquigarrow}}_{\widehat{\varsigma}}} \widehat{a_k}_{\widehat{\varsigma}'}$ due to the backward search, and also that $\widehat{a_k}_{\widehat{\varsigma}} \mathrel{\overset{*}{\widehat{\rightsquigarrow}}_{\widehat{\varsigma}}} \widehat{l}$ due to the garbage-free property of $\widehat{\varsigma}$ (since $\widehat{l} \in \widehat{\mathcal{S}}(\widehat{\varsigma})$ implies $\widehat{l} \in \widehat{\mathcal{R}}(\widehat{\varsigma})$, hence it must be that $\widehat{a_k}_{\widehat{\varsigma}} \mathrel{\overset{*}{\widehat{\rightsquigarrow}}_{\widehat{\varsigma}}} \widehat{l}$). As a result, all references that are traversed will become part of the same SCC, and therefore be removed from $\widehat{\phi}$. Hence, if cycle detection traverses k references, we have that $c = k + 1$ (due to the traversal of k references) and $\Phi(\widehat{\varsigma}') = \Phi(\widehat{\varsigma}) - k$ (due to the removal of k references), which results in $\widehat{c} = c + (\Phi(\widehat{\varsigma}') - \Phi(\widehat{\varsigma})) = 1$. In both cases, the insertion only requires amortized $\mathcal{O}(1)$ operations. ◀

B Supplementary Definitions for $(\widehat{\rightarrow}_{\mathsf{arc++}})$

Figure 12 shows the updated auxiliary transition relation $(\widehat{\rightarrow}_{\mathsf{0++}})$.

$$\frac{\begin{array}{c}\widehat{a} = \widehat{\mathsf{alloc}}(x, \widehat{\varsigma}) \qquad \widehat{a}'_k = \widehat{\mathsf{alloc}}_k(e_1, \widehat{\varsigma}) \qquad \widehat{\rho}' = \widehat{\rho}[x \mapsto \widehat{a}] \\ \widehat{\kappa} = \langle \widehat{a}, e_2, \widehat{\rho}', \widehat{a}_k \rangle \qquad \langle \widehat{\phi}', \widehat{\pi}' \rangle = \textsc{extend}(\widehat{a}'_k, \widehat{\mathcal{T}}_{\widehat{\mathsf{Kont}}}(\widehat{\kappa}), \widehat{\phi}, \widehat{\pi})\end{array}}{\underbrace{\langle \mathsf{let}\; x = e_1 \;\mathsf{in}\; e_2, \widehat{\rho}, \widehat{\sigma}, \widehat{\sigma}_k, \widehat{a}_k, \widehat{\phi}, \widehat{\pi} \rangle}_{\hat{\varsigma}} \;\widehat{\rightarrow}_{\mathsf{0++}}\; \langle e_1, \widehat{\rho}, \widehat{\sigma}, \widehat{\sigma}_k \sqcup [\widehat{a}'_k \mapsto \widehat{\kappa}], \widehat{a}'_k, \widehat{\phi}', \widehat{\pi}' \rangle} \quad (\text{E-Let})$$

$$\frac{\begin{array}{c}\widehat{a} = \widehat{\mathsf{alloc}}(x, \widehat{\varsigma}) \qquad \widehat{\mathcal{A}}(f, \widehat{\rho}, \widehat{\sigma}) \ni \langle \lambda x.e', \widehat{\rho}' \rangle \\ \widehat{\mathcal{A}}(ae, \widehat{\rho}, \widehat{\sigma}) = \widehat{v} \qquad \langle \widehat{\phi}', \widehat{\pi}' \rangle = \textsc{extend}(\widehat{a}, \widehat{\mathcal{T}}_{\mathcal{P}(\widehat{\mathsf{Clo}})}(\widehat{v}), \widehat{\phi}, \widehat{\pi})\end{array}}{\underbrace{\langle f\; ae, \widehat{\rho}, \widehat{\sigma}, \widehat{\sigma}_k, \widehat{a}_k, \widehat{\phi}, \widehat{\pi} \rangle}_{\hat{\varsigma}} \;\widehat{\rightarrow}_{\mathsf{0++}}\; \langle e', \widehat{\rho}'[x \mapsto \widehat{a}], \widehat{\sigma} \sqcup [\widehat{a} \mapsto \widehat{v}], \widehat{\sigma}_k, \widehat{a}_k, \widehat{\phi}', \widehat{\pi}' \rangle} \quad (\text{E-Call})$$

$$\frac{\begin{array}{c}\widehat{\mathcal{A}}(ae, \widehat{\rho}, \widehat{\sigma}) = \widehat{v} \qquad \widehat{\sigma}_k(\widehat{a}_k) \ni \langle \widehat{a}, e', \widehat{\rho}', \widehat{a}'_k \rangle \\ \langle \widehat{\phi}', \widehat{\pi}' \rangle = \textsc{extend}(\widehat{a}, \widehat{\mathcal{T}}_{\mathcal{P}(\widehat{\mathsf{Clo}})}(\widehat{v}), \widehat{\phi}, \widehat{\pi})\end{array}}{\langle ae, \widehat{\rho}, \widehat{\sigma}, \widehat{\sigma}_k, \widehat{a}_k, \widehat{\phi}, \widehat{\pi} \rangle \;\widehat{\rightarrow}_{\mathsf{0++}}\; \langle e', \widehat{\rho}', \widehat{\sigma} \sqcup [\widehat{a} \mapsto \widehat{v}], \widehat{\sigma}_k, \widehat{a}'_k, \widehat{\phi}', \widehat{\pi}' \rangle} \quad (\text{E-Return})$$

Figure 12 Auxiliary transition relation using abstract reference counting with cycle detection.

The updated definition for $\widehat{\mathsf{collect}}$ is given below. We slightly abuse notation for set removal here: for a partitioning $\widehat{\pi}$, $\widehat{\pi} \setminus S$ really means $(\widehat{\pi} \setminus S) \cup \bigcup_{s \in S} \bigcup_{e \in s} \{\{e\}\}$, while for a store $\widehat{\sigma}$, $\widehat{\sigma} \setminus S$ means $\widehat{\sigma} \setminus \bigcup_{s \in S} s$ (and analogous for $\widehat{\sigma}_k$).

$$\widehat{\mathsf{collect}}(\widehat{\varsigma}, \widehat{\varsigma}') = \langle e_{\hat{\varsigma}'}, \widehat{\rho}_{\hat{\varsigma}'}, \widehat{\sigma}_{\hat{\varsigma}'} \setminus \widehat{G},\, \widehat{\sigma_{k}}_{\hat{\varsigma}'} \setminus \widehat{G},\, \widehat{a_{k}}_{\hat{\varsigma}'}, \widehat{\phi}', \widehat{\pi}_{\hat{\varsigma}'} \setminus \widehat{G} \rangle$$

$$\text{where } \langle \widehat{\phi}', \widehat{G} \rangle = \widehat{\mathsf{check*}}(\widehat{C}_0, \emptyset, \widehat{\phi}_{\hat{\varsigma}'}) \; \widehat{C}_0 = \{\mathsf{find}(\widehat{\pi}_{\hat{\varsigma}'}, \widehat{l}) \mid \widehat{l} \in \widehat{\mathcal{T}}_{\widehat{\Sigma}}(\widehat{\varsigma})\}$$

$$\widehat{\mathsf{check*}}(\widehat{C}, \widehat{G}, \widehat{\phi}) = \begin{cases} \langle \widehat{\phi}, \widehat{G} \rangle & \text{if } \widehat{C} = \emptyset \\ \widehat{\mathsf{check}}(\widehat{scc}, \widehat{C} \setminus \{\widehat{scc}\}, \widehat{G}, \widehat{\phi}) & \text{otherwise, where } \widehat{scc} \in \widehat{C} \end{cases}$$

$$\widehat{\mathsf{check}}(\widehat{scc}, \widehat{C}, \widehat{G}, \widehat{\phi}) = \begin{cases} \widehat{\mathsf{dealloc}}(\widehat{scc}, \widehat{C}, \widehat{G}, \widehat{\phi}) & \text{if } |\widehat{\phi}(\widehat{scc})| = 0 \wedge \widehat{scc} \notin \widehat{R} \\ \widehat{\mathsf{check*}}(\widehat{C}, \widehat{G}, \widehat{\phi}) & \text{otherwise} \end{cases}$$

$$\text{where } \widehat{R} = \{\mathsf{find}(\widehat{\pi}_{\hat{\varsigma}'}, \widehat{l}) \mid \widehat{l} \in \widehat{\mathcal{T}}_{\widehat{\Sigma}}(\widehat{\varsigma}')\}$$

$$\widehat{\mathsf{dealloc}}(\widehat{scc}, \widehat{C}, \widehat{G}, \widehat{\phi}) = \widehat{\mathsf{check*}}(\widehat{C} \cup \widehat{S}, \widehat{G} \cup \{\widehat{scc}\}, \widehat{\phi} \ominus \bigsqcup_{\widehat{scc}' \in \widehat{S}} [\widehat{scc}' \mapsto \widehat{scc}])$$

$$\text{where } \widehat{S} = \{\mathsf{find}(\widehat{\pi}_{\hat{\varsigma}'}, \widehat{l}') \mid \exists \widehat{l} \in \widehat{scc} \wedge \widehat{l} \;\widehat{\rightsquigarrow}_{\hat{\varsigma}'}\; \widehat{l}' \wedge \widehat{l}' \notin \widehat{scc}\}$$

C Supplementary Code Listings

The following code is used to generate Figure 1, and is taken from [1]. It is also used for the collatz benchmark in Section 5.

```
(define (div2* n s)
  (if (= (* 2 n) s)
      n
      (if (= (+ (* 2 n) 1) s)
          n
          (div2* (- n 1) s))))
(define (div2 n)
  (div2* n n))
(define (hailstone* n count)
  (if (= n 1)
      count
      (if (even? n)
          (hailstone* (div2 n) (+ count 1))
          (hailstone* (+ (* 3 n) 1) (+ count 1)))))
(define (hailstone n)
  (hailstone* n 0))
(hailstone 5)
```

Eventually Sound Points-To Analysis with Specifications

Osbert Bastani
University of Pennsylvania, Philadelphia, USA
obastani@seas.upenn.edu

Rahul Sharma
Microsoft Research, Bangalore, India
rahsha@microsoft.com

Lazaro Clapp
Stanford University, USA
lazaro@stanford.edu

Saswat Anand
Stanford University, USA
saswat@cs.stanford.edu

Alex Aiken
Stanford University, USA
aiken@cs.stanford.edu

Abstract

Static analyses make the increasingly tenuous assumption that all source code is available for analysis; for example, large libraries often call into native code that cannot be analyzed. We propose a points-to analysis that initially makes optimistic assumptions about missing code, and then inserts runtime checks that report counterexamples to these assumptions that occur during execution. Our approach guarantees *eventual* soundness, which combines two guarantees: (i) the runtime checks are guaranteed to catch the first counterexample that occurs during any execution, in which case execution can be terminated to prevent harm, and (ii) only finitely many counterexamples ever occur, implying that the static analysis eventually becomes statically sound with respect to all remaining executions. We implement **Optix**, an eventually sound points-to analysis for Android apps, where the Android framework is missing. We show that the runtime checks added by **Optix** incur low overhead on real programs, and demonstrate how **Optix** improves a client information flow analysis for detecting Android malware.

2012 ACM Subject Classification Theory of computation → Program analysis

Keywords and phrases specification inference, static points-to analysis, runtime monitoring

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2019.11

Acknowledgements This work was supported by NSF grant CCF-1160904, and is also based on research sponsored by the Air Force Research Laboratory, under agreement number FA8750-12-2-0020. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

1 Introduction

To guarantee soundness, static analyses often assume that all program source code can be analyzed. This assumption has become tenuous as programs increasingly depend on large libraries and frameworks that are prohibitively difficult to analyze [34]. For example, mobile app stores can use static analysis to improve the quality of published apps by searching for malicious behaviors [23, 21, 5, 25] or security vulnerabilities [20, 39, 17]. However, Android apps depend extensively on the Android framework, which makes frequent use of

33rd European Conference on Object-Oriented Programming (ECOOP 2019).
Editor: Alastair F. Donaldson; Article No. 11; pp. 11:1–11:28
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

native code and reflection, both of which are practical barriers to static analysis and are often ignored by the analysis [5, 21, 25]. Furthermore, the Android framework contains deep call hierarchies, which can pose problems since static points-to analyses typically have limited context sensitivity [9]. Thus, the Android framework is often omitted from the static analysis [52, 8, 12]. We refer to such omitted code as *missing*. In any large software system, there are inevitably parts that are missing and cannot be handled soundly [34].

Two possible approaches are to sacrifice either soundness (by making optimistic assumptions about missing code) or precision (by making pessimistic assumptions about missing code). For many applications, pessimistic assumptions are so imprecise that they make the static analysis results unusable; therefore, soundness is often sacrificed instead, but doing so is a significant compromise [34]. For example, consider malware detection – a security analyst must examine every app that is potentially malicious, making false positives costly, yet unsoundness can be exploited by a knowledgeable attacker to avoid detection.

Broadly speaking, two alternative strategies have been proposed to handle missing code: (i) using runtime instrumentation, and (ii) using handwritten specifications. The first approach uses runtime instrumentation to enforce soundness – e.g., dynamic information flow control can be used to prevent information leaks [6, 14, 18]. More recent systems have focused on using dynamic analysis to fill gaps in a static analysis. These systems first statically analyze the program, keeping track of any unsound assumptions made by the static analysis, and then instrument the program to check if these assumptions are violated during runtime; if so, then execution can be terminated or continued in a safe environment such as a sandbox. This approach has been applied to type checking [22], reflective call targets [10], and determining reachable code [7]; more general systems have also been proposed [11, 15]. However, the runtime instrumentation used by these systems is typically limited – e.g., they check either lightweight type-based properties, or program invariants that can be checked locally.

In the second approach, the human user writes *specifications* (also known as *models*, *annotations*, or *stubs*) that summarize missing code [52, 8]. This approach is very flexible, since specifications can be used to model arbitrary missing code – e.g., systems have used specifications to model part [5, 25] or all [21] of the Android framework, including production systems [19]. However, specifications are costly to write, since many thousands of specifications may be needed [8]. In addition, handwritten specifications can be error prone [28], and must be updated whenever the missing code changes. Approaches have been proposed for *inferring* specifications [52, 8, 12, 28, 9], but the user must manually check each inferred specification, including ones that turn out to be wrong or irrelevant.

In this paper, we study the problem of handling unsoundness in a static points-to analysis for Android apps, where part or all of the Android framework is omitted from the analysis. We focus on points-to analysis since it lies at the core of many static analyses, and we believe that clients (e.g., static information flow analysis) can be designed around our analysis.

We propose a system that handles missing code by combining runtime monitoring and specifications. Given a program (e.g., an Android app), our system first runs the static points-to analysis that optimistically assumes missing code is empty. If no errors are found, then our system instruments the program to detect counterexamples to the optimistic assumptions – i.e., *missing points-to edges* that occur during an execution but are missing from the (optimistic) static analysis. Points-to edges are a whole-program property, so instrumenting programs to detect missing points-to edges is challenging. In particular:

- Naïvely using a dynamic points-to analysis to detecting counterexamples can incur huge overhead – e.g., as much as 20× [12] or even two orders of magnitude [38].
- It is often impossible to insert runtime checks into missing code (e.g., native code). Thus, we restrict our analysis to instrument only *available* code (in our case, the app code).

We describe how we address these challenges in more detail below. Next, the instrumented program is published (e.g., on Google Play). If our instrumentation ever detects a counterexample, then it is reported back to the publisher (e.g., Google), who can update their specifications and re-run the static analysis. Finally, we show how detected counterexamples can be used to infer specifications that summarize missing code – intuitively, these specifications transfer the knowledge gained from the counterexample to benefit the analysis of future programs. While inferred specifications need to be validated by a human, our approach focuses specification inference on addressing gaps in the static analysis that occur in actual program executions. Importantly, adding new specifications (either handwritten or inferred) can help reduce instrumentation overhead.

With an appropriate instrumentation scheme, our system satisfies two key properties:

- **Eventual soundness:** As soon as a counterexample occurs during execution, it is detected by the program instrumentation and reported to the static analysis. Furthermore, only finitely many counterexamples are ever reported.
- **Precision:** The analysis is at least as precise as having all specifications available.

The key property of interest is eventual soundness, which combines two guarantees. The first guarantee is a *dynamic soundness* guarantee analogous to that provided by dynamic type checking: at the cost of some runtime overhead, we guarantee that unsoundness is detected at the point when it occurs, which allows us to prevent damage (e.g., leaking of sensitive information) from occuring.

The second guarantee is that with every reported counterexample, the specifications become progressively more complete – in particular, the specifications guarantee that the static analysis is sound with respect to all executions observed so far. More precisely, suppose that a counterexample is reported for an execution e. Based on this information, the static analysis either discovers a bug (e.g., an information leak), in which case the program is repaired or removed, or concludes that the program is safe even with the updated specifications. In the latter case, for any subsequent execution identical to e, no counterexamples will be reported since the static analysis is now sound with respect to all behaviors exhibited in e and has concluded that all of these behaviors are safe.

Furthermore, note that we guarantee that only finitely many counterexamples are ever reported (this observation is a simple consequence of the fact that even in the worst case, there are only a finite number of potential counterexamples). Thus, the static analysis eventually becomes statically sound with respect to all subsequent executions. Even though we cannot detect when convergence is reached, it suggests that in practice, fewer and fewer counterexamples are reported over time. Indeed, we empirically observe this trend in our evaluation. Even if the static analysis itself never fully converges, the dynamic soundness guarantee ensures that the overall system is sound.

As described above, our key contribution is an instrumentation scheme for detecting counterexamples that ensures eventual soundness. To address the challenge of high runtime overhead, we leverage the fact that to be eventually sound, we do not need the program instrumentation to report *every* counterexample that occurs during an execution. Instead, it is sufficient that for any execution, the instrumentation detects the *first* counterexample to occur. For example, let $x \hookrightarrow o$ and $y \hookrightarrow o$ be two *potentially missing* points-to edges; if we can guarantee that for any execution, $x \hookrightarrow o$ can only occur after $y \hookrightarrow o$ has already occured (and we are furthermore guaranteed to detect that $y \hookrightarrow o$ has occured before $x \hookrightarrow o$ can occur), then we only need to monitor whether $y \hookrightarrow o$ occurs. By leveraging this property, we substantially reduce the amount of required instrumentation. For programs where instrumentation in performance-critical parts is required, the overhead can be further reduced by manually adding specifications summarizing the relevant missing code.

For the challenge of being unable to instrument missing code, note that because we use specifications, we are already unable to discover relationships about the missing code. Indeed, for many clients, only relationships between variables in the available code are of interest – e.g., Android malware can be characterized by relationships between variables in the app code alone [21]. However, these relationships typically depend on relationships between variables in the missing code. For points-to analysis, we cannot observe when variables in the app might be aliased because they both point to the same object allocated in missing code. To address this issue, our analysis introduces *proxy objects* that correspond to concrete objects allocated in missing code,[1] which enable us to soundly and precisely compute client relations that refer only to available code (e.g., aliasing and concrete types).

We implement our eventually sound points-to analysis in a tool called **Optix**[2], which analyzes Android apps treating the entire Android framework as missing. We show that our instrumentation typically incurs low overhead – the median overhead is 4.6%, the overhead is less than 20% for more than 90% of apps in our benchmark, and the highest is about 50%. The overhead of the outliers can be reduced as described above; in particular, only a few manually provided specifications are needed to reduce the overhead of the outliers to reasonable levels (see Section 8.1). Also, we show that the instrumentation can be used to detect missing points-to specifications in the information flow client from [21], which computes explicit information flows [42]. This tool uses specifications to model missing code (i.e., the Android framework). We empirically show that we can detect missing specifications that are relevant to the information flow client. In summary, our contributions are:

- We propose an eventually sound points-to analysis for programs with calls to missing code that is also precise and automatic (Section 3). In particular, our analysis adds runtime instrumentation in the available code that detects and reports counterexamples, and can guarantee that soundness is never compromised (i.e., malicious functionality never gets executed) by terminating execution as soon as a counterexample is detected.
- We minimize instrumentation to reduce runtime overhead (Section 3) and introduce proxy objects to handle allocations in missing code (Section 4).
- We implement **Optix**, a points-to analysis for Android apps that treats the entire Android framework as missing.
- We show that the instrumentation overhead is manageable (Section 8), and that **Optix** can detect missing specifications relevant to the explicit information flow client from [21]. The largest app in our benchmark has over 300K lines of Jimple code.

2 Overview

Consider the program in Figure 1. Suppose that a security analyst asks whether the program leaks the return value of `mkStr` to the Internet via a call to `sendHttp`, which requires knowing that `str` and `dataCopy` may be aliased. We use points-to analysis to determine which variables may be aliased. In particular, a points-to analysis computes *points-to edge* $x \hookrightarrow o$ if variable x may point to a concrete object $\bar{o}$ allocated at allocation statement $o \in \mathcal{O}$ (called an *abstract object*) during execution. Two variables may be aliased if they may point to the same abstract object. Our example program exhibits points-to edges such as `list` $\hookrightarrow o_{\text{list}}$, `str` $\hookrightarrow o_{\text{str}}$, and `dataCopy` $\hookrightarrow o_{\text{str}}$, so the points-to analysis concludes that `str` and `dataCopy` may be aliased.

[1] The term *proxy object* is ours, but the concept has occurred in prior work [8].
[2] **Optix** stands for Optimistic Points-To Information from eXecutions.

```
void main() { // program             String mkStr() { // library
  String str = mkStr();                String libStr = new String(); // o_str
  List list = new List(); // o_list    return libStr; }
  list.add(str);                     void sendHttp(String str) { ... } // library
  Object data = list.get(0);         class List { // library
  if(randBool()) {                     Object f;
    Object dataCopy = data;            void add(Object ob) { f = ob; }
    sendHttp(dataCopy); }}             Object get(int i) { return f; } }
```

Figure 1 Program `main` (left) calls various library functions, for which the analyst provides specifications (right). Abstract objects o_{list} and o_{str} are labeled in comments.

Suppose that the library code is missing. For example, static analyses for Android apps often have difficulty analyzing Android framework code. In particular, the framework code makes substantial use of native code and reflection, which are too difficult to analyze, and are thus unsoundly ignored by most state-of-the-art static analyses [52, 5, 8], and because it uses deep call hierarchies, which can cause significant imprecision, since static points-to analyses often have limited context sensitivity [9]. Thus, the Android framework is missing from the perspective of the static analyses.

For many clients (including static information flow analysis), it suffices to compute edges for *visible* variables $x \in \mathcal{V}_P$ in the available code; however, these edges often depend on relationships in the missing code. Pessimistically assuming that missing code can be arbitrary is very imprecise, e.g., we may have `data` $\hookrightarrow o_{\text{list}}$ in case the implementation of `get` is `return this`. Alternatively, optimistically assuming that missing code is empty can be unsound, for example, failing to compute `data` $\hookrightarrow o_{\text{str}}$ and `dataCopy` $\hookrightarrow o_{\text{str}}$. Such *dynamic* points-to edges that are not computed statically are *missing*.

A typical approach in practice is to provide *specifications*, which are code fragments that overapproximate the points-to behaviors of library functions; see Figure 1 for examples. For instance, because our static points-to analysis collapses arrays into a single field, we can overapproximate the array of elements stored by the `List` class as a single field `f`.

Suppose that the analyst has provided specifications for frequently used library functions such as `mkStr` and `sendHttp`, but a long tail of specifications remain missing, including those for `add` and `get`. Therefore, the (optimistic) static information flow analysis incorrectly concludes that `dataCopy` cannot point to `str`, and that `mkStr` therefore does not leak to the Internet. Furthermore, dynamic information flow control cannot be applied since the missing code cannot be instrumented without modifying every end user's Android installation.

Our analysis instruments the Android app to detect whether *counterexamples* to the optimistic assumption that every missing specification is empty; this instrumentation only inserts runtime checks in the available code. The instrumented app is published on Google Play. If the instrumentation observes that a counterexample occurs during an execution, then the counterexample is reported back to Google Play, which recomputes the static analysis to account for this new information. Our example program `main` is instrumented to record the concrete objects pointed to by `libStr` and `data`. When the program is run: (i) `libStr` points to concrete object $\bar{o}_{\text{str}}$, so our analysis concludes that $\bar{o}_{\text{str}}$ is allocated at o_{str}, and (ii) `data` points to $\bar{o}_{\text{str}}$, so our analysis concludes that `data` $\hookrightarrow o_{\text{str}}$ and reports this counterexample. Upon receiving this report, we add `data` $\hookrightarrow o_{\text{str}}$ to the known counterexamples.

Given a new counterexample $x \hookrightarrow o$, the static analysis at the very least learns that $x \hookrightarrow o$ is a points-to edge that may occur. There are two ways in which the static analysis can generalize from this fact. First, it can compute additional missing points-to edges that are consequences of this fact according to the rules of the static analysis. For example, given the counterexample `data` $\hookrightarrow o_{\text{str}}$, our static analysis additionally computes its consequence `dataCopy` $\hookrightarrow o_{\text{str}}$, and determines that `str` and `dataCopy` may be aliased. Thus, the security analyst learns that the return value of `mkStr` may leak to the Internet, and can report any newly discovered bugs to the developer. In this case, the leak is discovered even if `randBool` returns false and the data is not leaked in that specific execution.

Second, the static analysis can use *specification inference* to try to identify which missing specification may have been the "cause" of the missing points-to edge. By doing so, the static analysis generalizes the counterexample to eliminate unsoundness when analyzing future apps. In Section 5, we show how our tool leverages specification inference to automatically infer candidate specifications that "explain" the counterexample. For example, given counterexample `data` $\hookrightarrow o_{\text{str}}$, the specification inference algorithm would infer the specifications for `add` and `get` shown in Figure 1. One caveat is that the inferred specifications must be validated by a human, since it is impossible to guarantee that they are correct. We show that in practice, the inference algorithm has high accuracy.

Next, we describe how our analysis instruments apps to detect missing points-to edges. Naïvely, we could use a dynamic points-to analysis, which instruments every allocation, assignment, load, and store operation in the program to determine all of the dynamic points-to edges that occur during an execution. However, this approach requires far more instrumentation than necessary. In particular, suppose that multiple counterexamples occur during an execution; to be eventually sound, the instrumentation only has to detect the first one that occurs during execution. Leveraging this property enables us to substantially reduce the required instrumentation. For example, note that the missing points-to edge `dataCopy` $\hookrightarrow o_{\text{str}}$ can only occur during execution where the missing points-to edge `data` $\hookrightarrow o_{\text{str}}$ has already occured. Furthermore, once `data` $\hookrightarrow o_{\text{str}}$ has occured, it is added to the static analysis, which computes `dataCopy` $\hookrightarrow o_{\text{str}}$ as a consequence. Therefore, we never need to detect or report `dataCopy` $\hookrightarrow o_{\text{str}}$.

Another challenge with the instrumentation is how to handle allocations in missing code. For example, if the specification for `mkStr` were also missing, then our analysis cannot instrument `libStr` to determine that $\bar{o}_{\text{str}}$ was allocated at o_{str}. Nevertheless, we can reason about such missing abstract objects based on observations in available code. In particular, suppose we instrument `str` and `list`. During execution, this instrumentation detects that `str` points to a concrete object $\bar{o}_{\text{str}}$. Since $\bar{o}_{\text{str}}$ was not allocated at o_{list}, it must have been allocated in `mkStr`. We represent this fact by introducing a *proxy object* p_{mkStr} pointed to by the return value $r_{\texttt{mkStr}}$ of `mkStr`. We discuss proxy objects in Section 4.

Finally, we propose an eventually sound points-to analysis. Future work is needed to design an eventually sound information flow analysis; we describe a candidate in Section 9, but implementing and evaluating this analysis is beyond the scope of our work.

3 Eventually Sound Points-To Analysis

We describe our eventually sound points-to analysis, summarized in Figure 3.

3.1 Background and Assumptions

Consider a program P (whose code is available) containing calls to functions in a library L (whose code is missing). There are five kinds of statements: allocations ($x \leftarrow X()$, where $X \in \mathcal{C}$ is a class), assignments ($x \leftarrow y$, where $x, y \in \mathcal{V}_P$ are program variables), loads ($x \leftarrow y.f$, where $f \in \mathcal{F}$ is a field), stores ($x.f \leftarrow y$), and calls to library functions

$$\textbf{1.}\ \text{(allocation)}\ \frac{x \leftarrow X(),\ o = (x \leftarrow X())}{x \hookrightarrow o \in \Pi}$$

$$\textbf{2.}\ \text{(assignment)}\ \frac{x \leftarrow y,\ y \hookrightarrow o \in \Pi}{x \hookrightarrow o \in \Pi}$$

$$\textbf{3.}\ \text{store}\ \frac{x.f \leftarrow y,\ x \hookrightarrow o' \in \Pi,\ y \hookrightarrow o \in \Pi}{o'.f \hookrightarrow o \in \Pi_O}$$

$$\textbf{4.}\ \text{load}\ \frac{x \leftarrow y.f,\ y \hookrightarrow o' \in \Pi, o'.f \hookrightarrow o \in \Pi_O}{x \hookrightarrow o \in \Pi}$$

$$\textbf{5.}\ \text{(missing)}\ \frac{x \hookrightarrow o \in \Pi_{\text{miss}}}{x \hookrightarrow o \in \Pi}$$

Figure 2 Rules to compute sound points-to sets. Rules 1-4 are standard. Rule 5 adds reported counterexamples to the analysis.

$m \in \mathcal{M}$ library ($x \leftarrow m(y)$). We omit control flow statements since our static analysis is flow-insensitive. We let p_m (resp., r_m) denote the parameter (resp., return value) of library function m. For convenience, we assume that each library function has exactly one argument, and that there are no functions in P.

Our static may points-to analysis, shown in Figure 2, is a standard flow- and context-insensitive analysis for computing points-to edges $\Pi \subseteq \mathcal{V}_P \times \mathcal{O}$ [3, 46, 8]; we describe how our results can be extended to context- and object-sensitive analyses with on-the-fly callgraph construction in Section 6.4. Rule 1 handles object allocations $x \leftarrow X()$. In particular, recall that an abstract object is an allocation statement $o = (x \leftarrow X())$, representing all concrete objects allocated at that statement. Then, the rule says that x may point to o. Rule 2 handles assignments $x \leftarrow y$ – i.e., if y may point to o, then x may point to o as well. Rules 3-4 handle program loads and stores. They introduce a new relationship $\Pi_O \subseteq \mathcal{O} \times \mathcal{F} \times \mathcal{O}$, which denotes points-to relationships between abstract objects. Intuitively, the relationship $o'.f \hookrightarrow o \in \Pi_O$ (written $o'.f$ *may point to* o) means that a field f of abstract object o' may reference an abstract object o. Rule 3 handles stores – given a statement $x.f \leftarrow y$, it says that if y may point to o and x may point to o', then $o'.f$ may point to o. Rule 4 handles loads – given a statement $x \leftarrow y.f$, it says that if y may point to o' and $o'.f$ may point to o, then x may point to o. Together, rules 3 and 4 are equivalent to the single rule

$$\frac{x \leftarrow y.f,\ z.f \leftarrow w,\ y \hookrightarrow o' \in \Pi,\ z \hookrightarrow o' \in \Pi,\ w \hookrightarrow o \in \Pi}{x \hookrightarrow o \in \Pi}.$$

Rule 5 handles known counterexamples $\Pi_{\text{miss}} \subseteq \mathcal{V}_P \times \mathcal{O}$. A function call $x \leftarrow m(y)$ is treated as an assignment of y to the parameter p_m and an assignment of the return value r_m to x.

We initially make three simplifying assumptions. First, we assume library functions do not contain allocations; we remove this assumption in Section 4. Second, we make the *disjoint fields assumption*, which says that $\mathcal{F}_L \cap \mathcal{F}_P = \emptyset$, where $\mathcal{F}_L$ (resp., $\mathcal{F}_P$) are fields accessed by load and store statements in the library (resp., program), i.e., there are no *shared fields* $f \in \mathcal{F}_L \cap \mathcal{F}_P$. [3] We discuss how to weaken this assumption in Section 6.1. Third, the programs we consider do not have callbacks; we discuss how to handle callbacks in Section 6.2. Finally, **Optix** assumes that library functions do not access global variables; we describe how we could remove this assumption in Section 9.

[3] In practice, a shared field is typically a public field in the library code that is accessed by the program.

3.2 Eventual Soundness

We first define soundness relative to an execution:

▶ **Definition 1.** Let Π be a points-to set. We say an execution e *exhibits a counterexample* with respect to Π if during the execution, there is a dynamic points-to edge $x \hookrightarrow o \notin \Pi$. We say Π is *sound* with respect to e if e does not exhibit any counterexamples.

Consider a points-to analysis that for a sequence of instrumented executions $e_1, e_2, \ldots$ computes a sequence of points-to sets $\Pi_1, \Pi_2, \ldots$, both indexed by the natural numbers $i \in \mathbb{N}$. Here, Π_i is computed as a function of the previous points-to set Π_{i-1} and the counterexamples from e_i (if any). Note that the instrumentation for e_{i+1} can be chosen adaptively based on Π_i and that $\Pi_i \subseteq \Pi_j$ if $i \leq j$.

▶ **Definition 2.** The points-to analysis is *eventually sound* if for any sequence $e_1, e_2, \ldots$ of executions, (i) for any execution e_i, the instrumentation detects and reports the first counterexample (if any) that occurs during the execution, and (ii) there are only finitely many $i \in \mathbb{N}$ such that the execution e_i exhibits a counterexample with respect to the previous points-to set Π_{i-1}.

Intuitively, the first property says that counterexamples are detected as soon as they occur (which ensures that any bugs or malicious behaviors are detected as soon as they occur). The second property implies that the points-to sets eventually become sound with respect to all subsequent executions – more precisely, there exists $n \in \mathbb{N}$ such that the points-to set Π_n is sound with respect to all executions e_i such that $n \leq i < \infty$.[4]

We remark that we do *not* require that the different executions use the same inputs, random seeds, or have the same execution time. We can also handle programs that run continuously – we simply split the execution into intervals (e.g., 1st hour, 2nd hour, ...) and treat each interval as a different execution. However, one of our optimizations may need to be disabled, since it requires that the instrumentation be modified over time, which may not be possible for a continuously running program.

▶ **Definition 3.** The points-to analysis is *precise* if for every $i \in \mathbb{N}$, the points-to set Π_i is a subset of the points-to set computed by analyzing the implementation of the missing code.

Note that while progress towards static soundness is guaranteed, it is not possible to report how many sources of static unsoundness remain at any point in time. Even if all program paths are executed, there may be missing points-to edges – e.g., in the following, suppose that `foo` is missing; then, if `randInt` never evaluates to `0`, `y` $\hookrightarrow o$ remains missing:

```
void main() { // program
  Object x = new Object(); // o
  Object y = foo(x); }
```

```
Object foo(Object ob) { // library
  Object[] arr = new Object[2];
  arr[0] = ob;
  return arr[randInt()]; }
```

Despite the inability to quantify progress, the property of eventual soundness is useful, since (in addition to guaranteeing dynamic soundness) it guarantees that only a finite number of counterexamples can possibly occur. In particular, this property implies that the number of counterexamples reported must decrease over time (eventually to zero). For example, suppose we try to construct an eventually sound interval analysis for a program $x \leftarrow m()$

[4] Note the upper bound $i < \infty$; this property only holds for executions indexed by the natural numbers $\mathbb{N}$.

with an integral variable x by abstracting a set of counterexamples with the smallest interval that contains all the counterexamples. Such an analysis is *not* eventually sound. On the other hand, an analysis that abstracts counterexamples with $(-\infty, \infty)$ is sound and therefore (vacuously) eventually sound. Finally, the former analysis is eventually sound (but not precise) if after n counterexamples, the analysis outputs $(-\infty, \infty)$.

Also, it is permissible for a counterexample to simply never occur in any execution – e.g., in the above code, if the call to `randInt` in `foo` always returns `1`, then the counterexample $\texttt{y} \hookrightarrow o$ does not occur in any execution. Eventual soundness is still satisfied, since it is defined relative to the sequence of observed executions – if a counterexample exists but is never observed, then the analysis is still statically sound for all observed executions.

3.3 Naïve Algorithm

We first describe a naïve eventually sound points-to analysis.

Optimistic analysis. We use the static analysis in Figure 2 to compute static points-to edges Π, assuming that calls to library functions are no-ops – in particular, the set of counterexamples is initially empty, i.e., $\Pi_{\text{miss}} \leftarrow \emptyset$.

Runtime checks. A *monitor* is instrumentation added to a statement $x \leftarrow *$ (where $*$ stands for any valid subexpression). After executing this statement, the monitor issues a *report* $(x \leftarrow *, \bar{o})$ – i.e., it records the value of the concrete object $\bar{o}$ pointed to by x after executing $x \leftarrow *$. Note that upon executing, a monitor only records a single integer – i.e., the memory address pointed to by x; it does not traverse the heap more deeply. Thus, a monitor is very lightweight instrumentation. A *monitoring scheme* M is a set of program statements to be monitored. Our goal is to design monitoring schemes that satisfy the following:

▶ **Definition 4.** We say a monitoring scheme M is *sound* if for any execution, M reports the first counterexamples that occurs (if any), and we say M is *precise* if it only reports counterexamples, i.e., it does not report false positives.

Naïvely, it is sound and precise to monitor every variable $x \in \mathcal{V}_P$. Then, we can map each concrete object $\bar{o}$ to its allocation:

▶ **Definition 5.** An *abstract object mapping* for an execution is a mapping $\bar{o} \rightsquigarrow o$, where $\bar{o}$ is a concrete object allocated at abstract object o.

For every report $(x \leftarrow X(), \bar{o})$, we add $\bar{o} \rightsquigarrow o$ to the abstract object mapping, where $o = (x \leftarrow X())$. Then, for every report $(x \leftarrow *, \bar{o})$ and $\bar{o} \rightsquigarrow o$, we conclude that $x \hookrightarrow o$ occurred dynamically; if missing, we report it as a counterexample. In our example, we monitor `libStr`, detect that $\bar{o}_{\text{str}} \rightsquigarrow o_{\text{str}}$, and report counterexample $\texttt{data} \hookrightarrow \bar{o}_{\text{str}}$.

Updating the static analysis. We add every reported counterexample to Π_{miss}. Our static analysis in Figure 2 adds Π_{miss} to Π and computes the consequences of these added edges. Continuing our example, our static analysis adds $\texttt{data} \hookrightarrow o_{\text{str}}$ to Π (rule 5), and computes its consequence $\texttt{dataCopy} \hookrightarrow o_{\text{str}}$ (rule 2).

Guarantees. Let Π^* be the points-to edges computed in the case that $\Pi_{\text{miss}} = \Pi^*_{\text{miss}}$ holds, where Π^*_{miss} is the set of all missing points-to edges. Then, our analysis is:

Data Structure	Rules for Construction
monitors	(allocation) $M_{\text{alloc}} = \mathcal{O}_P$ (function call) $\dfrac{x \leftarrow m(y)}{(x \leftarrow m(y)) \in M_{\text{call}}}$
reports	(allocation) $\dfrac{x \leftarrow X(),\ x \hookrightarrow \bar{o}}{(x \leftarrow X(), \bar{o}) \in R_{\text{alloc}}}$ (function call) $\dfrac{x \leftarrow m(y),\ x \hookrightarrow \bar{o}}{(x \leftarrow m(y), \bar{o}) \in R_{\text{call}}}$
abstract object mapping	(program abstract objects) $\dfrac{(x \leftarrow X(), \bar{o}) \in R_{\text{alloc}}}{\bar{o} \rightsquigarrow o = (x \leftarrow X())}$ (proxy objects) $\dfrac{(*, \bar{o}) \notin R_{\text{alloc}},\ \forall i \in \{1, ..., k\}\ (x_i \leftarrow m_i(y_i), \bar{o}) \in R_{\text{call}}}{\bar{o} \rightsquigarrow p = \{m_1, ..., m_k\}}$
missing points-to edges	$\dfrac{(x \leftarrow m(y), \bar{o}) \in R_{\text{call}},\ \bar{o} \rightsquigarrow o,\ x \hookrightarrow o \notin \Pi}{x \hookrightarrow o \in \Pi_{\text{miss}}}$
optimistic static points-to edges	$\Pi =$ (apply Figure 2 with the constructed Π_{miss})

Figure 3 Given a program P, **Optix** adds monitors to P. It uses reports issued by these monitors during executions to compute the counterexamples Π_{miss}, which the static analysis in Figure 2 uses to compute optimistic points-to edges $\Pi \subseteq \mathcal{V}_P \times (\mathcal{O}_P \cup \mathcal{P})$.

- **Eventually sound:** Since we monitor every variable and abstract object, we are guaranteed to detect any counterexample, including the first to occur during execution. Furthermore, since Π^*_{miss} is finite, only finitely many counterexamples are ever reported. Thus, there exists some execution $i_0 \in \mathbb{N}$ of the program during which the last counterexample is reported. Then, the points-to set Π_i computed by our algorithm after the ith execution is sound for all executions $i \geq i_0$.
- **Precise:** Any sound set of points-to edges Π' must contain the missing points-to edges Π^*_{miss}. Therefore, $\Pi_{\text{miss}} \subseteq \Pi^*_{\text{miss}} \subseteq \Pi'$. Since computing a transitive closure is monotone, it follows that $\Pi \subseteq \Pi^* \subseteq \Pi'$.
- **Automatic:** Our static analysis requires no human input.

In our example, the static points-to set Π is sound after the counterexample `data` $\hookrightarrow o_{\text{str}}$ is reported, since the static analysis then computes the remaining missing points-to edge `dataCopy` $\hookrightarrow o_{\text{str}}$.

3.4 Optimized Monitoring

We now describe how to reduce monitoring.

Restricting to function calls. Recall that monitoring `dataCopy` is unnecessary – the missing edge `dataCopy` $\hookrightarrow o_{\text{str}}$ is computed by the static analysis once `data` $\hookrightarrow o_{\text{str}}$ is reported, so it suffices to monitor `data`. In general, it suffices to monitor function calls and allocations:

▶ **Proposition 6.** The monitoring scheme $M_{\text{min}} = M_{\text{alloc}} \cup M_{\text{call}}$ is sound and precise, where $M_{\text{alloc}} = \mathcal{O}$ and $M_{\text{call}} = \{x \leftarrow m(y) \mid m \in \mathcal{M}\}$.

We give a proof in Appendix A.1. Figure 3 shows our algorithm using M_{min}.

Restricting to leaked abstract objects. We can further reduce the size of M_{alloc} – library functions can only access abstract objects reachable from the parameter y of a call $x \leftarrow m(y)$, which implies that the return value r_m can only point to such an abstract object o, so it suffices to restrict M_{alloc} to include abstract objects that may leak into missing code. It is even sound to use the monitoring scheme $\tilde{M}_{\text{alloc}}$, which monitors allocations o such that o may be explicitly passed to the library via a function call $x \leftarrow m(y)$, where $y \hookrightarrow o$:

$$\tilde{M}_{\text{alloc}} = \{o \in \mathcal{O} \mid y \hookrightarrow o \in \Pi \text{ where } x \leftarrow m(y)\}.$$

This monitoring scheme is subtler than the schemes described previously, since the monitors $\tilde{M}_{\text{alloc}}$ depend on the current points-to edges Π. Therefore, the instrumentation may need to be updated when counterexamples are reported and Π is updated. In particular, if $y \hookrightarrow o$ is newly added to Π, where $x \leftarrow m(y)$, then o is added to $\tilde{M}_{\text{alloc}}$ so the instrumentation must be updated. We can soundly use $\tilde{M}_{\text{min}} = \tilde{M}_{\text{alloc}} \cup M_{\text{call}}$:

▶ **Proposition 7.** The monitoring scheme $\tilde{M}_{\text{min}}$ constructed using the current points-to edges Π is sound and precise.

We give a proof in Appendix A.2. Since the number of possible counterexamples is still finite, at some point no further counterexamples are reported. By Proposition 7, no counterexamples occur in any subsequent executions, i.e., Π is sound for all subsequent executions.

Minimality. Our monitoring scheme is minimal in the following sense:

▶ **Proposition 8.** Assume that the rules used to compute $\tilde{M}_{\text{alloc}}$ do not generate any false positives, i.e., for every allocation $o \in \tilde{M}_{\text{alloc}}$, there exists an execution during which a concrete object allocated at o is passed as an argument to a library function. Then, for any strict subset $M \subsetneq \tilde{M}_{\text{min}}$, there exist implementations of the library and program executions such that M fails to report a counterexample, i.e., using M is not eventually sound.

In other words, our monitoring scheme is minimal except for potential imprecision when computing $\tilde{M}_{\text{alloc}}$. We give a proof in Appendix A.3.

4 Abstract Objects in the Library

We now remove the assumption that no allocations occur inside missing code.

4.1 Proxy Objects

Suppose that allocations can occur inside library code. Let $\mathcal{O} = \mathcal{O}_P \cup \mathcal{O}_L$, where abstract objects in $\mathcal{O}_P$ are in available code and abstract objects in $\mathcal{O}_L$ are in missing code. Then, our analysis cannot compute points-to edges $x \hookrightarrow o$, where $o \in \mathcal{O}_L$. As described previously, we assume that the static analysis only needs to compute relations involving program values. However, points-to edges $x \hookrightarrow o \in \mathcal{V}_P \times \mathcal{O}_L$ (i.e., x is in the program but o is not) are often needed to compute relations between program variables, e.g., aliasing and concrete types.

For example, in Figure 1, if `mkStr` is missing, then o_{str} is missing, so our static analysis cannot compute `str` $\hookrightarrow o_{\text{str}}$ (among others). Furthermore, we do not assume the ability to instrument missing code, so we cannot dynamically detect these points-to edges. However, this points-to edge is needed to determine that `str` may have type `String`, and that `str` and `data` may be aliased.

We handle allocations in library code by constructing the following:

▶ **Definition 9.** A *proxy object mapping* ϕ maps $\bar{o} \rightsquigarrow p$, where $\bar{o}$ is a concrete object allocated in missing code, and $p = \phi(\bar{o}) \in \mathcal{P}$ is a fresh abstract object called a *proxy object*; here, $\mathcal{P}$ is the set of all proxy objects.

In other words, ϕ is the abstract object mapping for concrete objects allocated in missing code. We describe how to construct ϕ and $\mathcal{P}$ below.

Given ϕ, our analysis proceeds as before. It makes optimistic assumptions, initializes $\Pi_{\text{miss}} \leftarrow \emptyset$, and instruments the program using the monitoring scheme $\tilde{M}_{\text{min}}$ defined in Proposition 6. For any report $(x \leftarrow *, \bar{o})$, if $\bar{o}$ is not allocated at a visible allocation, our analysis concludes that $\bar{o}$ must have been allocated in missing code, so it adds $\bar{o} \rightsquigarrow p = \phi(\bar{o})$ to the abstract object mapping. Now, if a detected dynamic points-to edge $x \hookrightarrow p$ is missing, it is reported as a counterexample and added to $\Pi_{\text{miss}} \subseteq \mathcal{V}_P \times (\mathcal{O}_P \cup \mathcal{P})$, and Π is recomputed using the static analysis in Figure 2. As long as $\mathcal{P}$ is finite, then this approach is eventually sound, since there can only be a finite number of counterexamples $x \hookrightarrow p$.

In our example, `str` is monitored since `mkStr` is missing. Upon execution, our instrumentation detects `str` $\hookrightarrow \bar{o}_{\text{str}}$, and determines that $\bar{o}_{\text{str}}$ (allocated at o_{str}) is allocated in missing code. Supposing that $p_{\text{str}} = \phi(\bar{o}_{\text{str}}) \in \mathcal{P}$, our analysis adds $\bar{o}_{\text{str}} \rightsquigarrow p_{\text{str}}$ to the abstract object mapping. Thus, our instrumentation reports the counterexample `str` $\hookrightarrow p_{\text{str}}$. Assuming execution continues, then our instrumentation additionally reports the counterexample `data` $\hookrightarrow p_{\text{str}}$. Both counterexamples are added to Π_{miss}, from which our static analysis computes `dataCopy` $\hookrightarrow p_{\text{str}} \in \Pi$.

We now discuss how to construct ϕ and $\mathcal{P}$. The relevant information characterizing a concrete object is the following:

▶ **Definition 10.** The *dynamic footprint* of a concrete object $\bar{o}$ is the set of all visible variables that ever point to $\bar{o}$ during an execution.

The concrete type of $\bar{o}$ may also be available to the static analysis, which we discuss in Section 6.3. Aside from concrete types, the dynamic footprint contains all information about $\bar{o}$ available to the static analysis, namely, the visible variables that point to $\bar{o}$.

Then, the proxy object mapping ϕ should map each concrete object $\bar{o}$ to a proxy object p so that the corresponding *static footprint* $\{x \in \mathcal{V}_P \mid x \hookrightarrow p \in \Pi^*\}$ soundly overapproximates the dynamic footprint of $\bar{o}$ as precisely as possible. This way, clients of the points-to analysis can be eventually soundly and precisely computed (as long as they only depend on available information), e.g., it ensures that aliasing for program variables is eventually soundly and precisely computed (concrete types are eventually soundly and precisely computed using a simple extension; see Section 6.3).

On the other hand, ϕ should also avoid introducing unnecessary proxy objects, or else more executions may be required for the analysis to become sound. Two extremes highlight these opposing desirable properties:

- **Unbounded $\mathcal{P}$:** Map each concrete object to a fresh proxy object $\phi(\bar{o}) = p_{\bar{o}}$.
- **Singleton $\mathcal{P}$:** Map each concrete object to a single proxy object $\phi(\bar{o}) = p$.

On the one hand, if we use a fresh proxy object for every concrete object, then there would be an unbounded number of proxy objects, which would mean our algorithm is no longer eventually sound (since there may be an unbounded number of missing points-to edges). Alternatively, using a single proxy object can be very imprecise; for example, for *any* pair of calls $x \leftarrow m(y)$ and $x' \leftarrow m'(y')$, our analysis concludes that x and x' may be aliased.

We first describe an *ideal proxy object mapping*, which constructs $\mathcal{P}$ as the set of possible dynamic footprints, and constructs ϕ to map $\bar{o}$ to its dynamic footprint. Points-to sets computed using *any* static analysis together with the ideal proxy object mapping satisfy the above property, i.e., that the static footprints soundly overapproximate the dynamic footprints as precisely as possible.

Because the static analysis is flow-insensitive, the ideal proxy mapping is actually more precise than necessary. Therefore, our analysis uses a coarser proxy object mapping computed by our analysis, which essentially restricts the dynamic footprint to function return values. Finally, we show that this coarser proxy object mapping is as precise as the ideal proxy object mapping for our points-to analysis described in Figure 2.

4.2 Ideal Proxy Object Mapping

Our “ideal” construction of proxy objects exactly captures dynamic footprints:

▶ **Definition 11.** An *ideal proxy object* $\tilde{p} \in \tilde{\mathcal{P}} = 2^{\mathcal{V}_P}$ is a set of visible variables. The *ideal proxy object mapping* $\tilde{\phi}(\bar{o}) \in \tilde{\mathcal{P}}$ is the dynamic footprint of $\bar{o}$.

For a concrete object $\bar{o}$ allocated in missing code, we can compute $\tilde{\phi}(\bar{o})$ by monitoring all visible variables and identifying all visible variables that ever point to $\bar{o}$. In our example, suppose that we continue executing `main` even if a counterexample is detected and reported. Furthermore, suppose that the concrete object $\bar{o}_{\text{str}}$ is allocated at missing abstract object o_{str} in an execution where `randBool` returns `false`. Then, $\tilde{\phi}$ maps $\bar{o}_{\text{str}}$ to ideal proxy object $\tilde{p}_{\text{str}} = \{\texttt{str},\ \texttt{data}\}$. The reported counterexamples

$$\tilde{\Pi}_{\text{miss}} = \{\texttt{str} \hookrightarrow \tilde{p}_{\text{str}},\ \texttt{data} \hookrightarrow \tilde{p}_{\text{str}}\}$$

are added to our static analysis, which additionally computes $\texttt{dataCopy} \hookrightarrow \tilde{p}_{\text{str}}$.

Let $\tilde{\Pi}^*_{\text{miss}} \subseteq \mathcal{V}_P \times (\mathcal{O}_P \cup \tilde{\mathcal{P}})$ be the missing points to edges when using ideal proxy objects, and let $\tilde{\Pi}^* \subseteq \mathcal{V}_P \times (\mathcal{O}_P \cup \tilde{\mathcal{P}})$ be the points-to edges computed using $\Pi_{\text{miss}} = \tilde{\Pi}^*_{\text{miss}}$. Then:

▶ **Proposition 12.** If $x \hookrightarrow \bar{o}$ occurs during execution and $\bar{o}$ is allocated at abstract object o, then $x \hookrightarrow o \in \tilde{\Pi}^*$ (if $o \in \mathcal{O}_P$) or $x \hookrightarrow \tilde{p} \in \tilde{\Pi}^*$ (where $\tilde{p} = \tilde{\phi}(\bar{o})$).

In other words, clients of the points-to analysis that only refer to program variables are eventually sound. For example, if two program variables x and y may be aliased, then there must be some execution in which they both point to a concrete object $\bar{o}$. Then, our analysis finds points-to edges $x \hookrightarrow \tilde{p}$ and $y \hookrightarrow \tilde{p}$, where $\tilde{p} = \tilde{\phi}(\bar{o})$, so the alias analysis determines that x and y may be aliased. Also:

▶ **Proposition 13.** Let $\Pi \subseteq \mathcal{V}_P \times (\mathcal{O}_P \cup \mathcal{O}_L)$ be the points-to set computed using the static analysis in Figure 2 with all code available (and $\Pi_{\text{miss}} = \emptyset$). For $o \in \mathcal{O}_P$, if $x \hookrightarrow o \in \tilde{\Pi}^*$, then $x \hookrightarrow o \in \Pi$. For $\tilde{p} = \tilde{\phi}(\bar{o}) \in \tilde{\mathcal{P}}$, if $x \hookrightarrow \tilde{o} \in \Pi^*$, then $x \hookrightarrow o \in \Pi$, where o is the statement where $\bar{o}$ was allocated.

In other words, $\tilde{\Pi}^*$ is at least as precise as the points-to edges Π computed with all code available. We prove these two propositions in Appendix B.1 & B.2. In our example, with all code available, we compute $\texttt{str} \hookrightarrow o_{\text{str}}$, $\texttt{data} \hookrightarrow o_{\text{str}}$, and $\texttt{dataCopy} \hookrightarrow o_{\text{str}}$, which is equivalent to $\tilde{\Pi}^*$ (replacing $\tilde{p}_{\text{str}}$ with o_{str}).

4.3 Proxy Object Mapping

The ideal proxy object mapping is more precise than necessary. Continuing our example (where we assume execution continues even after counterexamples are detected and reported), consider a second execution where `randBool` returns true. Then, the concrete object $\bar{o}'_{\text{str}}$ allocated at missing abstract object o_{str} is mapped to the ideal proxy object $\tilde{p}'_{\text{str}} = \{\texttt{str}, \texttt{data}, \texttt{dataCopy}\}$. However, the static footprint of $\tilde{p}'$ equals that of $\tilde{p}$ (from the first execution, where `randBool` returns false), even though $\tilde{p} \neq \tilde{p}'$ – i.e., $\bar{o}_{\text{str}}$ and $\bar{o}'_{\text{str}}$ map to different ideal proxy objects, but their relevant points-to behaviors appear identical to the (flow-insensitive) static analysis. In fact, all information about a concrete object available to the static analysis can be summarized by the following:

▶ **Definition 14.** The *dynamic function footprint* of a concrete object $\bar{o}$ is the set of library functions $m \in \mathcal{M}$ such that $r_m \hookrightarrow \bar{o}$ during execution.

Now, we use the following proxy object mapping:

▶ **Definition 15.** A *proxy object* $p \in \mathcal{P} = 2^{\mathcal{M}}$ is a set of library functions. The *proxy object mapping* $\phi(\bar{o}) \in \mathcal{P}$ is the dynamic function footprint of $\bar{o}$.

To compute ϕ, it suffices to monitor calls $x \leftarrow m(y)$ to missing functions. Continuing our example, ϕ maps the concrete object $\bar{o}_{\text{str}}$ allocated at missing abstract object o_{str} to $p_{\text{str}} = \{\texttt{mkStr}, \texttt{get}\}$ regardless of the return value of `randBool`. If `randBool` returns true, then the reported counterexamples are

$$\Pi_{\text{miss}} = \{\texttt{str} \hookrightarrow p_{\text{str}},\ \texttt{data} \hookrightarrow p_{\text{str}},\ \texttt{data} \hookrightarrow p_{\text{str}}\},$$

in which case our static points-to analysis does not compute any additional edges. If `randBool` returns false, then the reported counterexamples are

$$\Pi_{\text{miss}} = \{\texttt{str} \hookrightarrow p_{\text{str}},\ \texttt{data} \hookrightarrow p_{\text{str}}\},$$

from which our static analysis also computes $\texttt{dataCopy} \hookrightarrow p_{\text{str}}$. The static footprint of p_{str} is the same either way, and also equals those of $\tilde{p}_{\text{str}}$ and $\tilde{p}'_{\text{str}}$.

Let $\Pi^*_{\text{miss}} \subseteq \mathcal{V}_P \times (\mathcal{O}_P \cup \mathcal{P})$ be the set of all missing points-to edges using proxy objects, and let $\Pi^* \subseteq \mathcal{V}_P \times (\mathcal{O}_P \cup \mathcal{P})$ be the points-to edges computed using $\Pi_{\text{miss}} = \Pi^*_{\text{miss}}$. Then:

▶ **Proposition 16.** For any abstract object $o \in \mathcal{O}_P$, $x \hookrightarrow o \in \tilde{\Pi}^* \Leftrightarrow x \hookrightarrow o \in \Pi^*$. Furthermore, for any concrete object $\bar{o}$ allocated in missing code, letting $\tilde{p} = \tilde{\phi}(\bar{o})$ and $p = \phi(\bar{o})$, we have $x \hookrightarrow \tilde{p} \in \tilde{\Pi}^* \Leftrightarrow x \hookrightarrow p \in \Pi^*$.

In other words, the points-to edges computed using our proxy object mapping is as sound and precise as using the ideal proxy object mapping. Therefore, using proxy objects is also sound and precise in the sense of Propositions 12 and 13. We prove this proposition in Appendix B.3. Finally, the following result says that the monitoring scheme described in Section 3.4 is still sound (it follows since we can compute ϕ using only M_{call}):

▶ **Proposition 17.** The monitoring scheme $\tilde{M}_{\text{min}}$ is sound and precise.

5 Specification Inference

Rather than simply adding reported missing points-to edges to Π_{miss}, we can use them to infer specifications summarizing missing code, which transfers information learned from the counterexample to other calls to the same library function. We use the specification inference algorithm in [8]. Given a reported missing points-to edge $x \hookrightarrow o$, this algorithm infers specifications in two steps:

```
Object m_gen(Object ob) {
  while(true) {
    ob = new Object();
    ob.f = ob;
    ob = ob.f; }
  return ob; }
```

```
Object m_res(Object ob) {
  Object r;
  this.f = ob;
  r = ob;
  r = this;
  r = this.f;
  return r; }
```

Figure 4 Pessimistic functions used for specification inference; m_{gen} (left) is fully general (assuming functions do not access global state), whereas m_{res} (right) is restricted in various ways. For simplicity, we omit the receiver in m_{gen}.

- **Pessimistic assumptions:** Take $\hat{m} = m_{\text{pess}}$ for every missing function $m \in \mathcal{M}$, for some function m_{pess} (see below), and run the static analysis using $\hat{m}$ in place of m.
- **Minimal statements:** Compute a minimal subset of *pessimistic statements* (i.e., statements in the functions m_{pess}) that are needed to compute $x \hookrightarrow o$ statically; these statements are the inferred specifications.

The second step involves computing the static analysis using a shortest-path style algorithm, where pessimistic statements are assigned weight 1 and all other statements are assigned weight 0. When computing the transitive closure according to the rules in Figure 2, a priority queue is used in place of a worklist, where the priority of each points-to edge in the queue is the number of pessimistic statements needed to derive it. In particular, the priority of the derived points-to edge is the sum of the weights of the statements in the premise as well as the priorities of points-to edges in the premise. Thus, if a computed points-to edge has positive weight, we know that it can only be derived using a statement in m_{pess}.

Pessimistic function. A key design choice is the pessimistic function m_{pess} to use. The choice in [8], which we term the *general* function m_{gen}, is shown in Figure 4 (left). The code shown in this figure is compilable Java code, except we have added a new field `f` to the `Object` class. In particular, this code can be analyzed by our static analysis. As described above, during specification inference, we run our static analysis using either m_{gen} or m_{res} in place of m for every missing function $m \in \mathcal{M}$.

Using m_{gen} is sound, assuming library functions do not access global fields; see [8] for a formal proof. Intuitively, the argument ob of m_{gen} can be assigned to its return value $r_{m_{\text{gen}}}$; furthermore, the while loop ensures that arbitrary fields in the argument ob can be stored into fields in the return value $r_{m_{\text{gen}}}$ or into other fields of ob. In principle, we could even modify m_{gen} to handle library functions that access global fields by adding statements `x = X.g` and `X.g = x`, for every global field `X.g`. However, none of the specifications we have written so far access global fields, so this restriction improves running time without unsoundness in practice.

Unfortunately, even with this restriction, using m_{gen} results in a huge search space of candidate specifications. As a consequence, when using m_{gen}, the specification inference algorithm infers many incorrect specifications – in particular, there may be many specifications that yield a missing points-to edge, so the algorithm may infer the wrong ones.

Instead, we use pessimistic assumptions that restrict the search space to only consider candidate specifications that are common in practice, thus reducing the possibility of inferring an incorrect specification (at the cost of being unable to infer more complex specifications). In particular, we only consider candidate specifications that (i) do not accesses deep field paths, (ii) only access receiver fields, and (iii) does not allocate objects. These constraints lead to the *restricted* function m_{res} shown in Figure 4 (right).

Proxy object specifications. One of the restrictions on m_{res} is to assume that the candidate specifications do not allocate objects. To alleviate the consequences of this restriction, we separately infer specifications that allocate objects. In particular, we infer *proxy object specifications* of the form $(X, \{m\})$, where $X \in \mathcal{C}$ and $m \in \mathcal{M}$ is a library function. This specification says that a new object of type X is allocated onto the return value of a function. We infer a proxy object specification for any proxy object $p \in \mathcal{P}$ we observe dynamically such that the function footprint of p consists of a single function m.

6 Extensions

6.1 Shared Fields

In Section 3.1, we made the assumption that no shared fields $f \in \mathcal{F}_P \cap \mathcal{F}_L$ exist. Our analysis handles a shared field f by converting stores $x.f \leftarrow y$ and loads $x \leftarrow y.f$ in the program into calls to setter and getter functions, respectively. To do so, we have to know which fields may be accessed by the library. We make the weaker assumption that the library does not access fields defined in the program – then, our analysis performs this conversion for every field f defined in the library that is accessed by the program.

6.2 Callbacks

Android apps can register callbacks to be invoked by Android when certain events occur, e.g., the program can implement the callback `onLocationChanged`, which is invoked when the user location changes. If callbacks are not specified, then the static analysis may unsoundly mark them as unreachable. We use the approach in [7] to eventually soundly compute reachable program functions. In particular, a *potential callback*, is a program function that overrides a framework function. Intuitively, potential callbacks are the functions "known" to the framework. For each potential callback m that is marked as unreachable by the static analysis, we instrument m to record whether m is ever reached. This algorithm is eventually sound since there are only finitely many potential callbacks. Also, the instrumentation eventually incurs no overhead – once no more counterexamples are reported, the instrumentation is never triggered.

In addition, some callbacks are passed parameters from the Android framework. For example, consider the code on the left:

```
void onLocationChanged(Location loc) {    void onLocationChanged() {
  Location copy = loc; }                    Location loc = Location.getLocation();
                                            Location copy = loc; }
```

Here, `loc` points to an abstract object o_{loc}. In this case, o_{loc} is allocated in the framework, but it may also be allocated in program code. We must specify the abstract objects that `loc` may point to, or else our points-to analysis is unsound. The code on the right replaces the parameter with a call that retrieves `loc` from the framework, which is semantically equivalent to the code on the left. Thus, we can think of `loc` as a "return value" passed to `onLocationChanged`; by Proposition 6, it suffices to monitor all callback parameters.

6.3 Concrete Types

Some client analyses additionally need the concrete type $X \in \mathcal{C}$ of abstract objects $o = (x \leftarrow X())$, for example, virtual call resolution. To compute concrete types for proxy objects, each monitor $x \leftarrow *$ additionally records the concrete type of the concrete object pointed to by x

after executing the statement. Then, the proxy objects are extended to $\mathcal{P} = \mathcal{C} \times 2^{\mathcal{M}}$, and the proxy object mapping ϕ maps $\bar{o} \rightsquigarrow p = (X, F)$, where $F \subseteq 2^{\mathcal{M}}$ is the dynamic function footprint of $\bar{o}$, and $X \in \mathcal{C}$ is the recorded concrete type of $\bar{o}$.

6.4 Context- and Object-Sensitivity

Our analysis extends to k-context-sensitive points-to analyses with two changes. First, the abstract objects considered are typically pairs $o = (c, h)$, where c is a calling context and h is an allocation statement, so monitors on allocation statements $x \leftarrow X()$ also record the top k elements of the current callstack. Second, the points-to edge typically keeps track of the calling context d in which a variable v may point to abstract object o. Therefore, monitors on calls to missing functions $x \leftarrow m(y)$ also record the top k elements of the current callstack.

In particular, our analysis may (i) detect that $(d, v) \hookrightarrow \bar{o}$ (i.e., d is the callstack when v pointed to $\bar{o}$), and (ii) $\bar{o} \rightsquigarrow (c, h)$ (i.e., $\bar{o}$ was allocated at statement h, and c is the callstack when $\bar{o}$ was allocated). Then, our analysis reports missing points-to edge $(d, v) \hookrightarrow (c, h)$. We use a 1-CFA points-to analysis in our evaluation; in this case, the calling context is simply the function in which the allocation or call to a missing function occurs. Our approach can be extended to handle object-sensitive analyses, by including instrumentation that records the calling context (which now includes the value of the receiver).

Finally, we can also handle on-the-fly callgraph construction – if a missing points-to edge $x \hookrightarrow o$ is reported, and there is a virtual function call $x.m()$ in the program, then the possible targets of $x.m()$ are updated to take into account the concrete type of o. The instrumentation may need to be updated based on this new information. Assuming the number of possible call targets is finite, this approach is eventually sound and precise.

7 Implementation

We have implemented our eventually sound points-to analysis, including all extensions described in Section 6 (using a 1-CFA points-to analysis), for Android apps in a tool called **Optix**. The missing code consists of Android framework methods, which we assume cannot be statically analyzed (since the Android framework heavily uses native code and reflection) or instrumented (which requires a custom Android installation). The static analysis framework we use predates **Optix**, and uses hand-written specifications to model missing code. Specifications have only been written for methods deemed relevant to a static information flow client – of the more than 4,000 Android framework classes, only 175 classes have specifications. Framework methods without specifications appear as missing code to our static analysis.

Optix instruments Android apps using our optimized monitoring scheme $\tilde{M}_{\text{min}}$. It computes eventually sound points-to sets and infers specifications based on reported missing points-to edges. We instrument apps using the Smali assembler and disassembler [26]. To monitor a statement `x=...`, we record (i) the value `System.identityHashCode(x)`, which identifies the concrete object pointed to by `x`, (ii) the concrete type `x.getClass()` of `x`, and (iii) the method containing the statement and the offset of that statement in the method. [5] This data is uploaded to a server in batches (by default, once every 500ms), which post-processes

[5] Even though `System.identityHashCode` is not guaranteed to return a unique hash, it is a sound overapproximation.

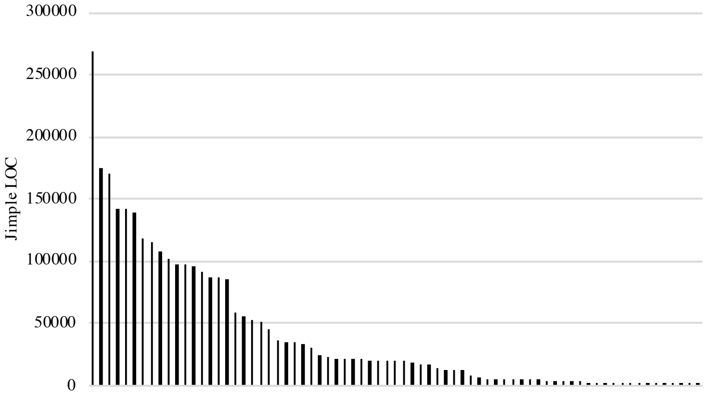


Figure 5 Sizes of the 73 apps in our benchmark in terms of Jimple list of code (LOC).

it to compute missing points-to edges and infer specifications. To obtain traces, we execute apps in the Android emulator and use Monkey [24] to inject touch events. We measure overhead using the Android profiler.

We have implemented the points-to analysis, the monitoring optimization, and the specification inference algorithm in a version of the Chord program analysis framework [40] modified to use Soot as a front end [48]. The specification inference algorithm is based on shortest-path context-free reachability, described in [8]. We use a 1-CFA points-to analysis. As we discuss Section 6.4, using our more precise points-to analysis is eventually sound.

We use the information flow client from [21], which uses specifications to model missing code. It performs a static explicit information flow analysis [42] based on a static points-to analysis. The information flow analysis is standard – it looks for paths from annotated sources (e.g., location, contacts, etc.) to annotated sinks (e.g., SMS messages, Internet, etc.) in the Android framework [23, 5]. All analyses are computed using BDDBDDB [49].

8 Evaluation

We evaluate **Optix** on a benchmark of 73 Android apps, including battery monitors, games, wallpaper apps, and contact managers. The benchmarks are from two sources: the majority (40) are provided by a major security corporation (the “industry benchmark”), and the remaining (33) are provided as part of the DARPA APAC project on Android malware (the “DARPA benchmark”). The industry benchmark includes apps collected from the Google Play Store; the apps are primarily malware that leak sensitive information such as location, contacts, SMS messages, etc. The DARPA apps were challenging malware instances developed by a third party contractor as part of the program. Examples of apps from the industry benchmark include an app for monitoring your battery consumption, side scrolling action game, an implementation of the game of Mahjong, a maze game, and apps that let you set animated wallpapers. Examples of apps from the DARPA benchmark include a note taking app, an app that lets you use SMS messages to command your phone to perform various tasks, an app that keeps track of your jogging routes, a news collator, an app for organizing your podcasts, and an app for sharing your location. We provide the industry benchmark; [6] we cannot release the DARPA benchmark, but provide brief descriptions of

[6] `https://drive.google.com/open?id=1LzRhwtPisWWwKTd7lnHy7CE5Gf9iRkH4`

Rank	Recording Overhead (%)				Data (MB/hr)	
	initial	updated	# specs	worst	initial	worst
1	52.0 (0.38)	31.0	15	90.8 (0.02)	0.80 (0.59)	1.59 (0.16)
2	50.0 (0.02)	17.6	10	77.6 (0.01)	0.39 (0.01)	0.57 (0.39)
3	38.7 (0.10)	6.7	5	72.4 (0.04)	0.29 (0.38)	0.46 (0.30)
4	33.5 (0.65)	6.9	1	70.6 (0.15)	0.24 (0.15)	0.33 (0.78)
5	31.7 (0.10)	19.7	5	61.6 (0.23)	0.24 (0.55)	0.33 (0.64)
6	26.7 (0.25)	–	–	52.5 (0.03)	0.23 (0.80)	0.25 (0.04)
median	4.6 (0.10)	–	–	8.3 (0.10)	0.01 (0.35)	0.02 (0.30)

Figure 6 The runtime overhead from recording data and the (compressed) size of the data generated in one hour. Each is divided into initial and worst-case. The "updated" overhead is obtained by adding specifications to reduce monitoring, and "# specs" is the number of specifications added to do so. For each column, the table shows the largest six values and the median value across our benchmark. The coefficient of variation (across three runs) is shown in parentheses.

the apps in this benchmark in Appendix C. Finally, in Figure 5, we show a plot of the sizes of the apps in Jimple lines of code (LOC). We omit 11 apps that fail to run on the standard Android emulator, leaving 62 apps. First, we use **Optix** to instrument each Android app and study the instrumentation overhead. Second, we show how **Optix** computes points-to edges over time, and show that the number of computed edges does not explode. Third, we show how our analysis can be used to improve an information flow client.

8.1 Instrumentation Overhead

We evaluate the runtime overhead of our monitoring scheme $\tilde{M}_{\text{min}}$ described in Section 3.4. Recall from Section 3.4 that our optimized instrumentation scheme may add instrumentation over time. We consider two settings:

- **Initial:** This configuration represents the instrumentation overhead for a new app using the current program analysis. In particular, we use the initial instrumentation scheme where $\tilde{M}_{\text{alloc}}$ is constructed with no known counterexamples (i.e., $\Pi_{\text{miss}} = \emptyset$). Also, we use all existing handwritten points-to specifications, representing the realistic scenario where some manually provided information is used in addition to automatic inference.
- **Worst:** This configuration represents the absolute upper bound on the overhead. In particular, we monitor apps using the worst-case instrumentation scheme where $\tilde{M}_{\text{alloc}}$ contains all abstract objects that may leak into missing code. Furthermore, we remove all handwritten points-to specifications.

We executed instrumented apps in a standard emulator using Monkey for one hour, and measure instrumentation overhead in each setting, on a 3.30GHz Intel Xeon E5-2667 CPU with 256 GB of memory. The server for collecting and analyzing reports was run concurrently on a different CPU core of the same machine. Our results are averaged over three runs.

Results. We show the highest runtime overheads in Figure 6, including the runtime overhead from recording data and the amount of data generated in an hour, for both the initial setting and the worst-case setting.[7] Columns "updated" and "# specs" are discussed below. We plot the runtime overhead of our recording instrumentation in Figure 7 (a), where the apps along the x-axis are sorted according to the overhead in the worst-case setting.

[7] We ran a small subset of apps on a real device and consistently measured smaller overhead; the emulator gives a coarser measure of execution time that we round up.

Discussion. The overhead incurred by recording data is less than 5% for more than half of the apps, showing that in most cases the automatically instrumented programs have acceptable performance. Even in the worst case, more than half the apps have less than 10% overhead. Still, there are outliers, with 5 apps incurring more than 20% overhead with initial instrumentation, and in the worst-case, 9 apps incurred more than 20% overhead. Unsurprisingly, the high-overhead outliers have instrumentation in inner loops of the app; in such cases the overhead can be reduced (see below). Finally, the amount of data generated is very small. Even in the worst case, for all but one of the apps, less than 1.0 MB of (compressed) data was generated in one hour. The median amount of data generated is about 2.0 KB, which is negligible. Data can therefore be stored and transmitted when the app is idle, so the overhead due to uploading data does not affect the user experience.

Reducing runtime overhead. Any program where instrumentation is required in a tight inner loop is particularly challenging for dynamic analysis. Standard sampling techniques can be used to reduce overhead in these cases [33], though eventual soundness may no longer hold. Alternatively, both $\tilde{M}_{\text{alloc}}$ and M_{call} decrease in size as specifications are added and reach zero when there are no missing specifications. For a given program, we can test the program to determine which monitors are frequently triggered, and compute which missing functions require specifications for these monitors to be removed. Providing or inferring specifications for these functions would allow us to remove the expensive monitors. We do so for the five apps with initial overhead greater than 20%. In Figure 6 (left), we show both the number of specifications we added for that app ("# spec") and the resulting overhead ("updated"). For all but the top app, we were able to reduce the overhead below 20% by adding specifications for at most 10 Android framework methods; again, the overhead can be reduced to any desired level by adding more specifications.

8.2 Reported Counterexamples

Next, we evaluate how the computed points-to edges vary over time. We use our algorithm to compute points-to sets based on the reported counterexamples from Section 8.1. We show that the number of reported counterexamples does not explode over time – otherwise, the number of counterexamples discovered in production may be unacceptably high. Furthermore, we show that a tail of reported counterexamples continues to occur for some apps, which shows that running instrumented apps in production is necessary.

This experiment uses the worst-case setting where all handwritten specifications have been removed. Note that the worst-case instrumentation detects all counterexamples since it never needs to be updated, so this approach is sound. It may actually overapproximate the number of counterexamples – e.g., if a points-to edge $x \hookrightarrow o$ can be computed from another counterexample $y \hookrightarrow o$ that was already detected, our worst-case instrumentation reports $x \hookrightarrow o$ even though it is no longer a counterexample.

Counterexamples over time. Figure 7 (b) shows the cumulative number of reported missing points-to edges as execution progresses. More precisely, for each point in the execution trace (x-axis), it shows what fraction of reported missing points-to edges were discovered before that point. The values are averaged over all apps. By definition, at the end of the trace ($x = 1.0$), the fraction of reported missing points-to edges also goes to $y = 1.0$.

As can be seen, a large fraction of reports are made early on, with about 65% of reports made within 20% of the execution trace. We expect the number of reported counterexamples to continue to converge over time, and should not grow substantially larger. However, the

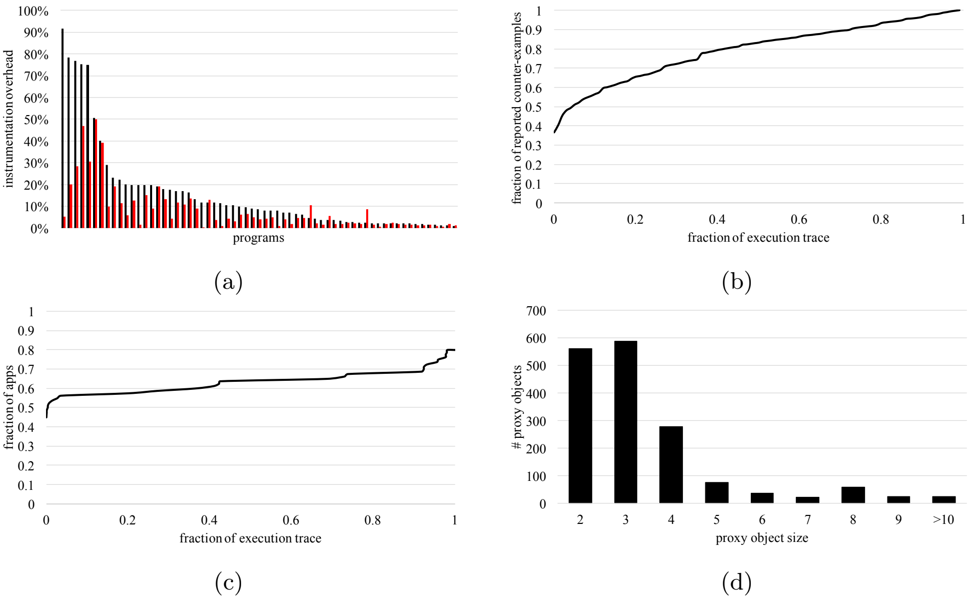


Figure 7 (a) Runtime overhead of our recording instrumentation in the worst-case setting (black) and the initial setting (red). The overheads are sorted by the overhead for the worst-case setting. (b) The x-axis is a fraction of the execution trace, and the curve shows the fraction of discovered missing points-to edges that discovered up to that point in the execution trace (averaged over all apps). (c) The x-axis is again a fraction of the execution trace, and the curve shows the number of apps for which no further missing points-to edges are reported after that point in the execution trace. (c) The distribution of the sizes of the proxy objects (i.e., the size of its dynamic function footprint), omitting footprints of size one.

curve is not yet flat at the end of the execution trace, which indicates that more missing points-to edges are still being reported. Therefore, it is important to continue monitoring these apps in production to detect additional counterexamples.[8]

Last discovered counterexample. Figure 7 (c) shows the point in the execution during which the final reported missing points-to edge occurs. More precisely, for each point in the execution trace (x-axis), it shows the fraction of the apps for which the final reported missing points-to edge was reported before that point. This curve goes to $y = 1.0$ at $x = 1.0$, but we cut off apps that have reported missing points-to edges in the last 1% of execution.

A large fraction of apps (about 45%) report no counterexamples. About 10% of apps report no further counterexamples after the first 5% of the trace. At the opposite end of the trace, about 20% of apps have the final reported counterexample in the last 1% to 10% of the trace, and 20% have the final reported counterexample in the final 1% of the trace, so more counterexamples likely remain. Again, this shows that we must continue to monitor apps in production.

[8] Since this curve is an overapproximation of the number of counterexamples, the true curve may actually be decreasing faster than this one. Nevertheless, we believe these results emphasize the importance of continued monitoring, since there are inevitably previously untested code paths that may contain bugs.

	Existing	Inferred	Correct	Accuracy
m_{res}	299	58	49	0.84
m_{gen}	299	159	33	0.22
proxy object	330	422	383	0.91

Figure 8 The number of specifications inferred using each m_{res} and m_{gen}, and the number of proxy object specifications inferred.

App	Jimple LOC	Time (min.)
0C2B78	322K	3.38
b9ac05	268K	1.06
highrail	247K	1.49
game	174K	0.08
androng	170K	0.48
median	19K	0.08

Figure 9 Statistics for the five largest apps used in our evaluation, including the number of Jimple lines of code (i.e., the intermediate representation used by Soot), and the running time of the specification inference algorithm.

Proxy object sizes. Since there are an exponential number of possible proxy objects (in the number of missing functions), we could hypothetically continue to discover many new proxy objects over time. In Figure 7 (d), we show the sizes of the dynamic function footprints of the reported proxy objects. More precisely, we show the number of reported proxy objects (y-axis) for different dynamic function footprint sizes. As can be seen, the vast majority (85%) of reported proxy objects have four or fewer functions in their dynamic function footprint. While there is a long tail of proxy objects with large function footprint sizes, there is no exponential blowup in the number of proxy objects discovered, ensuring that the analysis does not diverge due to proxy objects.

8.3 Specification Inference and a Static Information Flow Client

Finally, we evaluate whether **Optix** benefits an information flow client. We first infer specifications using the algorithm in Section 5, and then run the information flow client on various sets of specifications. The information flow analysis is standard – we look for paths from a set of annotated sources (e.g., location) to a set of annotated sinks (e.g., Internet) in the Android framework [23, 5, 21, 8]. We demonstrate that the inferred specifications enable clients to discover more information flows. However, many of the information flows remain undiscovered because the dynamic analysis is an underapproximation, which again motivates the need to run instrumented apps in production.

Specification inference. We remove all points-to specifications from **Optix**, and then infer specifications from reported counterexamples. Figure 8 summarizes the inferred specifications – a specification is correct if it exactly equals the existing specification (or the one we would have written). Using m_{res} is substantially more accurate than using m_{gen}, which does not infer a single additional specification. Compared to existing specifications, we inferred 174 new points-to specifications, of which 160 were proxy object specifications.

	Empty	Inf.	Base	Base ∪ Inf.	Ex. \ Inf.	Ex.	Ex. ∪ Inf.
specs.	0	432	189	621	371	629	803
flows	0	4	3	39	34	125	125
malware	0	2	3	22	12	49	49

Figure 10 Comparison of different sets of specifications: "Base" includes the most frequently used specifications, "Inf." includes the inferred specifications, and "Ex." includes all handwritten specifications. For each set of specifications, we show the number of specifications in that set ("specs."), the number of information flows computed using those specifications ("flows"), and the number of malicious apps identified, i.e., some malicious information flow was discovered ("malware").

In Figure 9, we show statistics for the five largest apps in our benchmark along with the running time of the inference algorithm (the information flow analysis runs much faster than the inference algorithm). As can be seen, inference scales even to very large apps.

Static information flow client. In Figure 10, we report the number of information flows and the number of malicious apps detected with varying sets of specifications (one malicious app can exhibit multiple flows). If we assume that all points-to specifications are missing ("Empty"), then the information flow client does not identify any information flows, whereas using inferred specifications ("Inf.") computes a small number of flows.

A more representative use case is where the analysis has an incomplete baseline consisting of the most commonly used specifications ("Base"). Our baseline contains specifications for the essential Android framework classes `Bundle` and `Intent`, for the commonly used data serialization classes `JSONArray`, `JSONObject`, and `BasicNameValuePair`, and for a few methods in `java.util`. As can be seen from Figure 10, when using the baseline in conjunction with the inferred specifications, the analysis computes a considerable number of additional flows compared to using the baseline alone (39 vs. 3). The reason the inferred specifications are more beneficial in this setting is that an information flow usually depends on multiple specifications – if a single one of these specifications is missing, then the flow is missing.

Compared to the existing, handwritten specifications ("Ex."), using inferred specifications (together with the baseline specifications) identifies almost a third of the information flows. However, random testing cannot reveal all malicious behavior, since malware developers often try to hide malicious behaviors by triggering them only in response to very specific events, for example, at a certain time [37]. Therefore, our instrumentation is necessary to ensure that we identify additional malicious behaviors as soon as or before they occur, thereby limiting potential damage. Note that we do not recover any new flows when combining inferred specifications with existing specifications – prior to our evaluation, we have already identified all specifications needed to recover flows in these apps.

Finally, as an alternative way to evaluate the value of the inferred specifications, we consider omitting the inferred specifications from the set of inferred specifications. Doing so limits the information flow client to identify only 34 flows, which demonstrates that the inferred specifications are crucial for finding many of the information flows in these apps.

9 Discussion

Global variables. We have assumed that library code does not access global variables; so far, none of the specifications we have written so far access global variables. We can extend our framework to handle global variables by instrumenting every load and store to global variables in the library, at the cost of additional runtime overhead. With this modification, our theoretical results continue to hold using the same proofs – in particular, we can think of loads and stores to global variables in the library as calls to setter and getter functions in the library that load and store data from those variables.

Dynamically loaded code. Our approach can be used for dynamically loaded code – the dynamically loaded code is taken to be the missing code, and the code that loads the dynamically loaded code is the available code. We guarantee eventual soundness for points-to edges in the available code. If points-to edges for dynamically loaded code must be computed, then the loaded code can be reported to the static analysis, but the analysis is no longer eventually sound – infinitely many reports may be issued since infinitely many different code fragments may be loaded.

Eventual soundness for clients. Our approach is automatically eventually sound for client analyses that depend only on aliasing information and concrete types for visible program variables (e.g., callgraph resolution). In general, missing code can introduce unsoundness into the static information flow analysis beyond missing points-to edges. For example, consider

```
void main() { // program
  int val = source();                  int add(int x, int y) {  // library
  int valDup = add(val, 1);              return x+y; }
  sink(valDup); }
```

which calls the missing function `add`. Even with a sound points-to analysis, the static analysis would not recover the taint flow from `source` to `sink`. Sources and sinks in missing code must be specified, since there is no way to detect whether calling missing code leaks information out of the system or introduces sensitive information into the system.

In general, we can perform eventually sound analysis for clients that are abstract interpretations with finite abstract domain (at least, satisfying the ascending chain condition) [13], if the abstraction function α can be computed for values in the available code based on observations in the available code alone. In particular, for a call $y \leftarrow m(x)$, the concrete values of x and y are recorded. Then, we can construct a transfer function f_m to be analyzed in place of m. Initially, $\bot = f_m(\alpha(x))$ for all x; whenever a previously unobserved relation $\alpha(y) = f_m(\alpha(x))$ is detected during execution, a report is issued and f_m updated. Since the abstract domain is finite, only finitely many reports can be issued. Thus, the analysis is eventually sound. Finally, we use our points-to analysis to handle aliasing.

The challenge with information flow is that the abstraction function cannot be computed from observations in the available code alone, since information flow is a property of the computation, not just the input-output values. It may be possible to use techniques such as multi-execution [16], which keep a pair of values $\langle x_{\text{private}}, x_{\text{public}} \rangle$ for each (visible) program variable x, where x_{private} may depend on sensitive data whereas x_{public} does not. For example, the value for program variable `val` may be $\langle 14, 0 \rangle$, where 14 is a sensitive value and 0 is a public value. Then, we can execute `add` using both $\texttt{x} = 14$ and $\texttt{x} = 0$, and obtain return value $r_{\texttt{add}} = \langle 15, 1 \rangle$. Since these two values differ, we conclude that $r_{\texttt{add}}$ depends on the sensitive input 14, and report that `add` transfers information from its argument `x` to its return value $r_{\texttt{add}}$. Essentially, this approach transforms the program so the abstraction function becomes computable. Alternatively, existing techniques for specification inference such as [8] may be used to infer specifications describing how information flows through missing code.

10 Related Work

Program monitoring. There has been work using runtime checks to complement static analysis. For instance, [10] proposes to use dynamic information to resolve reflective call targets, and then instruments the program to report additional counterexamples. Similarly, [7] proposes to compute reachable code by inserting runtime checks to report counterexamples to optimistic assumptions, and [22] uses a combination of static type checking and runtime checks to enforce type safety. General systems using dynamic analysis to detect gaps in the static analysis, typically local invariants assumed true by the static analysis, have also been proposed [11, 15]. Our setting is far more challenging, because (i) naïve dynamic points-to analyses incur unreasonable overhead [38, 12], and (ii) we cannot instrument missing code.

Additionally, [29] uses dynamic information to complement static points-to analysis. However, their analysis is unreasonably imprecise for programs that make substantial use of native code, since they pessimistically assume returns from native code can point to arbitrary abstract objects. For demanding, whole-program clients such as static taint analysis, such imprecision generates a huge number of false positives, since every abstract object that leaks into missing code becomes aliased with every return value from missing code. Even with such coarse assumptions, their runtime overhead can be higher than 300%, which is not suitable for use in production code. In contrast, our analysis is both completely precise and incurs reasonable overhead.

There has also been work identifying bugs [30, 31, 33] and information leaks [6, 16, 18] by monitoring production executions. Our work similarly monitors production code to identify unsoundness that can be used to find bugs, information flows, and so forth, but our approach differs in that we aim to use the reported counterexamples to compute *static* points-to sets that are eventually sound; these points-to sets can be used with any client. Finally, there has been work on sound gradual typing [44, 41], which uses runtime monitoring to enforce given type specifications. The key difference between our approach and sound gradual typing is that the results of our static analysis improve as more counterexamples are detected.

Specification inference. There has been work on inferring specifications, e.g., purely static approaches that interact with a human analyst [52, 8, 1], and ones that rely on dynamic traces [12, 28, 4, 9]. Purely static approaches can give certain soundness guarantees, but suffer from imprecision and rely heavily on interaction. In contrast, dynamic approaches are fully automatic, but necessarily incomplete since dynamic analysis is an underapproximation. Our goal is to develop a fully automatic approach where runtime checks are used to detect when specifications are missing. Furthermore, [2] enables sound callgraph analysis using only information available in the library interface by using the *separate compilation assumption*, which says that the library can be compiled separately from the program. This assumption is similar to our disjoint fields assumption (with extensions to shared fields and callbacks) – we assume that the only information about the program “known” to the library are fields and methods that appear in the library interface. While the callgraph can be computed with reasonable precision using pessimistic assumptions, the same is not true of points-to edges.

Program synthesis has been used to infer specifications from dynamic traces [28]. This approach requires fine-grained instrumentation (specifically, leveraging features of the Javascript language to obtain alias traces), but they recover all method functionality. Their algorithm for doing so uses MCMC on a restricted space of potential specifications. Our approach requires significantly less instrumentation, but our goal is only to recover aliasing behaviors, and our specifications are furthermore flow insensitive. There have been other approaches to synthesizing programs from traces [32, 27]. See [28] for a detailed discussion.

Static analysis with specifications. A large number of static analyses rely on specifications to model missing code, including a number specifically designed to detect Android malware using information flow analysis [21, 12, 25], as well as production systems designed to find bugs in Android apps [19]. In all of these systems, specifications are implemented as needed for the most frequently used library functions; thus, specifications relevant to the client may be missing. Thus, **Optix** can be used in conjunction with these tools to detect potential unsoundness due to missing specifications.

Static points-to analysis. There is a large literature on static points-to analysis [43, 3, 50, 36, 49, 45]. Our focus is on the new problem of automatic inference of precise points-to information when some of the code is missing.

Static information flow analysis. Static information flow analysis has been used to verify of security policies [35, 51, 23, 47, 5, 21, 25]. These approaches all depend on alias analysis, and many use specifications to improve precision and scalability. Our techniques for automatically synthesizing points-to specifications can make implementing any static analysis for large software systems, including information flow analysis, more practical.

11 Conclusion

We have described an approach to points-to analysis when code is missing. Our approach is completely precise and fully automatic, and while it forgoes ahead-of-time soundness, it achieves eventual soundness by using runtime checks in production code. In particular, our approach is dynamically sound in the sense that unsoundness (if any) is detected at the point when it occurs, thus enabling the user to terminate execution to prevent any damage from happening. We implement our approach in a tool called **Optix** to compute points-to sets for Android apps, where the Android framework is missing. For efficiency, **Optix** assumes that library functions do not access global variables; we have empirically found that this assumption holds. With this assumption, **Optix** achieves low runtime overhead and data usage on almost all apps in a large benchmark suite.

References

1 Aws Albarghouthi, Isil Dillig, and Arie Gurfinkel. Maximal specification synthesis. In *POPL*, 2016.

2 Karim Ali and Ondřej Lhoták. Averroes: Whole-program analysis without the whole program. In *ECOOP*, 2013.

3 Lars Ole Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, University of Cophenhagen, 1994.

4 Steven Arzt and Eric Bodden. StubDroid: automatic inference of precise data-flow summaries for the android framework. In *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*, pages 725–735. IEEE, 2016.

5 Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *PLDI*, 2014.

6 Thomas H Austin and Cormac Flanagan. Multiple facets for dynamic information flow. In *POPL*, 2012.

7 Osbert Bastani, Saswat Anand, and Alex Aiken. Interactively verifying absence of explicit information flows in Android apps. *OOPSLA*, 2015.

8 Osbert Bastani, Saswat Anand, and Alex Aiken. Specification inference using context-free language reachability. In *POPL*, 2015.

9 Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. Active Learning of Points-To Specifications. In *PLDI*, 2018.

10 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 *ICSE*, 2011.

11 Maria Christakis, Peter Müller, and Valentin Wüstholz. Collaborative verification and testing with explicit assumptions. In *International Symposium on Formal Methods*, pages 132–146. Springer, 2012.

12 Lazaro Clapp, Saswat Anand, and Alex Aiken. Modelgen: mining explicit information flow specifications from concrete executions. In *ISSTA*, 2015.

13 Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, 1977.

14 Willem De Groef, Dominique Devriese, Nick Nikiforakis, and Frank Piessens. FlowFox: a web browser with flexible and precise information flow control. In *CCS*, 2012.

15 David Devecsery, Peter M Chen, Jason Flinn, and Satish Narayanasamy. Optimistic Hybrid Analysis: Accelerating Dynamic Analysis through Predicated Static Analysis. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 348–362. ACM, 2018.

16 Dominique Devriese and Frank Piessens. Noninterference through Secure Multi-execution. In *IEEE Symposium on Security and Privacy*, 2010.

17 Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. An empirical study of cryptographic misuse in android applications. In *CCS*, 2013.

18 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*, 2014.

19 Facebook. Adding models, 2017. URL: `http://fbinfer.com/docs/adding-models.html`.

20 Sascha Fahl, Marian Harbach, Thomas Muders, Lars Baumgärtner, Bernd Freisleben, and Matthew Smith. Why Eve and Mallory love Android: An analysis of Android SSL (in) security. In *CCS*, 2012.

21 Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken. Apposcopy: Semantics-based detection of android malware through static analysis. In *FSE*, 2014.

22 Cormac Flanagan. Hybrid type checking. In *POPL*, 2006.

23 Adam P Fuchs, Avik Chaudhuri, and Jeffrey S Foster. Scandroid: Automated security certification of android, 2009.

24 Google. UI/Application Exerciser Monkey, 2016. URL: `https://developer.android.com/studio/test/monkey.html`.

25 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 *NDSS*. Citeseer, 2015.

26 Ben Gruver. Smali project homepage, 2016. URL: `https://github.com/JesusFreke/smali`.

27 Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *POPL*, 2011.

28 Stefan Heule, Manu Sridharan, and Satish Chandra. Mimic: Computing models for opaque code. In *FSE*, 2015.

29 Martin Hirzel, Daniel Von Dincklage, Amer Diwan, and Michael Hind. Fast online pointer analysis. *TOPLAS*, 2007.

30 Wei Jin and Alessandro Orso. BugRedux: reproducing field failures for in-house debugging. In *ICSE*, 2012.

31 Wei Jin and Alessandro Orso. F3: fault localization for field failures. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pages 213–223. ACM, 2013.

32 Tessa Lau, Pedro Domingos, and Daniel S Weld. Learning programs from traces using version space algebra. In *International Conference on Knowledge Capture*, 2003.

33 Ben Liblit, Alex Aiken, Alice X Zheng, and Michael I Jordan. Bug isolation via remote program sampling. In *PLDI*, 2003.

34 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 Soundiness: A Manifesto. *CACM*, 2015.

35 V Benjamin Livshits and Monica S Lam. Finding Security Vulnerabilities in Java Applications with Static Analysis. In *Usenix Security*, 2005.

36 Ana Milanova, Atanas Rountev, and Barbara G Ryder. Parameterized object sensitivity for points-to and side-effect analyses for Java. In *Software Engineering Notes*, 2002.

37 Bimal Kumar Mishra and Navnit Jha. Fixed period of temporary immunity after run of anti-malicious software on computer nodes. *Applied Mathematics and Computation*, 2007.

38 Markus Mock, Manuvir Das, Craig Chambers, and Susan J Eggers. Dynamic points-to sets: A comparison with static analyses and potential applications in program understanding and optimization. In *PASTE*, 2001.

39 Patrick Mutchler, Yeganeh Safaei, Adam Doupé, and John C. Mitchell. Target Fragmentation in Android Apps. In *IEEE Security and Privacy Workshops*, 2016.

40 Mayur Naik, Alex Aiken, and John Whaley. *Effective static race detection for Java*. ACM, 2006.

41 Aseem Rastogi, Nikhil Swamy, Cédric Fournet, Gavin Bierman, and Panagiotis Vekris. Safe & efficient gradual typing for TypeScript. In *POPL*, volume 50, pages 167–180, 2015.

42 Andrei Sabelfeld and Andrew C Myers. Language-based information-flow security. *IEEE Journal on selected areas in communications*, 2003.

43 Olin Shivers. *Control-flow analysis of higher-order languages*. PhD thesis, Carnegie Mellon University, 1991.

44 Jeremy G Siek and Walid Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, volume 6, pages 81–92, 2006.

45 Manu Sridharan and Rastislav Bodík. Refinement-based context-sensitive points-to analysis for Java. In *PLDI*, 2006.

46 Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. Demand-driven points-to analysis for Java. In *ACM SIGPLAN Notices*, volume 40 (10), pages 59–76. ACM, 2005.

47 Omer Tripp, Marco Pistoia, Stephen J Fink, Manu Sridharan, and Omri Weisman. TAJ: effective taint analysis of web applications. In *PLDI*, 2009.

48 Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot-a Java bytecode optimization framework. In *Conference of the Centre for Advanced Studies on Collaborative Research*, 1999.

49 John Whaley and Monica S Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI*, 2004.

50 Robert P Wilson and Monica S Lam. Efficient context-sensitive pointer analysis for C programs. In *PLDI*, 1995.

51 Yichen Xie and Alex Aiken. Static Detection of Security Vulnerabilities in Scripting Languages. In *USENIX Security*, 2006.

52 Haiyan Zhu, Thomas Dillig, and Isil Dillig. Automated inference of library specifications for source-sink property verification. In *APLAS*, 2013.

How to Avoid Making a Billion-Dollar Mistake: Type-Safe Data Plane Programming with SafeP4

Matthias Eichholz
Technische Universität Darmstadt, Germany

Eric Campbell
Cornell University, Ithaca, NY, USA

Nate Foster
Cornell University, Ithaca, NY, USA

Guido Salvaneschi
Technische Universität Darmstadt, Germany

Mira Mezini
Technische Universität Darmstadt, Germany

Abstract

The P4 programming language offers high-level, declarative abstractions that bring the flexibility of software to the domain of networking. Unfortunately, the main abstraction used to represent packet data in P4, namely header types, lacks basic safety guarantees. Over the last few years, experience with an increasing number of programs has shown the risks of the unsafe approach, which often leads to subtle software bugs.

This paper proposes SafeP4, a domain-specific language for programmable data planes in which all packet data is guaranteed to have a well-defined meaning and satisfy essential safety guarantees. We equip SafeP4 with a formal semantics and a static type system that statically guarantees header validity – a common source of safety bugs according to our analysis of real-world P4 programs. Statically ensuring header validity is challenging because the set of valid headers can be modified at runtime, making it a dynamic program property. Our type system achieves static safety by using a form of path-sensitive reasoning that tracks dynamic information from conditional statements, routing tables, and the control plane. Our evaluation shows that SafeP4's type system can effectively eliminate common failures in many real-world programs.

2012 ACM Subject Classification Software and its engineering → Formal language definitions; Networks → Programming interfaces

Keywords and phrases P4, data plane programming, type systems

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2019.12

Related Version `https://arxiv.org/abs/1906.07223`

Funding This work has been co-funded by the German Research Foundation (DFG) as part of the Collaborative Research Center (CRC) 1053 MAKI and 1119 CROSSING, by the DFG projects SA 2918/2-1 and SA 2918/3-1, by the Hessian LOEWE initiative within the Software-Factory 4.0 project, by the German Federal Ministry of Education and Research and by the Hessian Ministry of Science and the Arts within CRISP, by the National Science Foundation under grants CNS-1413972 and CCF-1637532, and by gifts from InfoSys and Keysight.

33rd European Conference on Object-Oriented Programming (ECOOP 2019).
Editor: Alastair F. Donaldson; Article No. 12; pp. 12:1–12:28
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

> I couldn't resist the temptation to put in a null reference [...] This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.
>
> – Tony Hoare

Modern languages offer features such as type systems, structured control flow, objects, modules, etc. that make it possible to express rich computations in terms of high-level abstractions rather than machine-level code. Increasingly, many languages also offer fundamental safety guarantees – e.g., well-typed programs do not go wrong [23] – that make entire categories of programming errors simply impossible.

Unfortunately, although computer networks are critical infrastructure, providing the communication fabric that underpins nearly all modern systems, most networks are still programmed using low-level languages that lack basic safety guarantees. Unsurprisingly, networks are unreliable and remarkably insecure – e.g., the first step in a cyberattack often involves compromising a router or other network device [26, 19].

Over the past decade, there has been a shift to more flexible platforms in which the functionality of the network is specified in software. Early efforts related to software-defined networking (SDN) [21, 6], focused on the control plane software that computes routes, balances load, and enforces security policies, and modeled the data plane as a simple pipeline operating on a fixed set of packet formats. However, there has been recent interest in allowing the functionality of the data plane itself to be specified as a program – e.g., to implement new protocols, make more efficient use of hardware resources, or even relocate application-level functionality into the network [15, 14]. In particular, the P4 language [4] enables the functionality of a data plane to be programmed in terms of declarative abstractions such as header types, packet parsers, match-action tables, and structured control flow that a compiler maps down to an underlying target device.

Unfortunately, while a number of P4's features were clearly inspired by designs found in modern languages, the central abstraction for representing packet data – header types – lacks basic safety guarantees. To a first approximation, a P4 header type can be thought of as a record with a field for each component of the header. For example, the header type for an IPv4 packet, would have a 4-bit version field, an 8-bit time-to-live field, two 32-bit fields for the source and destination addresses, and so on.

According to the P4 language specification, an instance of a header type may either be valid or invalid: if the instance is valid, then all operations produces a defined value, but if it is invalid, then reading or writing a field yields an undefined result. In practice, programs that manipulate invalid headers can exhibit a variety of faults including dropping the packet when it should be forwarded, or even leaking information from one packet to the next. In addition, such programs are also not portable, since their behavior can vary when executed on different targets.

The choice to model the semantics of header types in an unsafe way was intended to make the language easier to implement on high-speed routers, which often have limited amounts of memory. A typical P4 program might specify behavior for several dozen different protocols, but any particular packet is likely to contain only a small handful of headers. It follows that if the compiler only needs to represent the valid headers at run-time, then memory requirements can be reduced. However, while it may have benefits for language implementers, the design is a disaster for programmers – it repeats Hoare's "mistake," and bakes an unsafe feature deep into the design of a language that has the potential to become the de-facto standard in a multi-billion-dollar industry.

This paper investigates the design of a domain-specific language for programmable data planes in which all packet data is guaranteed to have a well-defined meaning and satisfy basic safety guarantees. In particular, we present SafeP4, a language with a precise semantics and a static type system that can be used to obtain guarantees about the validity of all headers read or written by the program. Although the type system is mostly based on standard features, there are several aspects of its design that stand out. First, to facilitate tracking dependencies between headers – e.g. if the TCP header is valid, then the IPv4 will also be valid – SafeP4 has an expressive algebra of types that tracks validity information at a fine level of granularity. Second, to accommodate the growing collection of extant P4 programs with only modest modifications, SafeP4 uses a path-sensitive type system that incorporates information from conditional statements, forwarding tables, and the control plane to precisely track validity.

To evaluate our design for SafeP4, we formalized the language and its type system in a core calculus and proved the usual progress and preservation theorems. We also implemented the SafeP4 type system in an OCaml prototype, P4Check, and applied it to a suite of open-source programs found on GitHub such as `switch.p4`, a large P4 program that implements the features found in modern data center switches (specifically, it includes over four dozen different switching, routing, and tunneling protocols, as well as multicast, access control lists, among other features). We categorize common failures and, for programs that fail to type-check, identify the root causes and apply repairs to make them well-typed. We find that most programs can be repaired with low effort from programmers, typically by applying a modest number of simple repairs.

Overall, the main contributions of this paper are as follows:

- We propose SafeP4, a type-safe enhancement of the P4 language that eliminates all errors related to header validity.
- We formalize the syntax and semantics of SafeP4 in a core calculus and prove that the type system is sound.
- We implement our type checker in an OCaml prototype, P4Check.
- We evaluate our type system empirically on over a dozen real-world P4 programs and identify common errors and repairs.

The rest of this paper is organized as follows. Section 2 provides a more detailed introduction to P4 and elaborates on the problems this work addresses. Section 3 presents the design, operational semantics and type system of SafeP4 and reports our type safety result. The results of evaluating SafeP4 in the wild are presented in Section 4. Section 5 surveys related work and Section 6 summaries the paper and outlines topics for future work.

2 Background and Problem Statement

This section introduces the main features of P4 and highlights the problems caused by the unsafe semantics for header types.

2.1 P4 Language

P4 is a domain-specific language designed for processing packets – i.e., arbitrary sequences of bits that can be divided into (i) a set of pre-determined *headers* that determine how the packet will be forwarded through the network, and (ii) a *payload* that encodes application-level data. P4 is designed to be protocol-independent, which means it handles both packets with standard header formats (e.g., Ethernet, IP, TCP, etc.) as well as packets with custom header formats defined by the programmer. Accordingly, a P4 program first *parses* the headers in the input packet into a typed representation. Next, it uses a *match-action pipeline*

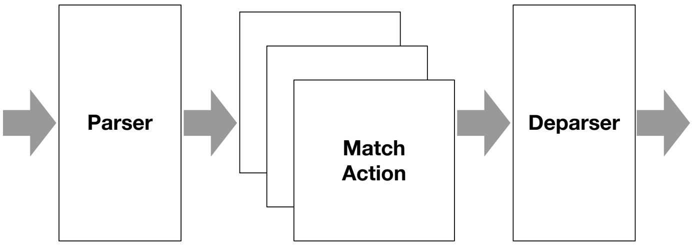


Figure 1 Abstract forwarding model.

to compute a transformation on those headers – e.g., modifying fields, adding headers, or removing them. Finally, a *deparser* serializes the headers back into into a packet, which can be output to the next device. A depiction of this abstract forwarding model is shown in Figure 1.

The match-action pipeline relies on a data structure called a *match-action table*, which encodes conditional processing. More specifically, the table first looks up the values being tested against a list of possible entries, and then executes a further snippet of code depending on which entry (if any) matched. However, unlike standard conditionals, the entries in a match-action table are not known at compile-time. Rather, they are inserted and removed at run-time by the control plane, which may be logically centralized (as in a software-defined network), or it may operate as a distributed protocol (as in a conventional network).

The rest of this section describes P4's typed representation, how the parsers, and deparsers convert between packets and this typed representation, and how control flows through the match-action pipeline.

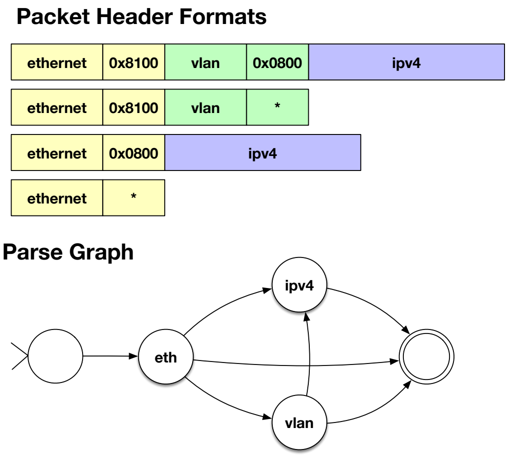
Header Types and Instances. Header types specify the internal representation of packet data within a P4 program. For example, the first few lines of the following snippet of code:

```
header_type ethernet_t {
  fields {
    dstAddr: 48;
    srcAddr: 48;
    etherType: 16;
  }
}
header ethernet_t ethernet;
header ethernet_t inner_ethernet;
```

declare a type (`ethernet_t`) for the Ethernet header with fields `dstAddr`, `srcAddr`, and `etherType`. The integer literals indicate the bit width of each field. The next two lines declare two `ethernet_t` instances (`ethernet` and `inner_ethernet`) with global scope.

Parsers. A P4 parser specifies the order in which headers are extracted from the input packet using a simple abstraction based on finite state machines. Extracting into an header instance populates its fields with the requisite bits of the input packet and marks the instance as valid. Figure 2 depicts a visual representation of a parse graph for three common headers: Ethernet, VLAN, and IPv4. The instance `ethernet` is extracted first, optionally followed by a `vlan` instance, or an `ipv4` instance, or both.

Tables and Actions. The bulk of the processing for each packet in a P4 program is performed using match-action tables that are populated by the control plane. A table (such as the one in Figure 3) is defined in terms of (i) the data it `reads` to determine a matching entry (if any), (ii) the `actions` it may execute, and (iii) an optional `default_action` it executes if no matching entry is found.



```
parser start {
  return parse_eth;
}
parser parse_eth {
 extract(ethernet);
 return select(latest.etherType){
   0x8100 : parse_vlan;
   0x0800 : parse_ipv4;
   default: ingress;
  }
}
parser parse_vlan {
  extract vlan {
  return select(latest.etherType){
    0x0800: parse_ipv4;
    default: ingress;
  }
}
parser parse_ipv4 {
  extract(ipv4);
  return ingress;
}
```

Figure 2 (Left) Header formats and parse graph that extracts an Ethernet header optionally followed by VLAN and/or IPv4 headers. (Right) P4 code implementing the same parser.

```
table forward {
 reads {
  ipv4 : valid;
  vlan : valid;
  ipv4.dstAddr: ternary;
 }
 actions = {
  nop;
  next_hop;
  remove;
 }
 default_action : nop();
}
```

Runtime Contents of forward

Pattern			Action	
ipv4	vlan	ipv4.dstAddr	Name	Data
1	0	10.0.0.*	next_hop	s, d
0	1	*	remove	

Figure 3 P4 tables. `forward` reads the validity of the `ipv4` and `vlan` header instances and the `dstAddr` field of the `ipv4` header instance, and calls one of its actions: `nop`, `next_hop`, or `remove`.

The behavior of a table depends on the entries installed at run-time by the control-plane. Each table entry contains a match pattern, an action, and action data. Intuitively, the match pattern specifies the bits that should be used to match values, the action is the name of a pre-defined function (such as the ones in Figure 4), and the action data are the arguments to that function. Operationally, to process a packet, a table first scans its entries to locate the first matching entry. If such a matching entry is found, the packet is said to "hit" in the table, and the associated action is executed. Otherwise, if no matching entry is found, the packet is said to "miss" in the table, and the `default_action` (which is a no-op if unspecified) is executed.

A table also specifies the *match-kind* that describes how each header field should match with the patterns provided by the control plane. In this paper, we focus our attention on `exact`, `ternary`, and `valid` matches. An `exact` match requires the bits in the packet be exactly equivalent to the bits in the controller-installed pattern. A `ternary` match allows wildcards in arbitrary positions, so the controller-installed pattern `0*` would match bit sequences `00` and `01`. A `valid` match can only be applied to a header instance and simply checks the validity bit of that instance.

```
action next_hop(src, dst) {
 modify_field(ethernet.srcAddr, src);
 modify_field(ethernet.dstAddr, dst);
 subtract_from_field(ipv4.ttl, 1);
}
```

```
action remove() {
 modify_field(
   ethernet.etherType,
   vlan.etherType);
 remove_header(vlan);
}
```

Figure 4 P4 actions.

For example, in Figure 3, the `forward` table is shown populated with two rules. The first rule tests whether `ipv4` is valid, `vlan` is invalid, and the first 24 bits of `ipv4.srcAddr` equal `10.0.0`, and then applies `next_hop` with arguments s and d (which stand for source and destination addresses). The second rule checks that `ipv4` is invalid, then that `vlan` is valid, and skips evaluating the value of `ipv4.dstAddr` (since it is wildcarded), to finally apply the `remove` action.

Actions are functions containing sequences of primitive commands that perform operations such as adding and removing headers, assigning a value to a field, adding one field to another, etc. For example, Figure 4 depicts two actions: the `next_hop` action updates the Ethernet source and destination addresses with action data from the controller; and the `remove` action copies EtherType field from the `vlan` header instance to the `ethernet` header instance and invalidates the `vlan` header.

Control. A P4 control block can use standard control-flow constructs to execute a pipeline of match-action tables in sequence. They manage the order and conditions under which each table is executed. The `ingress` control block begins to execute as soon as the parser completes. The `apply` command executes a table and conditionals branch on a boolean expression such as the validity of a header instance.

```
control ingress {
 if(valid(ipv4) or valid(vlan)) {
  apply(forward);
 }
}
```

The above code applies the `forward` table if one of `ipv4` or `vlan` is valid.

Deparser. The deparser reassembles the final output packet, after all processing has been done by serializing each valid header instance in some order. In P4$_{14}$, the version of P4 we consider in this paper, the compiler automatically generates the deparser from the parser – i.e., for our example program, the deparser produces a packet with Ethernet, VLAN (if valid), and IPv4 (if valid), in that order.

2.2 Common Bugs in P4 Programs

Having introduced the basic features of P4, we now present five categories of bugs found in open-source programs that arise due to reading and writing invalid headers – the main problem that SafeP4 addresses. There is one category for each of the following syntactic constructs: (1) parsers, (2) controls, (3) table reads, (4) table actions, and (5) default actions.

To identify the bugs we surveyed a benchmark suite of 15 research and industrial P4 programs that are publicly available on GitHub and compile to the BMv2 [25] backend.

```
/* UNSAFE */



parser parse_ethernet {
 extract(ethernet);
 return select(ethernet.etherType)
      {
  0x0800 : parse_ipv4;
  default : ingress;

 }
}
parser parse_ipv4 {
 extract(ipv4);
 return select(ipv4.protocol) {
  6 : parse_tcp;
  default : ingress;

 }
}
```

```
/* SAFE */
parser_exception unsupported {
  parser_drop;
}
parser parse_ethernet {
 extract(ethernet);
 return select(ethernet.etherType)
      {
  0x0800 : parse_ipv4;
  default :
   parser_error unsupported;
  }
}
parser parse_ipv4 {
 extract(ipv4);
 return select(ipv4.protocol) {
 6 : parse_tcp;
 default : parser_error
     unsupported;
  }
}
```

```
parser parse_tcp {
  extract(tcp);
  return ingress;
}
```

```
control ingress {
  if(tcp.syn == 1 and ...){...}
}
```

Figure 5 Left: unsafe code in NetHCF; Right: our type-safe fix; Bottom: common code.

Later, in Section 4, we will report the number of occurrences of each of these categories in our benchmark suite detected by our approach.[1]

2.2.1 Parser Bugs

The first class of errors is due to the parser being too conservative about dropping malformed packets, which increases the set of headers that may be invalid in the control pipeline. In most programs, the parser chooses which headers to `extract` based on the fields of previously-extracted headers using P4's version of a switch statement, `select`. Programmers often fail to handle packets fall through to the `default` case of these `select` statements.

An example from the NetHCF [34, 2] codebase illustrates this bug. NetHCF is a research tool designed to combat TCP spoofing. As shown in Figure 5, the parser handles TCP packets in `parse_ipv4` and redirects all other packets to the `ingress` control. Unfortunately, the `ingress` control (bottom right) does not check whether `tcp` is valid before accessing `tcp.syn` to check whether it is equal to `1`. This is unsafe since `tcp` is not guaranteed to be valid even though it is required to be valid in the `ingress` control.

To fix this bug, we can define a parser exception, `unsupported`, with an handler that drops packets, thereby protecting the `ingress` from having to handle unexpected packets. Note however, that this fix might not be the best solution, since it alters the original behavior of the program. However, without knowing the programmer's intention, it is generally not possible to automatically repair a program with undefined behavior.

[1] We focus on $P4_{14}$ programs in this paper, but the issues we address also persist in the latest version of the language, $P4_{16}$. We did not consider $P4_{16}$ in this paper due to the smaller number of programs currently available.

```
/* UNSAFE */
control ingress {

 process_cache();
 process_value();

 apply(ipv4_route);
}
```

```
/* SAFE */
control ingress {
  if(valid(nc_hdr)) {
    process_cache();
    process_value();
 }
 apply(ipv4_route);
}
```

```
control process_cache {
    apply(check_cache_exist);
    ...
}
```

```
table check_cache_exist {
    reads { nc_hdr.key }
    actions { ... }
}
```

Figure 6 Left: unsafe code in NetCache; Right: our type-safe fix; Bottom: Common code.

2.2.2 Control Bugs

Another common bug occurs when a table is executed in a context in which the instances referenced by that table are not guaranteed to be valid. This bug can be seen in the open-source code for NetCache [13, 15], a system that uses P4 to implement a load-balancing cache. The parser for NetCache reserves a specific port (8888) to handle its special-purpose traffic, a condition that is built into the parser, which extracts `nc_hdr` (i.e., the NetCache-specific header) only when UDP traffic arrives from port 8888. Otherwise, it performs standard L2 and L3 routing. Unfortunately, the `ingress` control node (Figure 6) tries to access `nc_hdr` before checking that it is valid. Specifically, the `reads` declaration for the `check_cache_exists` table, which is executed first in the `ingress` pipeline, presupposes that `nc_hdr` is valid. The invocation of the `process_value` table (not shown) contains another instance of the same bug.

To fix these bugs, we can wrap the calls to `process_cache` and `process_value` in an conditional that checks the validity of the header `nc_hdr`. This ensures that `nc_hdr` is valid when `process_cache` refers to it.

2.2.3 Table Reads Bugs

A similar bug arises in programs that contain tables that first match on the validity of certain header instances before matching on the fields of those instances. The advantage of this approach is that multiple types of packets can be processed in a single table, which saves memory. However, if implemented incorrectly, this programming pattern can lead to a bug, in which the `reads` declaration matches on bits from a header that may not be valid!

The `switch.p4` program exhibits an exemplar of this bug; it is a "realistic production switch" developed by Barefoot Networks, meant to be used "as-is, or as a starting point for more advanced switches" [18].

An archetypal example of table reads bugs is the `port_vlan_mapping` table of `switch.p4` (Figure 7). This table is invoked in a context where it is not known which of the VLAN tags is valid, despite containing references to both `vlan_tag_[0]` and `vlan_tag_[1]` in the `reads` declaration. Adroitly, the programmer has guarded the references to `vlan_tag_[`i`].vid` with keys that test the validity of `vlan_tag_[`i`]`, for $i = 1, 2$. Unfortunately, as written, it is impossible for the control plane to install a rule that will always avoid reading the value of an invalid header. The first match will check whether the `vlan_tag_[0]` instance is invalid, which is safe. However, the very next match will try to read the value of the `vlan_tag_[0].vid` field, even when the instance is invalid! This attempt to access an invalid header results in undefined behavior, and is therefore a bug.

```
/* UNSAFE */
table port_vlan_mapping {
 reads {
  vlan_tag_[0] : valid;
  vlan_tag_[0].vid : exact;
  vlan_tag_[1] : valid;
  vlan_tag_[1].vid : exact;
 } ...
}
```

```
/* SAFE */
table port_vlan_mapping {
 reads {
  vlan_tag_[0] : valid;
  vlan_tag_[0].vid : ternary;
  vlan_tag_[1] : valid;
  vlan_tag_[1].vid : ternary;
 } ...
}
```

Figure 7 Left: a table in `switch.p4` with unprotected conditional reads; Right: our type-safe fix.

It is worthy to note that this code is not actually buggy on some targets – in particular, on targets where invalid headers are initialized with `0`. However, `0`-initialization is not prescribed by the language specification, and therefore this code is not portable across other targets.

The naive solution to fix this bug is to refactor the table into four different tables (one for each combination of validity bits) and then check the validity of each header before the tables are invoked. While this fix is perfectly safe, it can result in a combinatorial blowup in the number of tables, which is clearly undesirable both for efficiency reasons and because it requires modifying the control plane.

Fortunately, rather than factoring the table into four tables, we can replace the `exact` match-kinds with `ternary` match-kinds, which permit matching with wildcards. In particular, the control plane can install rules that match invalid instances using an all-wildcard patterns, which is safe.

In order for this solution to typecheck, we need to assume that the control plane is well-behaved – i.e. that it will install wildcards for the `ternary` matches whenever the header is invalid. In our implementation, we print a warning whenever we make this kind of assumption so that the programmer can confirm that the control plane is well-behaved.

2.2.4 Table Action Bugs

Another prevalent bug, in our experience, arises when distinct actions in a table require different (and possible mutually exclusive) headers to be valid. This can lead to two problems: (i) the control plane can populate the table with unsafe match-action rules, and (ii) there may be no validity checks that we can add to the control to make all of the actions typecheck.

The `fabric_ingress_dst_lkp` table (Figure 8) in `switch.p4` provides an example of this misbehavior. The `fabric_ingress_dst_lkp` table reads the value of `fabric_hdr.dstDevice` and then invokes one of several actions: `term_cpu_packet`, `term_fabric_unicast_packet`, or `term_fabric_multicast_packet`. Respectively, these actions require the `fabric_hdr_cpu`, `fabric_hdr_unicast`, and `fabric_hdr_multicast` (respectively) headers to be valid. Unfortunately the validity of these headers is mutually exclusive.[2]

Since `fabric_hdr_cpu`, `fabric_hdr_unicast`, and `fabric_hdr_multicast` are mutually exclusive, there is no single context that makes this table safe. The only facility the table provides to determine which action should be called is `fabric_hdr.dstDevice`. However, the P4 program doesn't establish a relationship between the value of `fabric_hdr.dstDevice` and the validity of any of these three header instances. So, the behavior of this table is only well-defined when the input packets are well-formed, an unreasonable expectation for real switches, which may receive *any* sequence of bits "on the wire."

[2] There are other actions in the real `fabric_ingress_dst_lkp`, but these three actions demonstrate the core of the problem.

```
/* UNSAFE */
table fabric_ingress_dst_lkp {
 reads {
  fabric_hdr.dstDevice : exact;
 }



 actions {
  term_cpu_packet;
  term_fabric_unicast_packet;
  term_fabric_multicast_packet;
 }
}
```

```
/* SAFE */
table fabric_ingress_dst_lkp {
 reads {
  fabric_hdr.dstDevice : exact;
  fabric_hdr_cpu : valid;
  fabric_hdr_unicast: valid;
  fabric_hdr_multicast: valid;
 }
 actions {
  term_cpu_packet;
  term_fabric_unicast_packet;
  term_fabric_multicast_packet;
 }
}
```

Figure 8 Left: unsafe code in `switch.p4`; Right: our type-safe fix.

We fix this bug by including validity matches in the `reads` declaration, as shown in Figure 8. As in Section 2.2.3, this solution avoids combinatorial blowup and extensive control plane refactoring.

In order to type-check this solution, we need to make an assumption about the way the control plane will populate the table. Concretely, if an action a only typechecks if a header h is valid, and h is not necessarily valid when the table is applied, we assume that the control plane will only call a if h is matched as valid. For example, `fabric_hdr_cpu` is not known to be valid when (the fixed version of) `fabric_ingress_dst_lkp` is applied, so we assume that the control plane will only call action `term_cpu_packet` when `fabric_hdr_cpu` is matched as valid. Again, our implementation prints these assumptions as warnings to the programmer, so they can confirm that the control plane will satisfy these assumptions.

2.2.5 Default Action Bugs

Finally, the *default action* bugs occur when the programmer incorrectly assumes that a table performs some action when a packet misses. The NetCache program (described in Section 2.2.2) exhibits an example of this bug, too. The bug is shown in Figure 9, where the table `add_value_header_1` is expected to make the `nc_value_1` header valid, which is done in the `add_value_header_1_act` action. The control plane may refuse to add any rules to the table, which would cause all packets to miss, meaning that the `add_value_header_1_act` action would never be called and `nc_value_1` may not be valid. To fix this error, we simply set the default action for the table to `add_value_header_1_act`, which will force the table to remove the header no matter what rules the controller installs.

```
/* UNSAFE */
table add_value_header_1 {
 actions {
  add_value_header_1_act;
 }

}
```

```
/* SAFE */
table add_value_header_1 {
 actions {
  add_value_header_1_act;
 }
 default_action :
    add_value_header_1_act();
}
```

Figure 9 Left: unsafe code in NetCache; Right: our type-safe fix.

```
if(ethernet.etherType == 0x0800) {
 apply(ipv4_table);
} else if(ethernet.etherType == 0x086DD) {
 apply(ipv6_table);
}
```

```
if(valid(ipv4)) {
 apply(ipv4_table);
} else if(valid(ipv6)) {
 apply(ipv6_table);
}
```

Figure 10 Left: data-dependent header validation; Right: syntactic header validation.

2.3 A Typing Discipline to Eliminate Invalid References

In this paper, we propose a type system to increase the safety of P4 programs by detecting and preventing the classes of bugs defined in Section 2.2. These classes of bugs all manifest when a program attempts to access an invalid header – differentiating themselves only in their syntactic provenance. The type system that we present in the next section uses a path-sensitive analysis, coupled with occurrence typing [32], to keep track of which headers are guaranteed to be available at any program point – rejecting programs that reference headers that *might* be uninitialized – thus, preventing all references to invalid headers.

Of course, in general, the problem of deciding header-validity can depend on arbitrary data, so a simple type system cannot hope to fully determine all scenarios when an instance will be valid. Indeed, programmers often use a variety of data-dependent checks to ensure safety. For instance, the control snippet shown on the left-hand side of Figure 10 will not produce undefined behavior, given a parser that chooses between parsing an `ipv4` header when `ethernet.etherType` is `0x0800`, an `ipv6` header when `ethernet.etherType` is `0x86DD`, and throws a parser error otherwise.

While this code is safe in this very specific context, it quickly becomes unsafe when ported to other contexts. For example in `switch.p4`, which performs tunneling, the egress control node copies the `inner_ethernet` header into the `ethernet`; however the `inner_ethernet` header may not be valid at the program point where the copy is performed. This behavior is left undefined [7], a target is free to read arbitrary bits, in which case it could decide to call the `ipv4_table` despite `ipv4` being invalid.

To improve the maintainability and portability of the code, we can replace the data-dependent checks with validity checks, as illustrated by the control snippet shown on the right-hand side of Figure 10. The validity checks assert precisely the preconditions for calling each table, so that no matter what context this code snippet is called in, it is impossible for the `ipv4_table` to be called when the `ipv4` header is invalid.

In the next section, we develop a core calculus for SafeP4 with a type system that eliminates references to invalid headers, encouraging programers to replace data-dependent checks with header-validity checks.

3 SafeP4

This section discusses our design goals for SafeP4 and the choices we made to accommodate them, and formalizes the language's syntax, small-step semantics, and type system.

3.1 Design

Our primary design goal for SafeP4 is to develop a core calculus that models the main features of P4_{14} and P4_{16}, while guaranteeing that all data from packet headers is manipulated in a safe and well-defined manner. We draw inspiration from Featherweight Java [12] – i.e., we model the essential features of P4, but prune away unnecessary complexity. The result is a minimal calculus that is easy to reason about, but can still express a large number of real-world data plane programs. For instance, P4 and SafeP4 both achieve protocol independence by allowing the programmer to specify the types of packet headers and their order in the bit stream. Similarly, SafeP4 mimics P4's use of tables to interface with the control-plane and decide which actions to execute at run-time.

So what features does SafeP4 prune away? We omit a number of constructs that are secondary to how packets are processed – e.g., `field_list_calculations`, `parser_exceptions`, `counters`, `meters`, `action profiles`, etc. It would be relatively straightforward to add these to the calculus – indeed, most are already handled in our prototype – at the cost of making it more complicated. We also modify or distill several aspects of P4. For instance, P4 separates the parsing phase and the control phase. Rather than unnecessarily complicating the syntax of SafeP4, we allow the syntactic objects that represent parsers and controls to be freely mixed. We make a similar simplification in actions, informally enforcing which primitive commands can be invoked within actions (e.g., field modification, but not conditionals).

Another challenge arises in trying to model core behaviors of both $P4_{14}$ and $P4_{16}$, in that they each have different type systems and behaviors for evaluating expressions. Our calculus abstracts away expression typing and syntax variants by assuming that we are given a set of constants k that can represent values like `0` or `True`, or operators such as `&&` and `?:`. We also assume that these operators are assigned appropriate (i.e., sound) types. With these features in hand, one can instantiate our type system over arbitrary constants.

Another departure from P4 is related to the *add* command, which presents a complication for our expression types. The analogous `add_header` action in $P4_{14}$ simply modifies the validity bit, without initializing any of the fields. This means that accessing any of the header fields before they have been manually initialized reads a non-deterministic value. Our calculus neatly sidesteps this issue by defining the semantics of the $add(h)$ primitive to initialize each of the fields of h to a default value. We assume that along with our type constants there is a function *init* that accepts a header type η and produces a header instance of type η with all fields set to their default value. Note that we could have instead modified our type system to keep track of the definedness of header fields as well as their validity. However, for simplicity we choose to focus on header validity in this paper.

The portion of our type system that analyzes header validity, requires some way of keeping track of which headers are valid. Naively, we can keep track of a set of which headers are guaranteed to be valid on all program paths, and reject programs that reference headers not in this set. However, this coarse-grained approach would lead to a large number of false positives. For instance, the parser shown in Figure 2 parses an `ethernet` header and then either boots to `ingress` or parses an `ipv4` header and then either proceeds to the `ingress` or parses an `vlan` header. Hence, at the `ingress` node, the only header that is guaranteed to be valid is the `ethernet` header. However, it is certainly safe to write an `ingress` program that references the `vlan` header after checking it was valid. To reflect this in the type system we introduce a special construct called $valid(h)\ c_1\ else\ c_2$, which executes c_1 if h is valid and c_2 otherwise. When we type check this command, following previous work on occurrence typing [32], we check c_1 with the additional fact that h is valid, and we check c_2 with the additional fact that h is not valid.

Even with this enhancement, this type system would still be overly restrictive. To see why, let us augment the parser from Figure 2 with the ability to parse TCP and UDP packets: after parsing the `ipv4` header, the parser can optionally extract the `vlan`, `tcp`, or `udp` header and then boot control flow to ingress. Now suppose that we have a table `tcp_table` that refers to both `ipv4` and `tcp` in its `reads` declaration, and that `tcp_table` is (unsafely) applied immediately in the `ingress`. Because the validity of `tcp` implies the validity of `ipv4`, it should be safe to check the validity of `tcp` and then apply `tcp_table`. However, using the representation of valid headers as a set, we would need to ascertain the validity of `ipv4` and of `tcp`.

To solve this problem, we enrich our type representation to keep track of dependencies between headers. More specifically, rather than representing all headers guaranteed to be valid in a set, we use a finer-grained representation – a set of sets of headers that might be valid at the current program point. For a given header reference to be safe, it must to be a member of all possible sets of headers – i.e., it must be valid on all paths through the program that reach the reference.

Overall, the combination of an expressive language of types and a simple version of occurrence typing allows us to capture dependencies between headers and perform useful static analysis of the dynamic property of header validity.

The final challenge with formally modelling P4 lies in its interface with the control-plane, which populates the tables and provides arguments to the actions. While the control-plane's only methodology for managing switch behavior is to populate the match-action tables with forwarding entries, it is perfectly capable of producing undefined behavior. However, if we assume that the controller is well-intentioned, we can prove the safety of more programs.

In our formalization, to streamline the presentation, we model the control plane as a function $\mathcal{CA}(t, H) = (a_i, \bar{v})$ that takes in a table t and the current headers H and produces the action to call a_i and the (possibly empty) action data arguments $\bar{v}$. We also use a function $\mathcal{CV}(t) = \bar{S}$ that analyzes a table t and produces a list of sets of valid headers $\bar{S}$, one set for each action, that can be safely assumed valid when the entries are populated by the control plane. From the table declaration and the header instances that can be assumed valid, based on the match-kinds, we can derive a list of match key expressions $\bar{e}$ that must be evaluated when the table is invoked. Together, these functions model the run-time interface between the switch and the controller. In order to prove progress and preservation, we assume that $\mathcal{CV}$ and $\mathcal{CA}$ satisfy three simple correctness properties: (1) the control plane can safely install table entries that never read invalid headers, (2) the action data provided by the control plane has the types expected by the action, and (3) the control plane will only assume valid headers for an action that are valid for a given packet. See technical report for details.

3.2 Syntax

The syntax of SafeP4 is shown in Figure 11. To lighten the notation, we write $\bar{x}$ as shorthand for a (possibly empty) sequence $x_1, ..., x_n$.

A SafeP4 program consists of a sequence of declarations $\bar{d}$ and a command c. The set of declarations includes header types, header instances, and tables. Header type declarations describe the format of individual headers and are defined in terms of a name and a sequence of field declarations. The notation "$f : \tau$" indicates that field f has type τ. We let η range over header types. A header instance declaration assigns a name h to a header type η. The map $\mathcal{HT}$ encodes the (global) mapping between header instances and header types. Table declarations $t(\bar{h}, \overline{(e,m)}, \bar{a})$, are defined in terms of a sequence of valid-match header instances $\bar{h}$, a sequence of match-key expressions $\overline{(e,m)}$ read in the table, where e is an expression and m is the match-kind used to match this expression, and a sequence of actions $\bar{a}$. The notation $t.valids$ denotes the valid-match instances, $t.reads$ denotes the expressions, and $t.actions$ denotes the actions.

Actions are written as (uncurried) λ-abstractions. An action $\lambda\bar{x}.\ c$ declares a (possibly empty) sequence of parameters, drawn from a fresh set of names, which are in scope for the command c. The run-time arguments for actions (action data) are provided by the control

Commands

$$\begin{array}{lll} c ::= & & \\ \mid & \mathit{extract}(h) & \text{EXTRACTION} \\ \mid & \mathit{emit}(h) & \text{DEPARSING} \\ \mid & c_1 \,;\, c_2 & \text{SEQUENCE}^* \\ \mid & \mathit{if}(e)\ c_1\ \mathit{else}\ c_2 & \text{CONDITIONAL} \\ \mid & \mathit{valid}(h)\ c_1\ \mathit{else}\ c_2 & \text{VALIDITY} \\ \mid & t.\mathit{apply}() & \text{APPLICATION} \\ \mid & \mathit{skip} & \text{SKIP} \\ \mid & \mathit{add}(h) & \text{ADDITION}^* \\ \mid & \mathit{remove}(h) & \text{REMOVAL}^* \\ \mid & h.f = e & \text{MODIFICATION}^* \end{array}$$

Actions

$$\begin{array}{ll} a ::= \lambda \bar{x}.c & \text{ACTION} \end{array}$$

Expressions

$$\begin{array}{lll} e ::= & & \\ \mid & v & \text{VALUES} \\ \mid & h.f & \text{HEADER FIELD} \\ \mid & x & \text{VARIABLE} \\ \mid & k^n & \text{CONSTANT} \end{array}$$

Declarations

$$\begin{array}{lll} d ::= & & \\ \mid & t(\bar{h}, \overline{(e,m)}, \bar{a}) & \text{TABLE} \\ \mid & \eta\ \{\overline{f : \tau}\} & \text{HEADER TYPE} \\ \mid & h \mapsto \eta & \text{INSTANTIATION} \end{array}$$

Match Kinds

$m \in \{\mathit{exact}, \mathit{ternary}\}$

Constants

$k \in K$

Program

$\mathcal{P} ::= (\bar{d}, c)$

Values

$v \in V$

Header Types

$$\begin{array}{lll} \Theta ::= & & \\ \mid & 0 & \text{CONTRADICTION} \\ \mid & 1 & \text{EMPTY} \\ \mid & h & \text{INSTANCE} \\ \mid & \Theta_1 \cdot \Theta_2 & \text{CONCATENATION} \\ \mid & \Theta_1 + \Theta_2 & \text{CHOICE} \end{array}$$

Action Types

$\alpha ::= \bar{\tau} \to \Theta$

Expression Types

$$\begin{array}{ll} \tau ::= & \texttt{Bool} \\ \mid & \bar{\tau} \to \tau \\ \mid & \ldots \end{array}$$

Figure 11 Syntax of SafeP4.

plane. Note that we artificially restrict the commands that can be called in the body of the action to addition, removal, modification and sequence; these actions are identified with an asterisk in Figure 11.

The calculus provides commands for extracting (*extract*), creating (*add*), removing (*remove*), and modifying ($h.f = e$) header instances. The *emit* command is used in the deparser and serializes a header instance back into a bit sequence (*emit*). The *if*-statement conditionally executes one of two commands based on the value of a boolean condition. Similarly, the *valid*-statement branches on the validity of h. Table application commands ($t.\mathit{apply}()$) are used to invoke a table t in the current state. The *skip* command is a no-op.

The only built-in expressions in SafeP4 are variables x and header fields, written $h.f$. We let v range over values and assume a collection of n-ary constant operators $k^n \in K$.

For simplicity, we assume that every header referenced in an expression has a corresponding instance declaration. We also assume that header instance names h, header type names η, variable names x, and table names t are drawn from disjoint sets of names H,E,V, and T respectively and that each name is declared only once.

3.3 Type System

SafeP4 provides two main kinds of types, basic types τ and header types Θ as shown in Figure 11. We assume that the set of basic types includes booleans (for conditionals) as well as tuples and function types (for actions).

$$\begin{aligned}
[\![\Theta]\!] &\subseteq \mathcal{P}(\mathit{Header}) \\
[\![0]\!] &= \{\} \\
[\![1]\!] &= \{\{\}\} \\
[\![h]\!] &= \{\{h\}\} \\
[\![\Theta_1 \cdot \Theta_2]\!] &= [\![\Theta_1]\!] \bullet [\![\Theta_2]\!] \\
[\![\Theta_1 + \Theta_2]\!] &= [\![\Theta_1]\!] \cup [\![\Theta_2]\!]
\end{aligned}$$

$$\begin{aligned}
\mathcal{F}(h, f_i) &= \tau_i && \textit{Field lookup} \\
\mathcal{A}(a) &= \lambda\bar{x} : \bar{\tau}.\ c && \textit{Action lookup} \\
\mathcal{CA}(t, H) &= (a_i, \bar{v}) && \textit{Control-plane actions} \\
\mathcal{CV}(t) &= \bar{S} && \textit{Control-plane validity} \\
\mathcal{H}(e) &= \bar{h} && \textit{Referenced Header instances}
\end{aligned}$$

$$\begin{aligned}
\mathsf{maskable}(t, e, \mathit{exact}) &\triangleq \mathit{false} \\
\mathsf{maskable}(t, e, \mathit{ternary}) &\triangleq \mathcal{H}(e) \subseteq t.\mathit{valids}
\end{aligned}$$

Figure 12 Semantics of header types (left) and auxiliary functions (right).

A header type Θ represents a set of possible co-valid header instances. The type 0 denotes the empty set. This type arises when there are unsatisfiable assumptions about which headers are valid. The type 1 denotes the singleton denoting the empty set of headers. It describes the type of the initial state of the program. The type h denotes a singleton set, $\{\{h\}\}$ – i.e., states where only h is valid. The type $\Theta_1 \cdot \Theta_1$ denotes the set obtained by combining headers from Θ_1 and Θ_2 – i.e., a product or concatenation. Finally, the type $\Theta_1 + \Theta_2$ denotes the union of Θ_1 or Θ_2, which intuitively represents an alternative.

The semantics of header types, $[\![\Theta]\!]$, is defined by the equations in Figure 12. Intuitively, each subset represents one alternative set of headers that may be valid. For example, the header type $\mathtt{eth} \cdot (\mathtt{ipv4} + 1)$ denotes the set $\{\{\mathtt{eth}, \mathtt{ipv4}\}, \{\mathtt{eth}\}\}$.

To formulate the typing rules for SafeP4, we also define a set of operations on header types: `Restrict`, `NegRestrict`, `Includes`, `Remove`, and `Empty`. The restrict operator `Restrict` $\Theta\ h$ recursively traverses Θ and keeps only those choices in which h is contained, mapping all others to 0. Semantically this has the effect of throwing out the subsets of $[\![\Theta]\!]$ that do not contain h. Dually `NegRestrict` $\Theta\ h$ produces only those choices/subsets where h is invalid. `Includes` $\Theta\ h$ traverses Θ and checks that h is always valid. Semantically this says that h is a member of every element of $[\![\Theta]\!]$. `Remove` $\Theta\ h$ removes h from every path, which means, semantically that it removes h from ever element of $[\![\Theta]\!]$. Finally, `Empty` Θ checks whether Θ denotes the empty set. We can lift these operators to operate on sets of headers in the obvious way. An in-depth treatment of these operators can be found in the accompanying technical report.

3.3.1 Typing Judgement

The typing judgement has the form $\Gamma \vdash c : \Theta \Rrightarrow \Theta'$, which means that in variable context Γ, if c is executed in the header context Θ, then a header instance type Θ' is assigned. Intuitively, Θ encodes the sets of headers that may be valid when type checking a command. Γ is a standard type environment which maps variables x to type τ. If there exists Θ' such that $\Gamma \vdash c : \Theta \Rrightarrow \Theta'$, we say that c is well-typed in Θ.

The typing rules rely on several auxiliary definitions shown in Figure 12. The field type lookup function $\mathcal{F}(h, f_i)$ returns the type assigned to a field f_i in header h by looking it up from the global header type declarations via the header instance declarations. The action lookup function $\mathcal{A}(a)$ returns the action definition $\lambda\bar{x} : \bar{\tau}.\ c$ for action a. Finally, the function $\mathcal{CA}(t, H)$ computes the run-time actions for table t, while $\mathcal{CV}(t)$ computes t's assumptions about validity. Both of these are assumed to be instantiated by the control plane in a way that satisfies basic correctness properties – see technical report.

T-Zero

$$\frac{\texttt{Empty}\ \Theta_1}{\Gamma \vdash c : \Theta_1 \Mapsto \Theta_2}$$

T-Skip

$$\frac{}{\Gamma \vdash \mathit{skip} : \Theta \Mapsto \Theta}$$

T-Seq

$$\frac{\Gamma \vdash c_1 : \Theta \Mapsto \Theta_1 \qquad \Gamma \vdash c_2 : \Theta_1 \Mapsto \Theta_2}{\Gamma \vdash c_1 ; c_2 : \Theta \Mapsto \Theta_2}$$

T-If

$$\frac{\begin{array}{c}\Gamma ; \Theta \vdash e : \mathit{Bool} \\ \Gamma \vdash c_1 : \Theta \Mapsto \Theta_1 \qquad \Gamma \vdash c_2 : \Theta \Mapsto \Theta_2\end{array}}{\Gamma \vdash \mathit{if}\ (e)\ c_1\ \mathit{else}\ c_2 : \Theta \Mapsto \Theta_1 + \Theta_2}$$

T-IfValid

$$\frac{\begin{array}{c}\Gamma \vdash c_1 : \texttt{Restrict}\ \Theta\ h \Mapsto \Theta_1 \\ \Gamma \vdash c_2 : \texttt{NegRestrict}\ \Theta\ h \Mapsto \Theta_2\end{array}}{\Gamma \vdash \mathit{valid}(h)\ c_1\ \mathit{else}\ c_2 : \Theta \Mapsto \Theta_1 + \Theta_2}$$

T-Mod

$$\frac{\begin{array}{c}\texttt{Includes}\ \Theta\ h \\ \mathcal{F}(h, f) = \tau_i \qquad \Gamma ; \Theta \vdash e : \tau_i\end{array}}{\Gamma \vdash h.f = e : \Theta \Mapsto \Theta}$$

T-Extr

$$\frac{}{\Gamma \vdash \mathit{extract}(h) : \Theta \Mapsto \Theta \cdot h}$$

T-Emit

$$\frac{}{\Gamma \vdash \mathit{emit}(h) : \Theta \Mapsto \Theta}$$

T-Add

$$\frac{}{\Gamma \vdash \mathit{add}(h) : \Theta \Mapsto \Theta \cdot h}$$

T-Rem

$$\frac{}{\Gamma \vdash \mathit{remove}(h) : \Theta \Mapsto \texttt{Remove}\ \Theta\ h}$$

T-Apply

$$\frac{\begin{array}{c}\mathcal{CV}(t) = \bar{S} \qquad t.\mathit{actions} = \bar{a} \qquad t.\mathit{reads} = \bar{r} \\ \bar{e} = \{e_j \mid (e_j, m_j) \in \bar{r} \land \neg \mathsf{maskable}(t, e_j, m_j)\} \\ \cdot ; \Theta \vdash e_j : \tau_j \quad \text{for } e_j \in \bar{e} \\ \texttt{Restrict}\ \Theta\ S_i \vdash a_i : \bar{\tau}_i \to \Theta_i' \quad \text{for } a_i \in \bar{a}\end{array}}{\Gamma \vdash t.\mathit{apply}() : \Theta \Mapsto \left(\sum_{a_i \in \bar{a}} \Theta_i'\right)}$$

Figure 13 Command typing rules for SafeP4.

The typing rules for commands are presented in Figure 13. The rule T-Zero gives a command an arbitrary output type if the input type is empty. It is needed to prove preservation. The rules T-Skip and T-Seq are standard. The rule T-If a path-sensitive union type between the type computed for each branch. The rule T-IfValid is similar, but leverages knowledge about the validity of h. So the true branch c_1 is checked in the context $\texttt{Restrict}\ \Theta\ h$, and the false branch c_2 is checked in the context $\texttt{NegRestrict}\ \Theta\ h$. The top-level output type is the union of the resulting output types for c_1 and c_2. The rule T-Mod checks that h is guaranteed to be valid using the `Includes` operator, and uses the auxiliary function $\mathcal{F}$ to obtain the type assigned to $h.f$. Note that the set of valid headers does not change when evaluating an assignment, so the output and input types are identical. The rules T-Extr and T-Add assign header extractions and header additions the type $\Theta \cdot h$, reflecting the fact that h is valid after the command executes. Emitting packet headers does not change the set of valid headers, which is captured by rule T-Emit. The typing rule T-Rem uses the `Remove` operator to remove h from the input type Θ. Finally, the rule T-Apply checks table applications. To understand how it works, let us first consider a simpler, but less precise, typing rule:

$$\frac{\begin{array}{c}t.\mathit{reads} = \bar{e} \qquad \cdot ; \Theta \vdash e_i : \tau_i \quad \text{for } e_i \in \bar{e} \\ t.\mathit{actions} = \bar{a} \qquad \cdot ; \Theta \vdash a_i : \bar{\tau}_i \to \Theta_i' \quad \text{for } a_i \in \bar{a}\end{array}}{\cdot \vdash t.\mathit{apply}() : \Theta \Mapsto \left(\sum \Theta_i'\right)}$$

Intuitively, this rule says that to type check a table application, we check each expression it reads and each of its actions. The final header type is the union of the types computed for

$$\frac{\Gamma, \bar{x} : \bar{\tau} \vdash c : \Theta \Mapsto \Theta'}{\Gamma; \Theta \vdash \lambda\ \bar{x} : \bar{\tau}.c : \bar{\tau} \to \Theta'} \quad (\text{T-Action})$$

Figure 14 Action typing rule for SafeP4.

$$\text{T-Const}\ \frac{\texttt{typeof}(k) = \bar{\tau} \to \tau' \qquad \Gamma; \Theta \vdash e_i : \tau_i}{\Gamma; \Theta \vdash k(\bar{e}) : \tau'} \qquad \text{T-Var}\ \frac{x : \tau \in \Gamma}{\Gamma; \Theta \vdash x : \tau} \qquad \text{T-Field}\ \frac{\texttt{Includes}\ \Theta\ h \qquad \mathcal{F}(h, f) = \tau}{\Gamma; \Theta \vdash h.f : \tau}$$

Figure 15 Expression typing rules for SafeP4.

the actions. To put it another way, it models table application as a non-deterministic choice between its actions. However, while this rule is sound, it is overly conservative. In particular, it does not model the fact that the control plane often uses header validity bits to control which actions are executed.

Hence, the actual typing rule, T-Apply, is parameterized on a function $\mathcal{CV}(t)$ that models the choices made by the control plane, returning for each action a_i, a set of headers S_i that can be assumed valid when type checking a_i. From the reads declarations of the table declaration, we can derive a subset of the expressions read by the table – e.g., excluding expressions that can be wildcarded when certain validity bits are false. This is captured by the function $\mathsf{maskable}(t, e, m)$ (defined in Figure 12) , which determines whether a reads expression e with match-kind m in table t can be masked using a wild-card. The $\mathsf{maskable}$ function is defined using $\mathcal{H}(e)$, which returns the set of header instances referenced by an expression e.

In the example from Section 2.2.3, if an action a_j is matched by the rule $(0, *, 0, *)$, both S_j and e_j are empty.

The typing judgement for actions (Figure 14) is of the form $\Gamma; \Theta \vdash a : \bar{\tau} \to \Theta$, meaning that a has type $\bar{\tau} \to \Theta$ in variable context Γ and header context Θ. Given a variable context Γ and header type Θ, an action $\lambda\bar{x}.\ c$ encodes a function of type $\bar{\tau} \to \Theta'$, so long as the body c is well-typed in the context where Γ is extended with $x_i : \tau_i$ for every i.

The typing rules for expressions are shown in Figure 15. Constants are typechecked according to rule T-Constant, as long as each expression that is passed as an argument to the constant k has the type required by the `typeof` function. The rule T-Var is standard.

3.4 Operational Semantics

We now present the small-step operational semantics of SafeP4. We define the operational semantics for commands in terms of four-tuples $\langle I, O, H, c\rangle$, where I is the input bit stream (which is assumed to be infinite for simplicity), O is the output bit stream, H is a map that associates each valid header instance with a records containing the values of each field, and c is the command to be evaluated. The reduction rules are presented in Figure 16.

The command $\mathit{extract}(h)$ evaluates via the rule E-Extr, which looks up the header type in $\mathcal{HT}$ and then invokes corresponding deserialization function. The deserialized header value v is added to to the map of valid header instances, H. For example, assuming the header type $\eta = \{f : \mathit{bit}\langle 3\rangle;\ g : \mathit{bit}\langle 2\rangle; \}$ has two fields f and g and $I = 11000B$ where B is the rest of the bit stream following, then $\mathit{deserialize}_\eta(I) = (\{f = \mathit{110};\ g = \mathit{00}; \}, B)$.

$$\textsc{E-Extr}\quad \frac{\mathcal{HT}(h) = \eta \qquad \mathit{deserialize}_\eta(I) = (v, I')}{\langle I, O, H, \mathit{extract}(h)\rangle \rightarrow \langle I', O, H[h \mapsto v], \mathit{skip}\rangle}$$

$$\textsc{E-Emit}\quad \frac{\mathcal{HT}(h) = \eta \qquad \mathit{serialize}_\eta(H(h)) = \bar{B}}{\langle I, O, H, \mathit{emit}(h)\rangle \rightarrow \langle I, O.\bar{B}, H, \mathit{skip}\rangle}$$

$$\textsc{E-EmitInvalid}\quad \frac{h \notin \mathit{dom}(H)}{\langle I, O, H, \mathit{emit}(h)\rangle \rightarrow \langle I, O, H, \mathit{skip}\rangle}$$

$$\textsc{E-IfValidTrue}\quad \frac{h \in \mathit{dom}(H)}{\langle I, O, H, \mathit{valid}(h)\ c_1\ \mathit{else}\ c_2\rangle \rightarrow \langle I, O, H, c_1\rangle}$$

$$\textsc{E-IfValidFalse}\quad \frac{h \notin \mathit{dom}(H)}{\langle I, O, H, \mathit{valid}(h)\ c_1\ \mathit{else}\ c_2\rangle \rightarrow \langle I, O, H, c_2\rangle}$$

$$\textsc{E-Mod}\quad \frac{H(h) = r \qquad r' = \{r\ \mathit{with}\ f = v\}}{\langle I, O, H, h.f = v\rangle \rightarrow \langle I, O, H[h \mapsto r'], \mathit{skip}\rangle}$$

$$\textsc{E-Apply}\quad \frac{\mathcal{CA}(t, H) = (a_i, \bar{v}) \qquad \mathcal{A}(a_i) = \lambda\bar{x}.c_i}{\langle I, O, H, t.\mathit{apply}()\rangle \rightarrow \langle I, O, H, c_i[\bar{v}/\bar{x}]\rangle}$$

$$\textsc{E-Add}\quad \frac{\mathcal{HT}(h) = \eta \qquad \mathit{init}_\eta = v}{\langle I, O, H, \mathit{add}(h)\rangle \rightarrow \langle I, O, H[h \mapsto v], \mathit{skip}\rangle}$$

$$\textsc{E-AddValid}\quad \frac{h \in \mathit{dom}(H)}{\langle I, O, H, \mathit{add}(h)\rangle \rightarrow \langle I, O, H, \mathit{skip}\rangle}$$

$$\textsc{E-Rem}\quad \frac{}{\langle I, O, H, \mathit{remove}(h)\rangle \rightarrow \langle I, O, H \setminus h, \mathit{skip}\rangle}$$

Figure 16 Selected rules of the operational semantics of SafeP4; the elided rules are standard and can be found in the technical report.

$$\textsc{E-Const}\quad \frac{[\![k]\!](v_1, ..., v_n) = v}{\langle H, k(v_1, ..., v_n)\rangle \rightarrow v}$$

$$\textsc{E-Field}\quad \frac{H(h) = \{f_1 : n_1, ..., f_k : n_k\}}{\langle H, h.f_i\rangle \rightarrow n_i}$$

Figure 17 Selected rules of the operational semantics for expressions.

The rule E-Emit serializes a header instance h back into a bit stream. It first looks up the corresponding header type and header value in the header table $\mathcal{HT}$ and the map of valid headers respectively. The header value is then passed to the serialization function for the header type to produce a bit sequence that is appended to the output bit stream. Similarly, we assume that a serialization function is defined for every header type, which takes the bit values of the fields of a header value and concatenates them to produce a single bit sequence. We adopt the semantics of P4 with respect to emitting invalid headers. Emitting an invalid header instance – i.e., a header instance which has not been added or extracted – has no effect on the output bit stream (rule E-EmitInvalid). Notice also that the header remains unchanged in H.

Sequential composition reduces left to right, i.e., the left command needs to be reduced to *skip* before the right command can be reduced (rule E-Seq). The evaluation of conditionals (rules E-If, E-IfTrue, E-IfFalse) is standard. Both E-Seq, E-If, E-IfTrue and E-IfFalse are relegated to the technical report for brevity. The rules for validity checks (E-IfValidTrue, E-IfValidFalse) step to the true branch if $h \in \mathit{dom}(H)$ and to the false branch otherwise.

Table application commands are evaluated according to rule E-Tapply. We first invoke the control plane function $\mathcal{CA}(t, H)$ to determine an action a_i and action data v. Then we

$$\textsc{Ent-Empty}\ \frac{}{\cdot \models 1} \qquad \textsc{Ent-Inst}\ \frac{\mathit{dom}(H) = \{h\}}{H \models\ h} \qquad \textsc{Ent-Seq}\ \frac{H_1 \models \Theta_1 \quad H_2 \models \Theta_2}{H_1 \cup H_2 \models \Theta_1 \cdot \Theta_2} \qquad \textsc{Ent-ChoiceL}\ \frac{H \models \Theta_1}{H \models \Theta_1 + \Theta_2} \qquad \textsc{Ent-ChoiceR}\ \frac{H \models \Theta_2}{H \models \Theta_1 + \Theta_2}$$

Figure 18 The *Entailment* relation between header instances and header instance types.

use $\mathcal{A}$ to lookup the definition of a_i, yielding $\lambda\bar{x} : \bar{\tau}.\ c_i$ and step to $c_i[\bar{v}/\bar{x}]$. Note that for simplicity, we model the evaluation of expressions read by the table using the control-plane function $\mathcal{CA}$.

The rule E-Add evaluates addition commands $\mathit{add}(h)$. Similar to header extraction, the $\mathit{init}_\eta()$ function produces a header instance v of type η with all fields set to a default value and extends the map H with $h \mapsto v$. Note that according to E-Add-Exist, if the header instance is already valid, $\mathit{add}(h)$ does nothing. Finally, the rule E-Rem removes the header from the map H. Again, if a header h is already invalid, removing it has no effect.

The semantics for expressions is defined in Figure 17, using tuples $\langle H, e\rangle$, where H is the same map used in the semantics of commands and e is the expression to evaluate. The rule E-Field reduces header field expressions to the value stored in the heap H for the respective field. To evaluate constants via the rule E-Const (omitting the obvious congruence rule), we assume that there is an evaluation function for constants $[\![k]\!](\bar{v}) = v$ that is well-behaved – i.e., if $\texttt{typeof}(k) = \bar{\tau} \to \tau'$ and $\overline{v : \tau}$, then $.;. \vdash [\![k]\!](\bar{v}) : \tau'$. We use these facts to prove progress and preservation.

3.5 Safety of SafeP4

We prove safety in terms of progress and preservation. Both theorems make use of the relation $H \models \Theta$ which intuitively holds if H is described by Θ. The formal definition, as given in Figure 18, satisfies $H \models \Theta$ if and only if $\mathit{dom}(H) \in [\![\Theta]\!]$.

We prove type safety via progress and preservation theorems. The respective proofs are mostly straightforward for our system – we highlight the unusual and nontrivial cases below an relegate the full proofs to the technical report.

▶ **Theorem 1** (Progress). *If* $\cdot \vdash c\!:\!\Theta \Rrightarrow \Theta'$ *and* $H \models \Theta$*, then either,*
- $c = \mathit{skip}$*, or*
- $\exists\langle I', O', H', c'\rangle.\ \langle I, O, H, c\rangle \to \langle I', O', H', c'\rangle$.

Intuitively, progress says that a well-typed command is fully reduced or can take a step.

▶ **Theorem 2** (Preservation). *If* $\Gamma \vdash c\!:\!\Theta_1 \Rrightarrow \Theta_2$ *and* $\langle I, O, H, c\rangle \to \langle I', O', H', c'\rangle$*, where* $H \models \Theta_1$*, then* $\exists\Theta_1', \Theta_2'.\ \Gamma \vdash c\!:\!\Theta_1' \Rrightarrow \Theta_2'$ *where* $H' \models \Theta_1'$ *and* $\Theta_2' < \Theta_2$.

More interestingly, preservation says that if a command c is well-typed with input type Θ_1 and output type Θ_2, and c evaluates to c' in a single step, then there exists an input type Θ_1' and an output type Θ_2' that make c' well-typed. To make the inductive proof go through, we also need to prove that Θ_1' describes the same maps of header instance H as Θ_1, and Θ_2' is semantically contained in Θ_2. We define syntactic containment to be $\Theta_1 < \Theta_2 \triangleq [\![\Theta_1]\!] \subseteq [\![\Theta_2]\!]$. (These conditions are somewhat reminiscent of conditions found in languages with subtyping.)

Proof. By induction on a derivation of $\Gamma \vdash c : \Theta_1 \Rrightarrow \Theta_2$, with a case analysis on the last rule used. We focus on two of the most interesting cases. See technical report for the full proof.

Case T-IfValid: $c = \mathit{valid}(h)\ c_1\ \mathit{else}\ c_2$ and $\Gamma \vdash c_1 : \texttt{Restrict}\ \Theta_1\ h \Rrightarrow \Theta_{12}$ and $\Gamma \vdash c_2 : \texttt{NegRestrict}\ \Theta_1\ h \Rrightarrow \Theta_{22}$ and $\Theta_2 = \Theta_{12} + \Theta_{22}$.

There are two evaluation rules that apply to c, E-IfValidTrue and E-IfValidFalse

Subcase E-IfValidTrue: $c' = c_1$ and $h \in \mathit{dom}(H)$ and $H' = H$.

Let $\Theta_1' = \texttt{Restrict}\ \Theta_1\ h$ and $\Theta_2' = \Theta_{12}$. We have $\Gamma \vdash c' : \Theta_1' \Rrightarrow \Theta_2'$ by assumption, we have $H \models \Theta_1'$ by a lemma formalizing the relationship between Restrict and $(\models)$ (see tech report), and we have $\Theta_2' < \Theta_2$ by the definition of $<$ and the semantics of union.

Subcase E-IfValidFalse: $c' = c_2$ and $h \notin \mathit{dom}(H)$ and $H' = H$.

Symmetric to the previous case.

Case T-Apply: $c = t.\mathit{apply}()$ and $\mathcal{CV}(t) = (\bar{S}, \bar{e})$ and $t.\mathit{actions} = \bar{a}$ and $\cdot; \Theta \vdash e_j : \tau_j$ for $e_j \in \bar{e}$ and $\texttt{Restrict}\ \Theta_1\ S_i \vdash a_i : \bar{\tau}_i \to \Theta_i'$ for $a_i \in \bar{a}$ and $\Theta_2 = \sum(\Theta_i')$

Only one evaluation rule applies to c, E-Apply. It follows that $\mathcal{CA}(t, H) = (a_i, \bar{v})$, and $c' = c_i[\bar{v}/\bar{x}]$ where $\mathcal{A}(a_i) = \lambda\bar{x}.\ c_i$. By inverting T-Action, we have $\Gamma, \bar{x} : \bar{\tau}_i; \vdash c_i : \texttt{Restrict}\ \Theta\ S_i \Rrightarrow \Theta_i'$. By control plane assumption (2), we have $\cdot; \cdot \vdash \bar{v} : \bar{\tau}_i$. By the substitution lemma, we have $\Gamma \vdash c_i[\bar{v}/\bar{x}] : \texttt{Restrict}\ \Theta\ S_i \Rrightarrow \Theta_i'$. Let $\Theta_1' = \texttt{Restrict}\ \Theta\ S_i$ and $\Theta_2' = \Theta_i'$. We have shown that $\Gamma \vdash c' : \Theta_1' \Rrightarrow \Theta_2'$, we have that $H' \models \Theta_1'$ by control plane assumption (3), and we have $\Theta_2' < \Theta_2$ by the definition of $<$ and the semantics of union types. ◀

4 Experience (Evaluation)

We implemented our type system in a tool called P4Check that automatically checks P4 programs and reports violations of the type system presented in Figure 13. P4Check uses the front-end of `p4v` [20] and handles the full P4_{14} language.[3] Our key findings, which are reported in detail below, show (i) that our type system finds bugs "in the wild" and (ii) that the programmer effort needed to repair programs to pass our type checker is modest.

4.1 Overview of Bugs in the Wild

We ran P4Check on 15 open source P4_{14} programs[4] of varying sizes and complexity, ranging from 143 to 9060 lines of code. Our criteria for selecting programs was: (1) each program had to be open source, (2) available on GitHub, and (3) compile without errors, (4) and be written either by industrial teams developing production code or by researchers implementing standard or novel network functionality in P4 – i.e., we excluded programs primarily used for teaching. Out of the 15 subject programs only 4 passed our type checker, all of which were simple implementations of routers or DDoS mitigation that accepted only a small number of packet types and were relatively small (188–635 lines of code). For the remaining 11 programs (industrial and research) our checker found 418 type checking violations overall.

[3] We also have an open-source prototype implementation for P4_{16} that handles the most common features of P4_{16} (`https://github.com/cornell-netlab/p4check`).

[4] We chose to check P4_{14} instead of P4_{16}, since there are currently more P4_{14} programs available on GitHub.

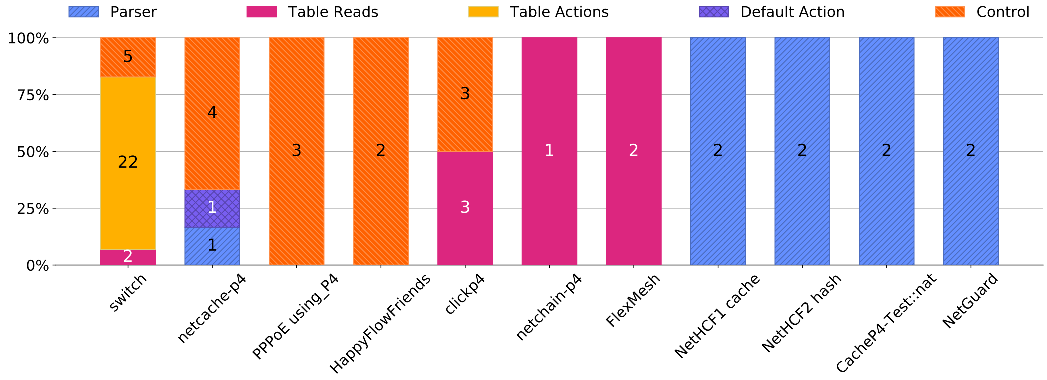


Figure 19 Proportional frequencies of each bug type per-program. The raw number of bugs for each program and category is reported at the top of each stacked bar.

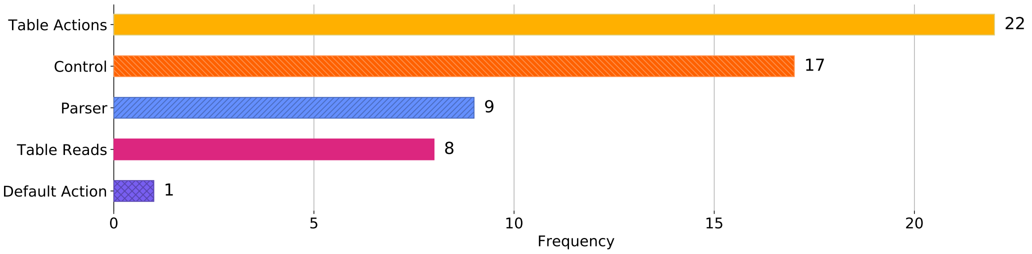


Figure 20 Frequency of each bug across all programs. The raw number of bugs in each category is reported to the right of the bar.

Frequently, multiple violations produced by P4Check have the same root cause. For example, if a single action `rewrite_ipv4` that rewrites fields `srcAddr` and `dstAddr` for an `ipv4` header is called in a context that cannot prove that `ipv4` is valid, then both references to `ipv4.srcAddr` and `ipv4.dstAddr` will be reported as violations, even though they are due to the same *control* bug (Section 2.2.2) – namely that `rewrite_ipv4` was not called in a context that could prove the validity of `ipv4`. To address this issue, we applied another metric to quantify the number of bugs (inspired by the method proposed by others [17]): we equate the number of bugs in each program with the number of bug *fixes* required to make the program in question pass our type checker. Using this metric, we counted 58 bugs.

We classified the bugs according to the classes described in Section 2.2. Figure 19 depicts the per-program breakdown of the frequency of each bug class, and Figure 20 depicts the overall frequency of each bug. Notice that even though table action bugs were the most frequent bug (with 22 occurrences), they were only found in a single program (`switch.p4`). These bugs are especially prevalent in this program because of its heavy reliance on correct control-plane configuration. Conversely, there were 9 occurrences across 5 programs for both parser bugs and table reads bugs.

Readers familiar with previous work on `p4v` [20], a recent P4 verification tool, may notice that we detected no default action bugs for the `switch.p4` program, while `p4v` reported many! The reasons for this are two-fold. First, `p4v` allows programmers to verify complex properties, which means that it can express fine-grained conditions on tables and relationships between them. In contrast, we make heuristic assumptions about P4 programs that automatically eliminate many bugs, including some default action bugs. Second, our repairs are often

```
./h.p4, line 350, cols 12-21: error tcp not guaranteed to be valid
./h.p4, line 118, cols 8-16: error ipv4 not guaranteed to be valid
./h.p4, line 101, cols 42-50: error ipv4 not guaranteed to be valid
./h.p4, line 320, cols 8-15: error tcp not guaranteed to be valid
./h.p4, line 362, cols 12-19:error tcp not guaranteed to be valid
./h.p4, line 362, cols 29-36: error tcp not guaranteed to be valid
./h.p4, line 295, cols 60-69: error tcp not guaranteed to be valid
./h.p4, line 107, cols 8-16: error ipv4 not guaranteed to be valid
./h.p4, line 101, cols 42-50: error ipv4 not guaranteed to be valid
./h.p4, line 163, cols 8-16: error ipv4 not guaranteed to be valid
./h.p4, line 101, cols 42-50: error ipv4 not guaranteed to be valid
```

```
./h.p4, line 350, cols 12-21: error tcp not guaranteed to be valid
./h.p4, line 320, cols 8-15: error tcp not guaranteed to be valid
./h.p4, line 362, cols 12-19: error tcp not guaranteed to be valid
./h.p4, line 362, cols 29-36: error tcp not guaranteed to be valid
./h.p4, line 295, cols 60-69: error tcp not guaranteed to be valid
```

Figure 21 Curated output from P4Check for the parser bug in NetHCF before (above) and after (below) modifying `parse_ethernet`.

coarse-grained and may enforce a stronger guarantee on the program than may be necessary; using first-order logic annotations, `p4v` programmers manually specify the weakest (and hence more complex) assumptions.

We make no claims about the completeness of our taxonomy. For example, we found one instance, in the HappyFlowFriends program, where the programmer had mistakenly instantiated metadata m as a header, and consequently did not parse m (since metadata is always valid) causing m to (ironically) always be invalid.

4.2 P4Check in Action

We reprise the canonical examples of each class of bugs from Section 2.2, describing how P4Check detects them and discussing ways to fix them.

4.2.1 Parser Bugfixes

Recall Figure 5, which exhibits the parser bug. The bug occurs because the parser, which extracts IPv4-TCP packets, boots unexpected packets (such as IPv6 or UDP packets) directly to `ingress`, which then assumes that both the `ipv4` and `tcp` headers are valid, even though the parser does not guarantee this fact.

In terms of our type system, the parser produces packets of type $\texttt{ethernet} \cdot (1 + \texttt{ipv4} \cdot (1 + \texttt{tcp}))$; however the control only handles packets of type $\texttt{ethernet} \cdot \texttt{ipv4} \cdot \texttt{tcp}$. Hence, when typecheck this example, P4Check reports every reference to `tcp` and `ipv4` in the whole program as a violation of the type system. As shown in the top half of Figure 21, we get an error message at every reference to `ipv4` or `tcp`. The ubiquity of the reports intimates a mismatch between the parsing and the control types, which gives the programer a hint as how to fix the problem.

When we modify the `default` clause in `parse_ethernet`, as in Figure 5, and run our tool again, all of the `ipv4` violations are removed from the output, as shown in the bottom half of Figure 21. Then fixing the `parse_ipv4` parser, as in Figure 5, causes our tool to output no violations. In particular, the type upon entering the `ingress` control function is $\texttt{ethernet} \cdot \texttt{ipv4} \cdot \texttt{tcp}$, so all subsequent references to `ipv4` and `tcp` are safe.

```
port.p4, line 248, cols 8-24: warning: assuming either vlan_tag_[0]
    matched as valid or vlan_tag_[0].vid wildcarded

port.p4, line 250, cols 8-24: warning: assuming either vlan_tag_[1]
    matched as valid or vlan_tag_[1].vid wildcarded
```

```
fabric.p4 line 42, cols 41-67: warning: assuming fabric_header_cpu
    matched as valid for rules with action terminate_cpu_packet

fabric.p4, line 57, cols 17-54: warning: assuming fabric_header_unicast
     matched as valid for rules with action
    terminate_fabric_unicast_packet

fabric.p4, line 81, cols 17-56: warning: assuming
    fabric_header_multicast matched as valid for rules with action
    terminate_fabric_multicast_packet
```

Figure 22 Warnings printed after fixing `switch.p4`'s reads bug (top), and its actions bug (bottom).

4.2.2 Control Bugfixes

Recall that a control bug occurs when the incoming type presents a choice between two instances that are not handled by subsequent code. The program shown in Figure 6 uses a parser that produces the type $\Theta = \texttt{ethernet}\cdot(1+\texttt{ipv4}\cdot(1+\texttt{udp}\cdot(1+\texttt{nc\_hdr}\cdot\tau)+\texttt{tcp}))$, where τ is a type for caching operations. Note that `Includes` Θ `nc_hdr` does not hold. However, `process_cache` and `process_value` only type check in contexts where `Includes` Θ `nc_hdr` is true. P4Check reports type violations at every reference to `nc_hdr`. Fixing this error is simply a matter of wrapping the `process_cache()` call in a validity check as demonstrated in Figure 6. As NetCache handles TCP and UDP packets as well as its special-purpose packets, we simply apply the IPv4 routing table if the validity check for `nc_hdr` fails.[5]

4.2.3 Table Reads Bugfixes

Table reads errors, as shown in Figure 7, occur when a header h is included in the `reads` declaration of a table t with match kind k, and h is not guaranteed to be valid at the call site of t, and if $h \notin \texttt{valid\_reads}(t)$ or the match-kind of $k \neq \texttt{ternary}$.

In the case of the `port_vlan_mapping` table in Figure 7, there is a valid bit for both `vlan_tag_[0]` and `vlan_tag_[1]`, both of which are followed by `exact` matches. To solve this problem, we need to use the `ternary` match-kind instead, which allows the use of wildcard matching. When a field is matched with a wildcard, the table does not attempt to compute the value of the `reads` expression.

This fix assumes that the controller is well behaved and fills the `vlan_tag_[0].vid` with a wildcard whenever `vlan_tag_[0]` is matched as invalid (and similarly for `vlan_tag_[1]`). This also what the SafeP4 type system does, with its maskable checks in the T-Apply rule P4Check prints warnings describing these assumptions to the programmer (top of Figure 22), giving them properties against which to check their control plane implementation.

4.2.4 Table Action Bugfixes

Table actions bugs occur when at least one action cannot be safely executed in all scenarios. For example, the table `fabric_ingress_dst_lkp` shown in Figure 8 has a table action bug, which can be fixed by modifying the table's `reads` declaration. Recall that the

[5] Astute readers may detect a parser bug in this example. Hint, the `ipv4_route` table requires `Includes` Θ `ipv4` where Θ is type where it is applied.

parser will parse exactly one of the headers `fabric_hdr_cpu`, `fabric_hdr_unicast` and `fabric_hdr_multicast`, which means that when the table is applied at type Θ, exactly one of `Includes` Θ `fabric_hdr_i` for $i \in \{\texttt{cpu}, \texttt{unicast}, \texttt{multicast}\}$ will hold. Now, the action `term_cpu_packet` typechecks only with the (nonempty) type `Restrict` Θ `fabric_hdr_cpu`, and the actions `term_fabric_`i`_packet` only typecheck with the (nonempty) types `Restrict` Θ `term_fabric_`i`_packet` for $i = \texttt{unicast}, \texttt{multicast}$. P4Check suggests that this is the cause of the bug since it reports type violations for all of the references to these three headers in the control paths following from the application of `fabric_ingress_dst_lkp`.

The optimal[6] fix here is to augment the `reads` declaration to include a validity check for each contentious header. We then assume that the controller is well-behaved enough to only call actions when their required headers are valid, allowing us to typecheck each action in the appropriate type restriction. P4Check alerts the programmer whenever it makes such an assumption. We show these warnings for the fixed version of `fabric_ingress_dst_lkp` below the line in Figure 22.

4.2.5 Default Action Bugfix

Default action bugs occur when a programmer creates a wrapper table for an action that modifies the type, and forgets to force the table to call that action when the packet misses. The `add_value_header_1` table from Figure 9 wraps the action `add_value_header_1_act`, which calls the single line `add_header(nc_value_1)`.

The default action, when left unspecified, is `nop`, which means that if the pre-application type was Θ, then the post-application type is $\Theta + \Theta \cdot \texttt{nc\_value\_1}$, which does not include `nc_value_1`. Hence, P4Check reports every subsequent reference (on this code path) to `nc_header_1` to be a type violation.

To fix this bug, we need to set the default action to `add_value_1` – this makes the post-application type $\Theta \cdot \texttt{nc\_value\_1} + \Theta \cdot \texttt{nc\_value\_1} = \Theta \cdot \texttt{nc\_value\_1}$, which includes `nc_value_1`, thus allowing the subsequent code to typecheck.

4.3 Overhead

It is important to evaluate two kinds of overhead when considering a static type system: overhead on programmers and on the underlying implementation.

Typically, adding a static type system to a dynamic type system requires more work for the programmer – the field of gradual typing is devoted breaking the gargantuan task into smaller commit-sized chunks [5]. Surprisingly, in our experience, migrating real-world P4 code to pass the SafeP4 type system only required modest programmer effort.

To qualitatively evaluate the effort required to change an unsafe program into a safe one using our type system, we manually fixed all of the detected bugs. The programs that had bugs required us to edit between 0.10% and 1.4% of the lines of code. The one exception was PPPoE_using_P4, which was a 143 line program that required 6 line-edits (4%), all of which were validity checks. Conversely, `switch.p4` required 34 line edits, the greatest observed number, but this only accounted for 0.37% of the total lines of code in the program.

Each class of bugs has a simple one-to-two line fix, as described in Section 4.2: adding a validity check, adding a default action, or slightly modifying the parser. Each of these changes was straightforward to identify and simple to make.

[6] Another fix would be to refactor the single into multiple tables, each guarded by a separate validity check. However, combining this kind of logic in a single table helps conserve memory, so in striving to change the behavior of the program as little as possible, we propose modifying the table reads.

Another possible concern is that that extending tables with extra read expressions, or adding run-time validity checks to controls, might impose a heavy cost on implementations, especially on hardware. Although we have not yet performed an extensive study of the impact on compiled code, based on the size and complexity of the annotations we added, we believe the additional cost should be quite low. We were able to compile our fixed version of the `switch.p4` program to the Tofino architecture [24] with only a modest increase in resource usage. Overall, given the large number of potential bugs located by P4Check, we believe the assurance one gains about safety properties by using a static type system makes the costs well worth it.

5 Related Work

Probably the most closely related work to SafeP4 is `p4v` [20]. Unlike SafeP4, which is based on a static type system, `p4v` uses Dijkstra's approach to program verification based on predicate transformer semantics. To model the behavior of the control plane, `p4v` uses first-order annotations. SafeP4's typing rule for table application is inspired by this idea, but adopts simple heuristics – e.g., we only assume that the control plane is well-behaved – rather than requiring logical annotations.

Both `p4v` and P4Check can be used to verify safety properties of data planes modelled in P4 – e.g., that no read or write operations are possible on an invalid header. As it is often the case when comparing approaches based on types to those based on program verification, `p4v` can check more complex properties, including architectural invariants and program-specific properties – e.g., that the IPv4 time-to-live field is correctly decremented on every packet. However, in general, it requires annotating the program with formal specifications both for the correctness property itself and to model the behavior of the control plane.

McKeown et al. developed an operational semantics for P4 [22], which is translated to Datalog to verify safety properties and to check program equivalence. An operational semantics for P4 was also developed in the K framework [27], yielding a symbolic model checker and deductive verification tool [16]. Vera [30] models the semantics of P4 by translation to SymNet [31], and develops a symbolic execution engine for verifying a variety of properties, including header validity.

Compared to SafeP4, these approaches do not use their formalization of P4 as a foundation for defining a type system that addresses common bugs. To the best of our knowledge, SafeP4 is the first formal calculus for a P4-like packet processing language that provides correct-by-construction guarantees of header safety properties.

Other languages have used type systems to rule our safety problems due to null references. For example, NullAway [29] analyzes all Java programs annotated with `@Nullable` annotations, making path-sensitive deductions about which references may be null. Similar to the validity checks in SafeP4, NullAway analyses conditionals for null checks of the form `var != null` using data flow analysis.

Looking further afield, PacLang [9] is a concurrent packet-processing language that uses a linear type system to allow multiple references to a given packet within a single thread. PacLang and SafeP4 share the use of a type system for verifying safety properties but they differ in the kind of properties they address and, hence, the kind of type system they employ for this purpose. In addition, the primary focus in PacLang is on efficient compilation whereas SafeP4 is concerned with ensuring safety of header data.

Domino [28] is a domain-specific language for data plane algorithms supporting *packet transactions* – i.e., blocks of code that are guaranteed to be atomic and isolated from other transactions. In Domino, the programmer defines the operations needed for each packet without worrying about other in-flight packets. If it succeeds, the compiler guarantees performance at the line rate supported on programmable switches. Overall, Domino focuses on transactional guarantees and concurrency rather than header safety properties.

BPF+ [3] and eEBPF [8] are packet-processing frameworks that can be used to extend the kernel networking stack with custom functionality. The modern eBPF framework is based on machine-level programming model, but it uses a virtual machine and code verifier to ensure a variety of basic safety properties. Much of the recent work on eBPF focuses on techniques such as just-in-time compilation to achieve good performance.

SNAP [1] is a language for stateful packet processing based on P4. It offers a programming model with global state registers that are distributed across many physical switches while optimizing for various criteria, such as minimizing congestion. More specifically, the compiler analyses read/write dependencies to automatically optimize the placement of state and the routing of traffic across the underlying physical topology.

While our approach to track validity is network-specific, it is similar to taint analysis [33, 10, 11], which attempts to identify secure program parts that can be safely accessed.

Of course, there is a long tradition of formal calculi that aim to capture some aspect of computation and make it amenable for mathematical reasoning. The design of SafeP4 is directly inspired by Featherweight Java [12], which stands out for its elegant formalization of a real-world language in an extensible core calculus.

6 Conclusion

P4 provides a powerful programming model for network devices based on high-level and declarative abstractions. Unfortunately, P4 lacks basic safety guarantees, which often lead to a variety of bugs in practice. This paper proposes SafeP4, a domain-specific language for programmable data planes that comes equipped with a formal semantics and a static type system which ensures that every read or write to a header at run-time will be safe. Under the hood, SafeP4 uses a rich set of types that tracks dependencies beween headers, as well as a path-sensitive analysis and domain-specific heuristics that model common idioms for programming control planes and minimize false positives. Our experiments using an OCaml prototype and a suite of open-source programs found on GitHub show that most P4 applications can be made safe with minimal programming effort. We hope that our work can help lay the foundation for future enhancements to P4 as well as the next generation of data plane languages. In the future, we plan to explore enriching SafeP4's type system to track additional properties, investigate correct-by-construction techniques for writing control-plane code, and develop a compiler for the language.

References

1 Mina Tahmasbi Arashloo, Yaron Koral, Michael Greenberg, Jennifer Rexford, and David Walker. SNAP: Stateful Network-Wide Abstractions for Packet Processing. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 29–43, New York, NY, USA, 2016. ACM. doi:10.1145/2934872.2934892.

2 Jiasong Bai, Jun Bi, Menghao Zhang, and Guanyu Li. Filtering Spoofed IP Traffic Using Switching ASICs. In *Proceedings of the ACM SIGCOMM 2018 Conference on Posters and Demos*, pages 51–53. ACM, 2018.

3 Andrew Begel, Steven McCanne, and Susan L. Graham. BPF+: Exploiting Global Data-flow Optimization in a Generalized Packet Filter Architecture. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '99, pages 123–134, New York, NY, USA, 1999. ACM. doi:10.1145/316188.316214.

4 Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming Protocol-independent Packet Processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, July 2014. doi:10.1145/2656877.2656890.

5 John Peter Campora, Sheng Chen, Martin Erwig, and Eric Walkingshaw. Migrating gradual types. *Proceedings of the ACM on Programming Languages*, 2(POPL):15, 2017.

6 Martin Casado, Michael J Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. Ethane: Taking control of the enterprise. In *ACM SIGCOMM Computer Communication Review*, volume 37 (4), pages 1–12. ACM, 2007.

7 P4 Language Consortium. P4 Language Specification, Version 1.0.4. Technical report, Available at https://p4.org/specs/, 2017.

8 Jonathan Corbet. BPF: the universal in-kernel virtual machine, May 2014. Available at https://lwn.net/Articles/599755/,.

9 Robert Ennals, Richard Sharp, and Alan Mycroft. Linear Types for Packet Processing. In David Schmidt, editor, *Programming Languages and Systems*, pages 204–218, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

10 William G. J. Halfond, Alessandro Orso, and Panagiotis Manolios. Using Positive Tainting and Syntax-aware Evaluation to Counter SQL Injection Attacks. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '06/FSE-14, pages 175–185, New York, NY, USA, 2006. ACM. doi:10.1145/1181775.1181797.

11 Wei Huang, Yao Dong, and Ana Milanova. Type-Based Taint Analysis for Java Web Applications. In *Proceedings of the 17th International Conference on Fundamental Approaches to Software Engineering - Volume 8411*, pages 140–154, New York, NY, USA, 2014. Springer-Verlag New York, Inc. doi:10.1007/978-3-642-54804-8_10.

12 Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: A Minimal Core Calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, May 2001. doi:10.1145/503502.503505.

13 Xin Jin. netcache-p4, March 2018. URL: https://github.com/netx-repo/netcache-p4.

14 Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. NetChain: Scale-free sub-rtt coordination. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, April 2018. Best paper award.

15 Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 121–136. ACM, 2017.

16 Ali Kheradmand and Grigore Roşu. P4K: A formal semantics of P4 and applications. Technical Report https://arxiv.org/abs/1804.01468, University of Illinois at Urbana-Champaign, April 2018.

17 George T. Klees, Andrew Ruef, Benjamin Cooper, Shiyi Wei, and Michael Hicks. Evaluating Fuzz Testing. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, October 2018.

18 Chaitanya Kodeboyina. An open-source P4 switch with SAI support, June 2015. URL: https://p4.org/p4/an-open-source-p4-switch-with-sai-support.html.

19 Rahul Kumar and BB Gupta. Stepping stone detection techniques: Classification and state-of-the-art. In *Proceedings of the international conference on recent cognizance in wireless communication & image processing*, pages 523–533. Springer, 2016.

20 Jed Liu, William Hallahan, Cole Schlesinger, Milad Sharif, Jeongkeun Lee, Robert Soulé, Han Wang, Călin Caşcaval, Nick McKeown, and Nate Foster. P4V: Practical Verification for Programmable Data Planes. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, pages 490–503, New York, NY, USA, 2018. ACM. doi:10.1145/3230543.3230582.

21 Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, March 2008. doi:10.1145/1355734.1355746.

22 Nick McKeown, Dan Talayco, George Varghese, Nuno Lopes, Nikolaj Bjorner, and Andrey Rybalchenko. Automatically verifying reachability and well-formedness in P4 Networks, September 2016. URL: https://www.microsoft.com/en-us/research/publication/automatically-verifying-reachability-well-formedness-p4-networks/.

23 Robin Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17(3):348–375, December 1978.

24 Barefoot Networks. Tofino 2. URL: https://www.barefootnetworks.com/products/brief-tofino-2/.

25 Barefoot Networks. Behavioral Model, December 2018. URL: https://github.com/p4lang/behavioral-model.

26 TJ OConnor, William Enck, W Michael Petullo, and Akash Verma. Pivotwall: SDN-based information flow control. In *Proceedings of the Symposium on SDN Research*, page 3. ACM, 2018.

27 Grigore Roşu and Traian Florin Şerbănuţă. An Overview of the K Semantic Framework. *Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010. doi:10.1016/j.jlap.2010.03.012.

28 Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. Packet Transactions: High-Level Programming for Line-Rate Switches. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 15–28, New York, NY, USA, 2016. ACM. doi:10.1145/2934872.2934900.

29 Manu Sridharan. Engineering NullAway, Uber's Open Source Tool for Detecting NullPointerExceptions on Android, December 2018. URL: https://eng.uber.com/nullaway/.

30 Radu Stoenescu, Dragos Dumitrescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. Debugging P4 Programs with Vera. In *ACM SIGCOMM*, pages 518–532, New York, NY, USA, 2018. ACM. doi:10.1145/3230543.3230548.

31 Radu Stoenescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. SymNet: Scalable symbolic execution for modern networks. In *ACM SIGCOMM*, pages 314–327, New York, NY, USA, 2016. ACM. doi:10.1145/2934872.2934881.

32 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. doi:10.1145/1863543.1863561.

33 Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. A Sound Type System for Secure Flow Analysis. *J. Comput. Secur.*, 4(2-3):167–187, January 1996. URL: http://dl.acm.org/citation.cfm?id=353629.353648.

34 Menghao Zhang. Anti-spoof, November 2018. URL: https://github.com/zhangmenghao/Anti-spoof.

Fling – A Fluent API Generator

Yossi Gil
Technion I.I.T Computer Science Dept., Haifa, Israel
yogi@cs.technion.ac.il

Ori Roth
Technion I.I.T Computer Science Dept., Haifa, Israel
ori.rothh@gmail.com

Abstract

We present the first general and practical solution of the fluent API problem – an algorithm, that given a deterministic language (equivalently, LR(k), $k \geq 0$ language) encodes it in an unbounded parametric polymorphism type system employing only a polynomial number of types. The theoretical result is accompanied by an actual tool FLING– a fluent API compiler-compiler in the venue of YACC, tailored for embedding DSLs in JAVA.

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

Keywords and phrases fluent API, type system, compilation, code generation

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2019.13

Supplement Material ECOOP 2019 Artifact Evaluation approved artifact available at
`https://dx.doi.org/10.4230/DARTS.5.2.12`

Funding *Yossi Gil*: Technion I.I.T
Ori Roth: Technion I.I.T

1 Introduction

Given a formal language ℓ over some finite alphabet Σ, i.e., $\ell \subseteq \Sigma^*$ and a host programming language, e.g., JAVA [1], the *fluent API problem* is to generate $E = E(\ell)$, a set of definitions in the host language that encode ℓ such that membership in ℓ is equivalent to type-checking against E: Concretely, for a word $w = \sigma_1\sigma_2 \cdots \sigma_n$, $\sigma_i \in \Sigma$, $i = 1, \ldots, n$, the chain of method calls

$$\sigma_0.\sigma_1\texttt{()}.\sigma_2\texttt{()}.\cdots.\sigma_n\texttt{()}.\sigma_\$\texttt{()} \tag{1}$$

type checks (in the host language) against $E(\ell)$ if and only if $w \in \ell$. (Here variable σ_0 and method $\sigma_\$$ are specified by E.) The fluent API problem is parameterized by ℓ's ranking in the Chomsky hierarchy and the capabilities of the host language.

Attention was drawn to the problem since fluent APIs are a valuable software engineering technique, useful, e.g., when ℓ specifies an object protocol, or for embedding a domain specific language (DSL) in the host language. (See, e.g., [5] for motivation and applications.)

A straightforward solution for the limited set of regular languages has been known and used for years; yet, interesting DSLs and protocols are often not regular.

Previous *theoretical* advances at the problem remain unpractical due to algorithm complexity, size of $E(\ell)$, or, inherent time complexity of the type checking. Prior attempts to make these theoretical results *practical* remain ad hoc, with no clear specification of the class of formal languages that can be processed.

33rd European Conference on Object-Oriented Programming (ECOOP 2019).
Editor: Alastair F. Donaldson; Article No. 13; pp. 13:1–13:25
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1.1 Contribution

This work presents the first general and practical solution of the fluent API problem. To this end, we prove that any **D**eterministic **C**ontext **F**ree **L**anguage (DCFL) can be encoded in a type system that supports unbounded parametric polymorphism, while employing only a polynomial (in the size of the language specification) number of types.

Recall that DCFL is the class of formal languages that are recognizable by a **D**eterministic **P**ush**D**own **A**utomaton (DPDA). Our proof is supported by an *automata compiler* which converts a DPDA into JAVA definitions so that (1) holds.

The *generality* of the result is in two respects:

- Most programming languages, and hence most DSLs, are designed with practical parsing in mind. The DCFL class includes all languages for which an LL or LR grammar exists.
- The requirements from the type system of the host language are minimal. In particular, we assume the type parameterization:
  - does not allow the generic invoke functions found in the in its type parameter, nor expose any other property of this type
  - does not allow generic class make any assumption, nor place constraints on its type parameter, inherit from it, or supply a default value for it.
  - does not support partial specialization of generic classes (with the help of these, it is possible emulate a two-stack machine, and henceforth a Turing machine [9]),

  In fact, the generics we assume are as weak as found in ML. The only thing a generic can do with its parameter is to pass it as a parameter to another generic.

Conversely, the *practicality* of the result is in two respects:

- The time to type checking an expression such as (1) is linear in its size.
- The length of $E(\ell)$ is polynomial in the size of the DPDA that defines ℓ.

We further demonstrate the theoretical result in an actual tool FLING– a fluent API compiler-compiler in the style of YACC, tailored for embedding DSLs in JAVA. Given an LL(1) grammar $\mathcal{G}$, FLING generates appropriate JAVA definitions of modest size by which (1) type-checks, if and only if $\mathcal{G}$ derives w.

FLING's restriction to LL(1) is not inherent – extending FLING to support LR(1) languages (and hence all deterministic languages) is technical (though laborious). Further, FLING generates JAVA class definitions of the abstract syntax tree (AST) implicit by $\mathcal{G}$. Even further, FLING generates bodies for methods σ_i that generate the AST: When executed, the sequence (1) returns an instance of this AST that describes w, to be used by clients for further processing.

To demonstrate, we use FLING to define a fluent API language for writing regular expressions. In this API the regular expression $(ab?)^* \mid (0-9)^+$ is written as:

```
re.noneOrMore(exactly("a").and().option(exactly("b"))).or().oneOrMore(anyDigit()).$()
```

Composing the API begins with a grammar for the language ℓ_R of regular expressions:

$$\begin{aligned}
\langle \mathit{Expression} \rangle &::= \texttt{re}\ \langle \mathit{RE} \rangle \\
\langle \mathit{RE} \rangle &::= \texttt{exactly}\ \langle \mathit{Tail} \rangle \\
&\quad\mid \texttt{option}\ \langle \mathit{Tail} \rangle \\
&\quad\mid \texttt{noneOrMore}\ \langle \mathit{Tail} \rangle \\
&\quad\mid \texttt{oneOrMore}\ \langle \mathit{Tail} \rangle \\
&\quad\mid \texttt{either}\ \langle \mathit{Tail} \rangle \\
&\quad\mid \texttt{anyChar}\ \langle \mathit{Tail} \rangle \\
&\quad\mid \texttt{anyDigit}\ \langle \mathit{Tail} \rangle \\
\langle \mathit{Tail} \rangle &::= \texttt{and}\ \langle \mathit{RE} \rangle \mid\ \texttt{or}\ \langle \mathit{RE} \rangle \mid \varepsilon.
\end{aligned} \tag{2}$$

This grammar is then supplied into FLING; incidentally, the grammar specification for FLING uses the fluent API style:

Listing 1 Fling fluent API specifying the regex grammar depicted in (2).

```
start(Expression). // ⟨Expression⟩ is the start symbol
derive(Expression).to(re, RE). // ⟨Expression⟩ ::= re⟨RE⟩
derive(RE).to(exactly.with(String.class), Tail) // ⟨RE⟩ ::= exactly⟨Tail⟩
  or(option.with(RE), Tail). // | option⟨Tail⟩
  or(noneOrMore.with(RE), Tail). // | noneOrMore⟨Tail⟩
  or(oneOrMore.with(RE), Tail). // | oneOrMore⟨Tail⟩
  or(either.with(RE, RE), Tail). // | either⟨Tail⟩
  or(anyChar, Tail). // | anyChar⟨Tail⟩
  or(anyDigit, Tail). // | anyDigit⟨Tail⟩
derive(Tail).to(and, RE).or(or, RE).orNone(); // ⟨Tail⟩ ::= and⟨RE⟩| or⟨RE⟩| ε
```

The above chain of method calls produces, at run time, a Fling object of type BNF. This object can then be requested to emit the Java (indeed, also C++ [19]) code that implements the fluent API. To use this API, one must compile and link again this emitted code.

One may disregard the regular-expression example and others as being singular, asking whether it would be better to manually compose a fluent API for regular expressions and other examples, rather than developing a general purpose machinery. An answer would be in considering the problem of embedding XML literals and expressions as fluent code, by writing, e.g.,

```
XMLData t = data().th().td("Customer").td("Number").th(end).tr().td("A").td(3).tr(end).data(end);
```

to create an XML data object. To maintain correctness of the generated object, the fluent API that begins with **`data()`** must support the DTD (XML schema) of an **`XMLData`** object. Alas, this schema could be very different in different applications. A tool such as Fling is handy in creating a fluent API to support different XML schemas and their evolution through the software's lifetime.

It is interesting that previous efforts to introduce XML literals and expressions into programming languages fail to do the DTD specific type checking. Such is the case with XJ [10], an XML extension to Java in which all XML instances belong to the same type. Similarly, XML literals of Visual Basic do not distinguish between various DTDs. Also, attempts to introduce XML into $C^{\#}$ [11] [16, 15] ignored the issue of static type checking of distinct XML objects. To our knowledge, the only language supporting typed XML objects is Haskell [20]. With the current contribution, XML objects with DTD static typing can be easily introduced into Java, $C^{\#}$ and C++, for all DTDs that can be written as a deterministic language. (Note however that a DTD requirement that all rows in the table have the same number of columns, is not context free, for the same reason that the language $a^n b^n c^n$ is not context free.)

A concern often raised against fluent API carry is that the syntactic baggage of parenthesis and a '.' dot operator in each call obscures the syntax of the implemented language. This is true for Java in the implementation we offer. We believe it might be possible to tune the implementation to use fields rather than methods whereby minimizing the baggage to the '.' dot operator. The same is possible in languages such as Eiffel [12], in which it possible to omit the empty parenthesis in calls to methods that do not take parameters. A workaround to omit the parentheses is possible in $C^{\#}$ with getter methods.

We further note that one of the most successful examples of embedding a DSL in a host language, namely of SQL in $C^{\#}$ with LINQ, relies on a fluent API. SQL code excerpts found in $C^{\#}$ are free of any syntactic baggage and use the SQL syntax. An SQL excerpt is converted in a technical iteratively process (which involves little parsing if any) into a fluent API call chain. The definitions beyond this fluent API were handcrafted for the particular syntax. We believe it should be possible to extend this mechanism to support user defined DSLs.

1.2 Previous work

Gil and Levy [5] described the first non-trivial solution of the fluent API problem, recognizing the same class of languages as we do here, and making the same requirements from the type system of the host language as we do. However, in their construction $E(\ell)$ is exponential in the specification of ℓ. Moreover, since their result relies on on a very complex theoretical construction [3], no implementation is provided nor one does seem feasible.

Grigore [8] noticed that bounded below parametric polymorphism, e.g., JAVA's `super` constraints on generics, makes it possible to coerce the type checker into non-constant computation, going up and down the inheritance tree. With this observation, he was able to show that JAVA generics are Turing-complete and solve the fluent API problem for any recursive language ℓ. For a context free grammar language ℓ, his construction produces a polyomially sized $E(\ell)$. However type checking an expression of size n requires $O(n^9)$ time. In Grigore's words,

> *"...the degree of the polynomial is not exactly encouraging. There is room for improvement, and work to be done to achieve a practical parser generator for fluent interfaces."*

Indeed, practical parser generators for fluent APIs restrict themselves to using unbounded parametric polymorphism as we do here.

Fajita [14] is a tool for generating a fluent API definition from a given grammar specification. The class of grammars accepted by Fajita is contained in LL(1), but otherwise unspecified. Also, unlike FLING which generates an AST, Fajita can only be used for language recognition but not for language processing.

In contrast, *Silverchain* by Nakamaru et al. [18] is a real compiler-compiler. However, the class of languages it can process has not been defined. Unlike FLING, *Silverchain* fails on the many common languages that require an unbounded number of ε-transitions (see below Sect. 2).

Another related contribution is by Xu's [22] compiler-compiler for LL(1) languages provided they are given in Greibach Normal Form.

Outline. *Sect. 2 defines deterministic pushdown automata, establishes vocabulary and gives some intuition on the difficulty posed by ε-transitions. With these, we proceed in Sect. 3 to describe how deterministic pushdown automata can be emulated by a realtime device that uses tree encoding data structure, rather than a stack. Sect. 4 demonstrates (using JAVA) how the emulation can be compiled to any unbounded parametric type. The* FLING *tool and its implementation are the subject of Sect. 5. Sect. 6 concludes, mentioning directions for further research.*

2 Pushdown Automata

Intuitively, a pushdown automaton (PDA) is a finite state automaton (FSA) additionally equipped with a stack whose values are drawn from some finite set of stack symbols. A PDA has two kinds of transitions:

1. In a *consuming transition*, the PDA consumes an input letter and pops a symbol from the top of stack. Then, depending on the popped symbol, the input letter, and its current state, the PDA moves into another state, and pushes onto the stack a (possibly empty) sequence of stack symbols.
2. *An ε-transition* is similar to a consuming transition, except that no input letter is consumed: The selection of the next state and the symbols to push in the transition therefore depends solely on the stack's top and the current state of the automaton.

The PDA process the input letter by letter: For each letter, the automaton carries out a single consuming transition, followed by zero or more ε-transitions. There is no upper bound on the number of ε-transitions carried out for a single input letter. As our next example shows, this number may be linear in the input size.

2.1 Example

Consider a simple (balanced) parentheses language defined by the grammar of Fig. 1. In this language each opening parenthesis, '`(`', must be balanced with a closing '`)`', except that a closing square bracket, '`]`', closes *all* previous opening parentheses.

$$\begin{array}{rcl} \langle \mathit{Word} \rangle & ::= & \langle \mathit{Word} \rangle\ \langle \mathit{Word} \rangle \\ & | & \langle \mathit{Balanced} \rangle \\ & | & \langle \mathit{Squared} \rangle \\ & | & \varepsilon \end{array} \qquad \begin{array}{rcl} \langle \mathit{Balanced} \rangle & ::= & \langle \mathit{Balanced} \rangle\ \langle \mathit{Balanced} \rangle \\ & | & \texttt{(}\ \langle \mathit{Balanced} \rangle\ \texttt{)} \\ & | & \varepsilon \\ \langle \mathit{Squared} \rangle & ::= & \texttt{(}^{+}\ \langle \mathit{Balanced} \rangle\ \texttt{]} \end{array}$$

Figure 1 Grammar for a parentheses language in which '`(`' and '`)`' are balanced except that '`]`' closes all currently open parentheses.

To recognize this language a PDA should maintain a stack to count all unclosed opening parentheses. The stack should be cleared when a '`]`' is encountered. One such PDA is depicted in Fig. 2.

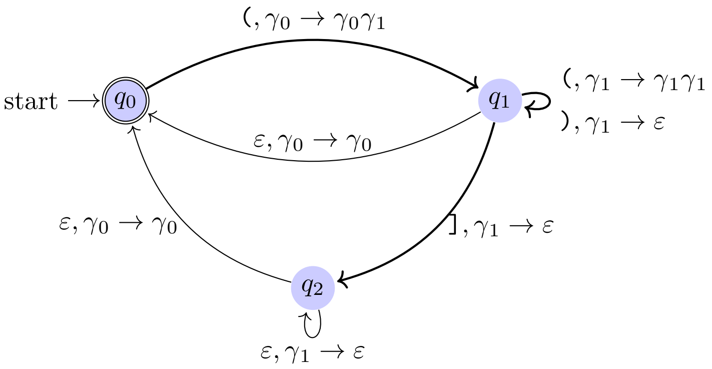

$\texttt{(}, \gamma_0 \to \gamma_0\gamma_1$
start
q_0
q_1
$\texttt{(}, \gamma_1 \to \gamma_1\gamma_1$
$\texttt{)}, \gamma_1 \to \varepsilon$
$\varepsilon, \gamma_0 \to \gamma_0$
$\varepsilon, \gamma_0 \to \gamma_0$
$\texttt{]}, \gamma_1 \to \varepsilon$
q_2
$\varepsilon, \gamma_1 \to \varepsilon$

Figure 2 A deterministic pushdown automaton recognizing the balanced parentheses language of Fig. 1.

The figure uses the usual representation of automata as a multi-graph whose nodes are the states of the *inner* FSA of the PDA, while multi-edges emanating from a node describe the transitions taken by the PDA when being in this state. Edge labels describe the dependency of the transition on both the current input letter and the top stack, *and* the sequence of stack symbols to be pushed in response to these. The figure also employs the convention that consuming transitions are depicted as thicker edges than ε-transitions.

The automaton in the figure can be in one of three inner states: q_0, q_1, and, q_2. It uses of stack symbols, γ_0, marking the bottom of the stack, and γ_1. The number of occurrences of γ_1 on the stack is count of the yet unmatched opening parentheses.

Focus on the self edge of q_1 and the two labels that annotate it.

The *first label* `(`$,\gamma_1 \to \gamma_1\gamma_1$ means that if the automaton is in this state, and if the current input letter is '`(`', and if the stack's top is γ_1, then the automaton carries out a consuming transition, in which the stack top is replaced by $\gamma_1\gamma_1$, hence maintaining the unary representation of n.

The *second label* `)`$,\gamma_1 \to \varepsilon$ over this edge means that if the current input letter is '`)`' then a γ_1 at the top of the stack is popped without being replaced.

Notice that if after popping, the symbol at the top of the stack is also γ_1, then no further processing of this input letter occurs. If however after popping, γ_0 is revealed, then the transition consuming '`)`' is followed by precisely one ε-transition, in which the automaton traverses the edge from q_1 to q_0. The label $\varepsilon, \gamma_0 \to \gamma_0$ on this edge demonstrates the notation for ε-transition: in this ε-transition the popped γ_0 is pushed back to the stack.

Any number of ε-transitions may occur if the input letter is '`]`': state q_1 is the only inner state in which this letter is legal, and the stack top must be γ_1: if this is not the case, the automaton aborts; Otherwise, it moves to inner state q_2, and then follows the self edge of this state in a sequence of n ε-transitions, popping all occurrences of γ_1 from the stack. Finally, the automaton follows another ε-transition which brings the automaton back into its initial internal state q_0.

2.2 Deterministic pushdown automata

We distinguish between deterministic and non-deterministic PDAs: At each point during its computation a *non-deterministic* PDA (NDPDA) may have several legal transitions (be they consuming or ε). For example, the automaton depicted in Fig. 2 is deterministic. An NDPDA chooses among these in the usual non-deterministic fashion: It accepts when there is a non-deterministic run which leads to an accepting state.

▶ **Definition 1** (DPDA). *A* deterministic pushdown automaton *(DPDA) M is a septuple $M = \langle \Sigma, Q, q_0, F, \Gamma, \gamma_0, \delta\rangle$ where Q is a finite set of* automaton states*, $q_0 \in Q$ is the* initial state*, $F \subseteq Q$ is the set of* accepting states*, Γ is a finite set of* stack symbols*, $\gamma_0 \in \Gamma$ is a designated symbol marking the initial* bottom of the stack*, and δ is the (partial)* transition function*:*

$$\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \to Q \times \Gamma^*; \tag{3}$$

A configuration *of a DPDA is a pair $\langle q, s\rangle \in Q \times \Gamma^*$ where q is M's inner FSA state and s is the stack contents.*

A DPDA gives an operational definition of a some (typically infinite) language $\ell \subseteq \Sigma^*$: Given a word $w \in \Sigma^*$, a DPDA M starts consuming input, left to right, at the configuration $\langle q_0, \gamma_0\rangle$. For each input letter it encounters, automaton M carries out a consuming transition and followed by a (possibly empty) sequence of ε-transitions. Automaton M aborts, rejecting the input and announcing $w \notin \ell$, if it either *(i)* tries to consume a letter when the stack is empty, or if *(ii)* the transition function is undefined for the current combination of input letter and top of stack symbol.

Suppose that M is in configuration $\langle q, s\rangle$ and that the stack top is symbol $\gamma \in \Gamma$, i.e., $s = \gamma s'$ where $s' \in \Gamma^*$. Then, if $\delta(q, \sigma, \gamma) = \langle q', \alpha\rangle$ the automaton moves through a consuming transition to configuration $\langle q', \alpha s'\rangle$. Alternatively, the automaton also moves to configuration $\langle q', \alpha s'\rangle$ in through an ε-transition, if $\delta(q, \varepsilon, \gamma) = \langle q', \alpha\rangle$.

A configuration is called *consuming* if M is about to consume an input letter. *Intermediate* configurations are the configurations in which M attempts to carry out an ε transition. Thus, a consuming configuration is one in which the automaton cannot make any ε transitions; an intermediate configuration is one in which such transitions are possible.

It follows from the determinism requirement that we can assume the initial configuration $\langle q_0, \gamma_0 \rangle$ is consuming: If this is not the case, then initially the automaton must choose between a possible ε-transition and a consuming transition.

The automaton accepts, announcing $w \in \ell$, and terminates if after w is consumed in full, it reaches a consuming configuration $\langle q, s \rangle$, where $q \in F$.

Notice that it is technically possible for a DPDA to loop indefinitely via pushing ε-transition, e.g., by repeatedly pushing symbols into the stack, or, by repeatedly replacing the stack's top. This singularity is easy to detect and can be ignored. We tacitly assume that a DPDA always halts.

2.3 Deterministic languages

Unlike FSAs, the expressive power of PDAs depends on whether they are deterministic or not: The set of languages accepted by NDPDAs is exactly the set of languages that can be specified by a context free grammar (CFG). In contrast, the set of languages accepted by DPDAs, also called the set of *deterministic languages*, is exactly the set of languages recognizable by an LR(1) parser [13]. Another important property of deterministic languages is that they are guaranteed to have an unambiguous grammar. (In contrast, some non-deterministic languages are inherently ambiguous, i.e., all CFGs for such a language are ambiguous.)

Parsing algorithms used in practice, e.g., LL(k) and LR(k) employ a deterministic automaton. In fact, the computation in all of the classical LL(1)-, SLR-, LALR-, and LR(1)-parsers is essentially that of a DPDA.

Deterministic languages are all context free, but not all context free languages are deterministic. For example, the language specified by the condition

$$\{\ ww^r \mid w \in \Sigma^+ \wedge w^r \text{ is } w \text{ in reverse order }\}, \tag{4}$$

or, equivalently, the context free grammar

$$\begin{aligned} \langle S \rangle \ ::= &\ a\ \langle S \rangle\ \ a \\ \mid &\ b\ \langle S \rangle\ \ b \\ \mid &\ \varepsilon, \end{aligned} \tag{5}$$

requires a PDA to "guess" when w ends and w^r begins. If the guess is correct, then the automaton must pop symbols of the stack. If it is not, more symbols must be pushed. A non-deterministic automaton can explore both options for each letter in the input. Such guessing cannot be done in a deterministic fashion.

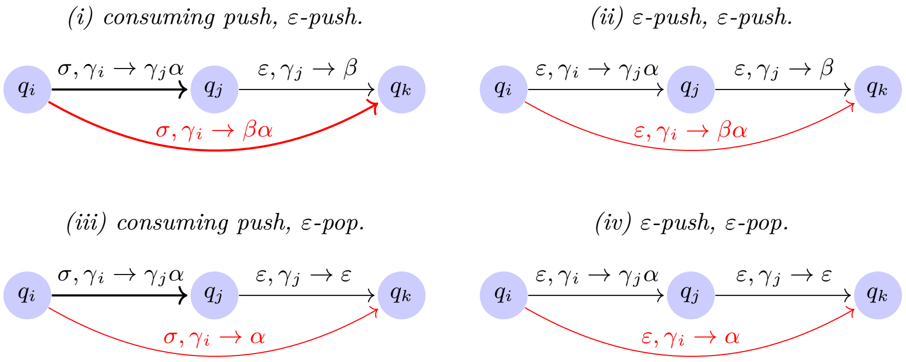
2.4 Simplification of DPDAs

A *real-time* automaton is an automaton that carries out precisely one transition for each input symbol. The challenge in producing a real-time pushdown automaton is in the consolidation of an unbounded number of (ε-) transitions into one. Next we do the first step towards this consolidation:

We say that the transition is *popping* if α is empty, and that it is *pushing* otherwise. We show how a given DPDA can be transformed into an equivalent one, in which pushing transitions are never followed by ε-transitions.

▶ **Lemma 2.** *For every DPDA M, there is a DPDA M' recognizing the same language as M, such that every pushing transition of M' leads to a consuming configuration, in which no further ε-transitions are possible.*

Proof. Consider any particular pushing transition. If no ε-transition follows it, we are done. Otherwise, we consolidate the transition with the subsequent one. We need to consider four cases, as depicted in Fig. 3.


(i) consuming push, ε-push.

(ii) ε-push, ε-push.

(iii) consuming push, ε-pop.

(iv) ε-push, ε-pop.

q_i
$\sigma, \gamma_i \to \gamma_j\alpha$
q_j
$\varepsilon, \gamma_j \to \beta$
q_k
$\sigma, \gamma_i \to \beta\alpha$
q_i
$\varepsilon, \gamma_i \to \gamma_j\alpha$
q_j
$\varepsilon, \gamma_j \to \beta$
q_k
$\varepsilon, \gamma_i \to \beta\alpha$
q_i
$\sigma, \gamma_i \to \gamma_j\alpha$
q_j
$\varepsilon, \gamma_j \to \varepsilon$
q_k
$\sigma, \gamma_i \to \alpha$
q_i
$\varepsilon, \gamma_i \to \gamma_j\alpha$
q_j
$\varepsilon, \gamma_j \to \varepsilon$
q_k
$\varepsilon, \gamma_i \to \alpha$

Figure 3 Four cases of consolidating a pushing transition with a subsequent ε-transition.

The top left of the figure shows case *(i)*, in which the first transition is consuming push and the second transition is an ε-push: A DPDA at state q_i and top stack symbol γ_i consumes letter σ removes γ_i, pushes back a sequence $\gamma_j\alpha$, and moves to state q_j. In this state, it carries out an ε-push in which γ_j is removed and replaced with a sequence β of stack symbols, and moves to state q_k.

The cumulative effect of the two transitions is then: consuming σ, moving from q_i to q_k and pushing the sequence $\beta\alpha$. As shown in the figure, the automaton can be transformed without perturbing its semantics, by replacing the edge $\langle q_i, q_j\rangle$ labeled $\sigma, \gamma_i \to \gamma_j\alpha$ (rendered in black in the figure) with an edge $\langle q_j, q_k\rangle$ with labeled $\sigma, \gamma_i \to \beta\alpha$ (rendered in red).

Notice that edge $\langle q_j, q_k\rangle$ with label $\varepsilon, \gamma_j \to \beta$ is not eliminated in the transformation: In the case that there are other edges leading to q_j, elimination of the edge would change the semantics of the automaton.

Case *(ii)* is similar, except that the first transition is not consuming. In this case, the transformation replaces edge $\langle q_i, q_j\rangle$ with label $\varepsilon, \gamma_i \to \gamma_j\alpha$ with a $\langle q_j, q_k\rangle$ edge with label $\varepsilon, \gamma_i \to \beta\alpha$.

These two cases assume that the second transition pushes a non-empty sequence β of stack symbols. In both cases, the consolidating transition is a pushing consuming transition, which could be consolidated further.

Cases *(iii)* and *(iv)* in Fig. 3 pertain to the situation in which the sequence β is empty, i.e., the second transition is an ε-pop. As shown in the figure, the transformations correspond respectively to cases *(i)* and *(ii)*, with the assumption $|\beta| = 0$. In these cases, the consolidated transition pushes α, which may, or may not be empty. If α is non-empty, then the consolidated transition is a push, which may be consolidated further with a succeeding ε-transition (if present).

We repeatedly apply the transformations depicted in Fig. 3 as long as they are applicable: If $\langle q_i, q_k\rangle$ is an edge that the transformation yields (marked in red in the figure) is pushing, then it must be further consolidated with any ε-edge outgoing from q_k.

Clearly, the process must stop, and when it does, no ε-transitions can follow a pushing transition. ◀

What impact does the transformation described in Lemma 2 have on the size of the automaton? Each step of the transformation consolidates two edges into one. Revisiting

Fig. 3 we see it is not always safe to remove edge $\langle q_j, q_k\rangle$: There might be yet another state $q_{i'}$ (not depicted in the figure) with an edge $\langle q_{i'}, q_j\rangle$. If $\langle q_j, q_k\rangle$ is eliminated then the four case analysis of q_i, q_j and q_k and the edges that connect them cannot be repeated for states $q_{i'}$, q_j and q_k and their edges.

The number of edges in the automaton may sometimes be reduced in the course of the transformation. However, the encoding of the automaton typically increases in size. In cases *(i)* and *(ii)* edge $\langle q_i, q_j\rangle$ with string α is replaced by edge $\langle q_i, q_k\rangle$ with string $\alpha\beta$. If $|\beta| > 1$, the length of the label increases. We argue that this increase is polynomial in the size of the specification of the automaton: The assumption that the automaton halts is tantamount to the claim that no ε-transition can occur twice in the processing of a single input letter, and hence can add $|\beta|$ symbols to each other edge at most once.

3 Realtime emulation of DPDAs with tree encoding

There are three steps in our algorithm for converting a given DPDA M into to a fluent API encoding of $L(M)$, the language recognized by M: *First* we convert M to an equivalent automaton where no pushing transition can be followed by an ε-transition, relying on Lemma 2 from the previous section (Sect. 2). *Second*, in this section we show how M can be encoded in a tree data structure that makes it possible to process an input letter in constant time. Effectively, we emulate the computation of M using this data structure. *Finally*, the following section (Sect. 4) will hows how this data structure and the constant time processing can be compiled to JAVA's type system, using only unbounded parametric polymorphism.

3.1 Encoding configurations as trees

Let $M = \langle\Sigma, Q, q_0, F, \Gamma, \gamma_0, \delta\rangle$ be implicit henceforth, and suppose that $Q = \{q_0, \ldots, q_h\}$ for some $h \geq 0$. Recall that configurations of M are pairs $\langle q, s\rangle \in Q \times \Gamma^*$, storing the inner state q and a string $s \in \Gamma^*$ that represents the entire contents of the stack.

For an intermediate configuration c define c^ε, the *ε-closure* of c, as the consuming configuration obtained from c after carrying out all possible ε-transitions. If c is consuming then $c^\varepsilon = c$. Observe that c^ε depends only on c, and not on the input.

The *encoding of a configuration* $c = \langle q, s\rangle$, written $e = e(c)$, is a complete tree of degree $h+1$: The root node of e carries labels q and α, where α is a prefix of s, i.e., $s = \alpha r$, $r \in \Gamma^*$, and where $|\alpha| > 0$ is bounded by some constant that depends on M, but not on the stack's depth.

More generally, any non-leaf sub-tree e' of e (including e itself) is an encoding of some configuration $c' = \langle q', s'\rangle$, where s' is a suffix of r, and has two labels associated with it:

1. a *state label*, which is the state q',
2. a *stack prefix label* α', a non-empty string of stack symbols whose length does not depend on the input.

We will use the field notation $e.q$ for the state label; $e.\alpha$ for the stack prefix label.

Consider, for example, the DPDA of Fig. 2 recognizing the parentheses language specified by Fig. 1. The configuration $\langle q_1, \gamma_1\gamma_1\gamma_1\gamma_0\rangle$ of this automaton (obtained, e.g., after reading the input prefix "`(((`") is encoded by our algorithm as depicted in Fig. 4.

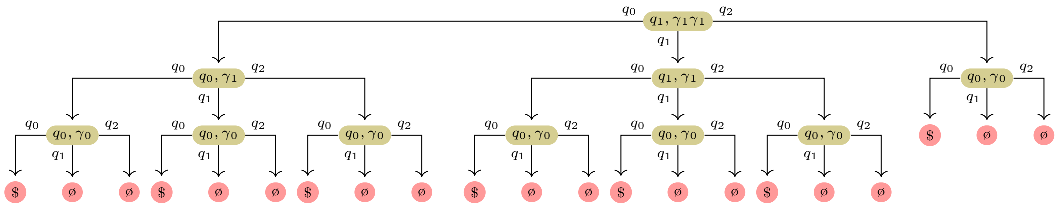


Figure 4 The tree encoding of configuration $\langle q_1, \gamma_1\gamma_1\gamma_1\gamma_0\rangle$ of the automaton of Fig. 2.

(We are oblivious to the question of whether encoding, as defined above, of a configuration is unique; incidentally, our algorithm always creates the same encoding for the same configuration.)

Since the automaton has three inner states q_0, q_1, and, q_2, the tree is, as can be seen in the figure, of degree 3. Examine now the path that starts at the root, and choose the edge labeled q_0 until a leaf is encountered. Collecting the stack prefix labels along this path, yields first the $\gamma_1\gamma_1$, then γ_1 and then γ_0, whereby reconstructing the full stack of the configuration $\langle q_1, \gamma_1\gamma_1\gamma_1\gamma_0\rangle$. Following the q_2 edge will only yield fragments of this stack contents. However, this curious property is not true for all encodings – the consolidation of pushing transitions may make changes to the stack contents, so that s' is not necessarily a suffix of s, even if e' occurs in e.

Notice that the same sub-trees occur several times in the figure. We will see below that even though tree encoding may be exponential in size, it has an efficient, linearly sized, representation in memory.

The emulation does not examine s' directly: Instead, it stores in the i^{th} child of e, written either as $e[i]$ or $e[q_i]$, the encoding of configuration $\langle q_i, s'\rangle^{\varepsilon}$. Intuitively, the emulation does not know to which state q_i automaton M will move when α is removed from the stack, so it stores the result of the computation for all possible values of q_i. More precisely, we have the recursive encoding property

▶ **Property 1** (Recursive encoding property). *If e is the encoding of configuration $c = \langle q, s\rangle$ and $e.\alpha = \alpha$, then $e[i]$ is the encoding of the ε-closure of the configuration of $\langle q_i, s'\rangle$, i.e.,*

$$\forall i = 0, \ldots, h \bullet e[i] = e\left(\langle q_i, s'\rangle^{\varepsilon}\right). \tag{6}$$

In general, configuration $\langle q_i, s'\rangle^{\varepsilon}$ depends on the stack contents s' and therefore computing its encoding may require an unbounded time. The emulation however these said values incrementally: whenever an encoding of a new configuration is generated, the values of its $h + 1$ children are computed from the encoding of the previous configuration and its immediate children, as explained below in Sect. 3.5.

Also notice that the length of the encoded stack decreases as one goes down the tree. Leaves thus encode configurations in which the stack is empty. We allow two kinds of leaves: the special node $\bot$, denoting the (pseudo) configuration of M rejecting the input, and, $\top$, denoting the (pseudo) configuration of accepting it. It will become evident that these special nodes represent acceptance or rejection even in the case that the stack is not emptied.

3.2 Emulating a DPDA with tree encoding

There is only one valid encoding of the initial configuration $\langle q_0, \gamma_0\rangle$, since there is only one way of selecting α. All children of this configuration are leaves. Leaf i is $\top$ if $q_i \in F$ and $\bot$ otherwise. The emulation of M iterates from this initial encoding following Alg. 1.

Algorithm 1 Function emulate(w) – emulate the computation of DPDA M on $w \in \Sigma^*$; returns $\top$ if M accepts w, and $\bot$ otherwise.

```
 1: Function emulate($w$):
 2: Let $e \leftarrow$ newNode($q_0, \gamma_0$)          // encoding of initial configuration $\langle q_0, \gamma_0 \rangle$
 3: For $i \leftarrow 0, \ldots, h$ do                    // fill in the $i^{th}$ child of $e$
 4:    If $q_i \in F$ then                              // $M$ accepting the input
 5:       $e[i] \leftarrow \top$                            // an accepting leaf
 6:    else                                            // $M$ rejecting it
 7:       $e[i] \leftarrow \bot$                            // a rejecting leaf
 8: For $\sigma \in w$ do                              // iterate on input letters, left to right
 9:      $e \leftarrow$ next($e, \sigma$)                  // encoding of next consuming configuration
10:    If $e = \bot$ then                              // $M$ was unable to proceed
11:       Return $\bot$                                // halt emulation rejecting the input
12: If $e.q \in F$ then                                // $M$ terminates in an accepting state
13:    Return $\top$                                   // accept the input
14: Return $\bot$          // else, $M$ terminated in a non-accepting state, reject the input
```

The emulation in the algorithm has three parts: In lines 2–7 the encoding of the initial configuration is computed. Input processing is in lines 8–11 – for each input letter, the emulator calls function next to compute the encoding of the next consuming configuration. If it is determined during the iteration that M does not have a valid transition, then the emulator aborts, rejecting the input. Finally, in lines 12–14, the emulator decides on accepting or rejecting the input, depending on whether M ended in an accepting state.

3.3 Computing the next encoding

The gist of the emulation is in function next(e, σ) (Alg. 2): given e, an encoding of a consuming configuration, and input letter σ, next(e, σ) returns the encoding of the consuming configuration obtained after consuming the input letter σ. This function computes the cumulative effect on the stack and the inner state of the consuming transition from e on σ and all ε-transitions that might follow.

Algorithm 2 Function next(e, σ) – given e, an encoding of a consuming configuration, and input letter σ, returns the encoding of the consuming configuration obtained after consuming σ.

```
1: Function next($e, \sigma$):
2: Let $q, \alpha \leftarrow$ label of $e$          // $e$ is encoding of $\langle q, s \rangle$, $s = \alpha s'$, $s'$ is unknown
3: Let $\gamma \leftarrow$ first($\alpha$)                 // Pop is possible since $\alpha \in \Gamma^+$
4: Let $\beta \leftarrow$ rest($\alpha$)           // $e$ is encoding of $\langle q, s \rangle$, $s = \gamma\beta s'$, $s'$ is unknown
5: If $\delta(q, \sigma, \gamma)$ is undefined then          // $\sigma$-consuming transition is undefined
6:    Return $\bot$                                  // The automaton rejects the input
7: Let $q', \alpha' \leftarrow \delta(q, \sigma, \gamma)$             // Compute the consuming transition
8: continue as in Alg. 4
```

We see that next first (lines 2–4) determines that e is encoding of a configuration $\langle q, \gamma\beta s' \rangle$, where $\gamma \in \Gamma$ is the head of the stack, $\beta \in \Gamma^*$ is the known stack prefix under it, and $s' \in \Gamma^*$

is the unknown remainder of the stack. In principle, next can examine the values of $e[i]$ to determine s', but doing so would lead to a computation that depends on the stack size, and hence on the input, which we would like to avoid.

In lines 5–7, next examines the consuming transition of M. If this transition is undefined, then the input is rejected. Otherwise, next determines that M moves to state q' and replaces γ with the string α in the consuming transition, i.e., M moves to intermediate configuration $c = \langle q', \alpha'\beta s'\rangle$.

Function next uses an auxiliary recursive function, consolidate(e) (Alg. 3), which recursively computes the effect on the stack and the inner state of all ε-transitions that might follow: given e, an encoding of an intermediate configuration c, the function returns the encoding of c^{ε}.

Algorithm 3 Recursive function consolidate(e) – given e, an encoding of an intermediate configuration, returns the encoding of the consuming configuration obtained from e after all ε-transitions were carried out.

```
1: Function consolidate($e$):
2: Let $q, \alpha$ ← label of $e$              // $e$ encodes $\langle q, s\rangle$, $s = \alpha s'$, $s'$ is unknown
3: Let $\gamma$ ← first($\alpha$)              // Pop is possible since $\alpha \in \Gamma^+$
4: Let $\beta$ ← rest($\alpha$)                // $e$ encodes $\langle q, s\rangle$, $s = \gamma\beta s'$, $s'$ is unknown
5: If $\delta(q, \varepsilon, \gamma)$ is undefined then     // No further $\varepsilon$-transitions are possible
6:     Return $e$                                // Automaton is ready to consume
7: Let $(q', \alpha')$ ← $\delta(q, \varepsilon, \gamma)$    // Compute the $\varepsilon$ transition
8: continue as in Alg. 4
```

We see that function consolidate is quite similar to next: It starts (lines 2–4) by determining that e is an encoding of a configuration $\langle q, \gamma\beta s'\rangle$, $\gamma \in \Gamma$, $\beta \in \Gamma^*$ and unknown remainder of the stack $s' \in \Gamma^*$. In lines 5–7, consolidate examines the forthcoming ε-transition of M. However, if this transition is undefined, it is determined no more ε-transitions are possible, and that e is in fact an encoding of a consuming transition. Function consolidate then simply returns e.

The similarity of next and consolidate goes further: after initial processing, they proceed identically: The common part of these two functions is described in Alg. 4, which, given e, an encoding of a configuration $\langle q, \gamma\beta s'\rangle$, and a transition from this configuration to new configuration $\langle q', \alpha'\beta s'\rangle$, returns the encoding of $\langle q', \alpha'\beta s'\rangle^{\varepsilon}$.

3.4 Correctness of the emulation

▶ **Lemma 3.** *If e is the encoding of an intermediate configuration c, then consolidate(e) returns the encoding of c^{ε}*

Proof. Let $c = \langle q, \alpha s'\rangle$, where $e.q = q$ and $e.\alpha = \alpha$. Recall that $|\alpha| \geq 1$ and the decomposition $\alpha = \gamma\beta$, $\gamma \in \Gamma$. By the lemma's assumption $e[i]$ is the encoding of $\langle q_i, s'\rangle^{\varepsilon}$ for $i = 0, \ldots, h$.

First notice that if $\delta(q, \varepsilon, \gamma)$ is undefined, then there is no ε-transition from c, c is a consuming configuration, and $c = c^{\varepsilon}$. In this case no further processing is required and consolidate returns e (line 6 in Alg. 3).

Otherwise the function focuses on the ε-transition leading from q to q' with label $\varepsilon, \gamma \to \alpha'$. In this transition M moves from c to configuration $c' = \langle q', \alpha'\beta s'\rangle$.

Algorithm 4 Common code of functions next (Alg. 2) and consolidate (Alg. 3) – given e, an encoding of an intermediate configuration $c = \langle q, \gamma\beta s' \rangle$, a state $q' \in Q$, and stack prefix α', the code considers an ε-transition from c to configuration $c' = \langle q', \alpha'\beta s' \rangle$, and returns e', the encoding of $(c')^{\varepsilon}$, the ε-closure of c'.

```
 1: If $|\beta| = 0$ then                                   // $|\alpha| = 1$
 2:   If $|\alpha'| = 0$ then                               // This is a popping transition. We encode $\langle q', s' \rangle^{\varepsilon}$
 3:     Return $e[q']$                                      // Configuration $\langle q', s' \rangle^{\varepsilon}$ is encoded by $e[q']$ by (6)
 4:   else                                                  // $|\beta| = 0$, pushing $\varepsilon$-transition: encode $\langle q', \alpha' s' \rangle$
 5:     Let $e' \leftarrow$ newNode$(q', \alpha')$          // $e'$ is encoding of consuming configuration $\langle q', \alpha' s' \rangle$
 6:     For $i \leftarrow 0, \ldots, h$ do                  // Copy the $i^{th}$ child of $e'$ from $e$
 7:       Let $e'[i] \leftarrow e[i]$                       // By definition, configuration $\langle q_i, s' \rangle^{\varepsilon}$ is encoded by $e[i]$ $(= e[q_i])$
 8:     Return $e'$                                         // Encoding of consuming configuration
 9: else                                                    // $|\beta| > 0, |\alpha| > 1$
10:   If $|\alpha'| = 0$ then                               // This is a popping transition. We encode $\langle q', \beta s' \rangle^{\varepsilon}$
11:     Let $e' \leftarrow$ newNode$(q', \beta)$            // $e'$ is encoding of intermediate configuration $\langle q', \beta s' \rangle$
12:     For $i \leftarrow 0, \ldots, h$ do                  // Create the $i^{th}$ child of $e'$
13:       Let $e'[i] \leftarrow e[i]$                       // Other than label $e'$ is the same as $e$
14:     Return consolidate$(e')$                            // Then, continue recursively to yield $\langle q', \beta s' \rangle^{\varepsilon}$
15:   else                                                  // $|\beta| > 0$, pushing $\varepsilon$-transition: encode $\langle q', \alpha'\beta s' \rangle$
16:     Let $e' \leftarrow$ newNode$(q', \alpha')$          // $e'$ is encoding of consuming configuration $\langle q', \alpha'\beta s' \rangle$
17:     For $i \leftarrow 0, \ldots, h$ do                  // Create the $i^{th}$ child of $e'$. We encode $\langle q_i, \beta s' \rangle^{\varepsilon}$
18:       Let $e'[i] \leftarrow$ newNode$(q_i, \beta)$      // Temporarily store the encoding of $\langle q_i, \beta s' \rangle$ in $e'[i]$
19:       For $j \leftarrow 0, \ldots, h$ do                // Create the $j^{th}$ child of $e'[i]$. We encode $\langle q_j, s' \rangle^{\varepsilon}$
20:         Let $e'[i][j] \leftarrow e[j]$                  // Encoding of $\langle q_j, s' \rangle^{\varepsilon}$ is $e[j]$ by (6)
21:       Let $e'[i] \leftarrow$ consolidate$(e'[i])$       // Recursive call to compute encoding of $\langle q_i, \beta s' \rangle^{\varepsilon}$
22:     Return $e'$                                         // Return consuming configuration $\langle q', \alpha'\beta s' \rangle$
```

Function consolidate then carries on in Alg. 4 to compute and return c'^{ε}. If the transition is popping, consolidate calls itself recursively, to compute the effect of further ε-transitions. In the case it is pushing, no recursion is required thanks to the transformation of Lemma 2.

We complete the proof by induction on the length of α.

Inductive base. The case $|\alpha| = 1$, i.e., $|\beta| = 0$ is managed in lines 1–8 of Alg. 4. There are two sub-cases to consider:

Popping transition. If α' is empty, (line 2) then the transition is popping and $c' = \langle q', s' \rangle$. By Property 1 the encoding of c'^{ε} is stored in $e[q']$ which consolidate returns (line 3).

Pushing transition. If α' is not empty (line 4) the transition is pushing, $c' = \langle q', \alpha' s' \rangle$ is consuming and cannot be followed by ε-transitions, i.e., $c'^{\varepsilon} = c'$ (Lemma 2). In choosing the prefix α' for the encoding e' of c', we have that the stack remainder of e and e' are the same, i.e., $s = s'$. Encoding e' is therefore created with label q', α' (line 5) and reusing the children of e (lines 6–7). It is then returned by the function without any further processing (line 8).

Inductive step. If $|\alpha| > 1$ then string β, which is one character shorter than α, is not empty. Function consolidate then proceeds in line 15 with the same two sub-cases:

Popping transition. If α' is empty (line 10), then this is a popping ε transition and $c' = \langle q', \beta s' \rangle$. Function consolidate then defines a new encoding e' (line 11) with the non-empty prefix β. With this selection of stack prefix, the stack remainder of e' is the same as that of e, and e' reuses the children of e (lines 12–13).

The recursive call (line 14) then deals with the subsequent ε-transitions. It returns the correct result by the inductive hypothesis (recall that $|\beta| = |\alpha| - 1$).

Pushing transition. If α' is not empty (line 15), then the ε-transition under consideration is pushing. The automaton reaches configuration $c' = \langle q', \alpha'\beta s'\rangle$, and since no ε-transitions follow a pushing transition, we know that c' is consuming and $c' = c'^{\varepsilon}$.
Function consolidate then generates a new encoding e' with labels q' and α' (line 16). The remainder that follows the stack prefix α' in this case is not s', but rather $\beta s'$. The stack remainder s' is obtained with the aid of an extra level in the tree.
Indeed, the children of e' are labeled with β as a stack prefix (line 18). The stack remainder of each of these h children is s'. We can therefore reuse the children of e in populating the h^2 grandchildren of e' (lines 19–20).
Revisiting line 18, we see that we set the i^{th} child of e' to the encoding of configuration $\langle q_i, \beta s'\rangle$. However, by Property 1, the i^{th} child of e' should contain not the encoding of this configuration, but of its ε-closure.
The recursive call to consolidate in line 21 fixes the children of e': After this call, each of its children stores, as required by Property 1, the encoding of $\langle q_i, \beta s'\rangle^{\varepsilon}$.
As before, the correctness of the recursive call on the children is guaranteed by the inductive hypothesis and the fact that β, the stack prefix of each of these children, is one character short of α. ◀

▶ **Lemma 4.** *Suppose consuming configuration c is encoded by the tree e, and c yields the intermediate configuration c' upon consuming the input letter σ. Then configuration c'^{ε} is encoded by next(e, σ).*

Proof. If $\delta(q, \sigma, \gamma)$ is undefined, then there is no suitable σ-consuming transition from c, hence $c = \bot$. In this case next returns $\bot$ (line 6 in Alg. 2).

Otherwise there is a σ-consuming transition leading to state q' and replacing γ with α'. This case is managed in Alg. 4: Observe that this algorithm does not use the input letter σ. The rest of the proof is identical to the inductive part in the proof of Lemma 3. ◀

▶ **Lemma 5.** *Suppose configuration $c = \langle q, s\rangle$ is encoded by e with label q, α. Then the computation time of consolidate(e) is bounded by $O(|Q|^{1+|\alpha|})$ and does not depend on $|s|$, the stack's depth.*

Proof. Let us examine the body of function consolidate: It calls itself recursively in lines 14 and 21. In the first case the function calls itself once, and in the second it calls itself $|Q|$ times.

In both cases the function is called on a configuration with label containing the string β which is shorter than α, $|\beta| = |\alpha| - 1$. Therefore the depth of the call tree is α, and at the worst case $|Q|$ recursive calls are made in each non-leaf invocation. The number of recursive calls is at most $|Q|^{|\alpha|}$.

The proof is completed by noticing that the amount of work in the body of the function is $O(|Q|)$. ◀

3.5 Use of memory

An encoding is represented by a tree data structure. Functions next and consolidate receive such a tree, and return another tree of this sort. The depth of these trees is linear in the stack size, and since their degree is $h + 1$, their total size may be exponential in the length of the input.

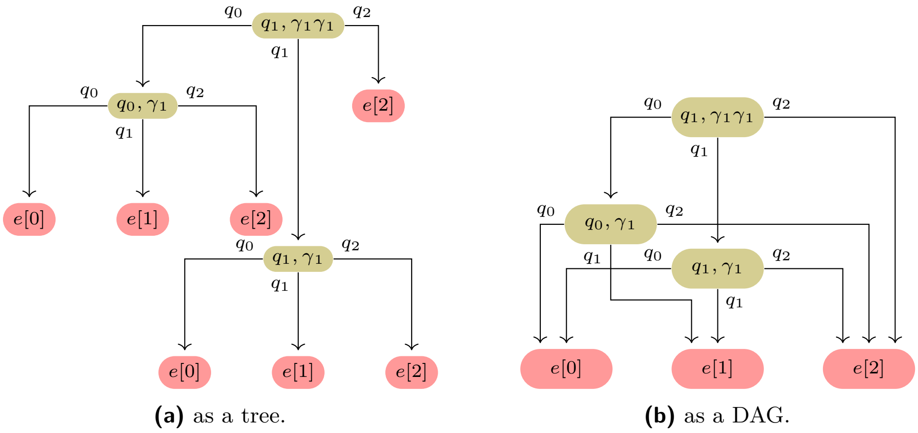
Examining the body of Alg. 4, we see that this exponential blowup is not an issue:

- The code receives as parameters references to children $e[0], \ldots, e[h]$ of encoding e: these references are copied in lines 7, 13, and 20, but never de-referenced.

- The body of next creates new tree nodes in lines 5, 11, 16, and 18 of Alg. 4.

  Examining these lines, we see that the number of these nodes is maximized if the algorithm executes lines 16, and 18. In this case, precisely $|Q|+1$ nodes are created before a recursive call is made. In total, next creates $O(|Q|^{|\alpha|})$ nodes.

We therefore represent encoding trees in memory as a compact DAG. As illustrated in Fig. 5, this encoding is natural: instead of copying children and grandchildren in creating a new encoding, one stores references to these.



(a) as a tree. **(b)** as a DAG.

Figure 5 Representation of a certain encoding as a tree (5a) and as a DAG (5b).

Since the number of primary calls to next is $|w|$, the number of letters in the input (Alg. 1), we have that the memory required for emulating the working of M is linear $|w|$. Fig. 6, showing the compact DAG encoding of in Fig. 4, demonstrates.

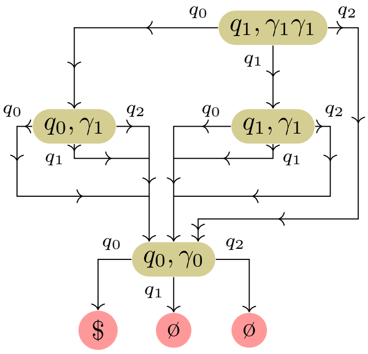


Figure 6 A DAG representation of tree encoding (Fig. 4) of configuration $\langle q_1, \gamma_1\gamma_1\gamma_1\gamma_0\rangle$ of the automaton of Fig. 2.

Comparing the figure to the DAG representation of the same configuration (Fig. 4 above), we see that only the nodes explicitly created by the algorithm are present in the DAG. Indeed, each of these nodes has three children, but as made clear by the figure, the sharing of children makes a significant saving in the size of the representation.

4 Compiling a Tree Encoding to Java

This section shows how to construct a Java fluent API definitions $E(\ell)$ for the language $\ell = \ell(M)$ recognized by a given DPDA M. These definitions should be such that the predicate $w \in \ell$ is equivalent to type checking the fluent API call chain expression $x = x(w)$ (see (1) above). Java is used for the sake of exposition: In essence, we show how to *compile* a DPDA specification into an *abstract* declaration in an unbounded parametric polymorphism type system (UPPTS): Correctness of compilation means that every run of the input specification (some programming language in the general compilation process, but DPDAs here), has an equivalent run in the output specification (written as machine code instructions in general, but as type declaration here), and vice versa.

An implementation of the construction is offered as part of the contribution[1] in the form of an automata compiler which translates a given DPDA to Java definitions. For example, to generate a fluent API for the balanced parentheses language of Fig. 1, a specification of the automaton (Fig. 2) that recognizes it is supplied to the compiler. The specification begins with three **enum** definitions, describing the finite sets of symbols Q, Σ and Γ[2]:

```
enum Q { q0, q1, q2 }
enum Σ { c, ɔ, Ɔ }
enum Γ { γ0, γ1 }
```

Observe the use of letter ‘c’ instead of ‘(’ (which is not a valid method name in Java). Also, letter ɔ (inverted lower case ‘c’) replaces ‘)’ and ‘Ɔ’ replace ‘]’. With these definitions, a Java model of the DPDA is constructed using the Builder design pattern (List. 2).

Listing 2 Supplying to the automata compiler the specification of the DPDA of Fig. 2.

```
DPDA<Q, Σ, Γ> M = new DPDA.Builder<>(Q.class, Σ.class, Γ.class).
  q0(q0). // Starting in state $q_0$
  F(q0). // $q_0$ is an accepting state
  γ0(γ0). // $\gamma_0$ is the bottom of stack marker
  δ(q0, c, γ0, q1, γ0, γ1). // pushing consuming transition $\delta(q_0, \gamma_0, \mathtt{c}) = \langle q_1, \gamma_1\gamma_0 \rangle$
  δ(q1, c, γ1, q1, γ1, γ1). // pushing consuming transition $\delta(q_1, \gamma_1, \mathtt{c}) = \langle q_1, \gamma_1\gamma_1 \rangle$
  δ(q1, ɔ, γ1, q1). // popping consuming transition $\delta(q_1, \gamma_1, \mathtt{ɔ}) = \langle q_1, \varepsilon \rangle$
  δ(q1, null, γ0, q0, γ0). // pushing ε−transition $\delta(q_1, \gamma_0, \varepsilon) = \langle q_0, \gamma_0 \rangle$
  δ(q1, Ɔ, γ1, q2). // popping consuming transition $\delta(q_1, \gamma_0, \mathtt{Ɔ}) = \langle q_2, \varepsilon \rangle$
  δ(q2, null, γ1, q2). // popping ε−transition $\delta(q_2, \gamma_1, \varepsilon) = \langle q_2, \varepsilon \rangle$
  δ(q2, null, γ0, q0, γ0). // pushing ε−transition $\delta(q_2, \gamma_0, \varepsilon) = \langle q_0, \gamma_0 \rangle$
  go(); // having accumulated the specification of $M$, build its model
```

The code in the listing builds in a fluent API fashion: the first three calls in the chain define the initial state, the accepting state, and the initial stack symbol. Then follows a series of calls to the Java function $\delta(\ldots)$. These are a piecemeal specification of the transition function of the modeled automaton: the call `δ(q0,c,γ0,q1,γ0,γ1)` is to say that the automaton in state q_0 and γ_0 at the stack top, moves to state q_1 and pushes back onto the stack γ_0 and then γ_1.

The final call in the above chain to `go()` concludes the construction, and the above code stores the DPDA model in variable `M` of generic class `DPDA`.

We stress that the Java code produced by the automata compiler is not meant to be run: the definitions in $J_{L(M)}$ are incomplete, and even if completed, evaluation of the fluent API expression does not produce any useful value, and will probably fail with run-time error. The generated code is used solely for type checking. Sect. 5 describes a more practical tool in which the evaluation of $x(w)$ returns the AST of word w.

[1] `https://github.com/OriRoth/jdpda`

[2] This, just as other code excerpts in the paper, is drawn from the implementation. Note that Java supports UTF characters

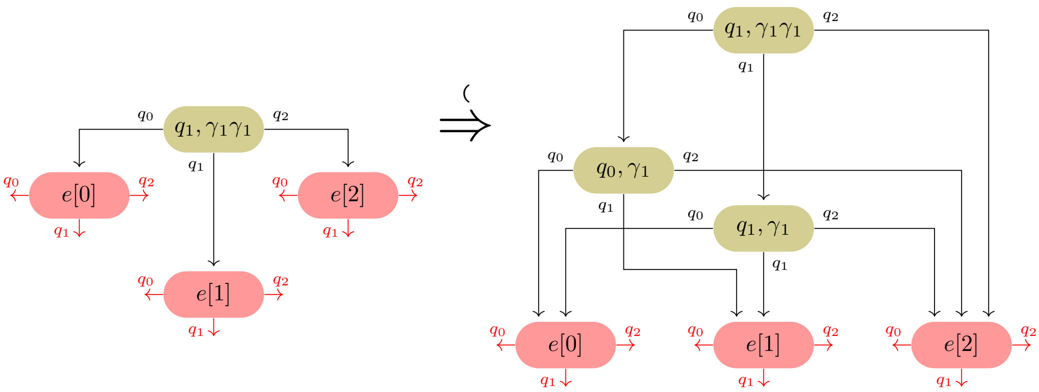


Figure 7 Partial evaluation of $\mathsf{next}(e(\langle q_1, \gamma_1\gamma_1 s'\rangle),\text{‘(’})$ presented as a DAG whose leaves are the implicit parameters (sub-trees) $e[0]$, $e[1]$ and $e[2]$.

4.1 Intuition

Compilation is based on the emulation algorithm described in the previous section. The main idea is partial evaluation: Instead of compiling to the type system the many details in Alg. 2, Alg. 3, and Alg. 4, the compiler relies on partial evaluation and caching. The Java type system is only used on the partially evaluation form.

Let $q \in Q$, $\alpha \in \Gamma^+$ and σ be fixed, and let e be defined by q and α and some stack remainder s'. Examining the three algorithms that make function $\mathsf{next}(e, \sigma)$ we see that for any such fixture the function constructs the same tree from the sub-trees $e[0], \ldots, e[h]$.

Consider for example the language of Fig. 1 and its DPDA (Fig. 2.) Suppose that M is in the consuming configuration $\langle q_1, \gamma_1\gamma_1 s'\rangle$, where $s' \in \{\gamma_0\gamma_1\}^*$ and that e is an encoding of this configuration with stack prefix $\alpha = \gamma_1, \gamma_1$. Indeed, trees $e[0]$, $e[1]$, and $e[2]$ depend on the actual contents of s, but next never inspects the contents of these trees.

For these values of q, α and for $\sigma =$‘(’, the call $\mathsf{next}(e, c)$ involves a call to $\mathsf{consolidate}$, which may even call itself recursively. Working out the details, one finds that $\mathsf{next}(e, c)$ is in effect the transformation depicted in Fig. 7.

As shown in the figure, the net effect of $\mathsf{next}(e(\langle q_1, \gamma_1\gamma_1 s'\rangle),\text{‘(’}$ is to transform the tree encoding on the left to the tree encoding in the right. This transformation does not examine nodes $e[0]$, $e[1]$, and $e[2]$ (marked in red in the figure). Partial evaluation of $\mathsf{next}(e(\langle q_1, \gamma_1\gamma_1 s'\rangle),\text{‘(’})$ is therefore an oblivious function of $e[0]$, $e[1]$ and $e[2]$.

More generally, we have that for every q, α and σ the partial evaluation of next has a simple DAG structure that represents an oblivious function of $e[0], \ldots, e[h]$. Our DPDA to Java compiler computes and caches these DAGs for each combination of q, α, and σ it encounters.

Let $D(q, \alpha, \sigma)$ denote the DAG defined by q, α, and σ. Note that function $D(q, \alpha, \sigma)$ may also assume the special value $\perp$ in the case that $\delta(q, \gamma, \sigma)$ (γ being the first symbol in α) is undefined.

We argue that D has a finite representation: q and σ are drawn from finite sets. To see that the number of distinct stack prefixes α is finite, examine again lines 5, 11, 16, and 18 of Alg. 4 in which $\mathsf{newNode}$ is invoked: In all of these the stack prefix label attached to the newly created node is either a label α' of a certain transition of M, or β, a suffix of an existing label.

4.2 Structure of the encoding

The Java code emitted by the compiler contains these definitions:

1. A designated type that represents the (leaf) encoding $\bot$ – the encoding of a rejecting automaton.
   ```
   interface ∅ { }
   ```
   Notice that this designated type is not parametric.
2. A designated type that represents the (leaf) encoding $\top$ – the encoding of an accepting automaton.
   ```
   interface $ { }
   ```
   Again, this designated type is not parametric.
3. Parametric state types, each designating an encoding label $\langle q, \alpha \rangle$, $q \in Q$ and $\alpha \in \Gamma^+$, and each taking $|Q|$ unbounded type parameters. As a matter of convention, the name of this type is the string of symbols $q\alpha$ separated by underscores: For example, for the encoding label $\langle q_1, \gamma_1\gamma_1 \rangle$ (described in Fig. 7) the compiler generates the parametric type
   ```
   interface q1_γ1_γ1<e0, e1, e2> {...}
   ```
4. A start variable (named `__` in the implementation), from which fluent API chains start. This variable is one of the generated parametric state types: specifically, the state type that bears the label of the initial configuration $\langle q_0, \gamma_0 \rangle$.
   The values of the parameters are either the rejection or the accepting designated types, depending on whether the state corresponding to the parameter is accepting or not (as in Alg. 1). In our example,
   ```
   q0_γ0<$, ∅, ∅>__= ...;
   ```

With these we have a representation of the tree encodings of configurations of M as a type obtained from the instantiation of of an appropriate Java generics: An encoding e with state label q, state prefix label $\alpha = \gamma_1\gamma_2 \cdots \gamma_k$ and children $e[0], \ldots, e[h]$ is represented by the following instantiation of the parametric state type

```
q_γ1_γ2···_γk<τ0, τ1, ..., τh>
```

with actual type parameters $\tau_0, \tau_1, \ldots, \tau_h$ being the type representation of child encodings $e[0], \ldots, e[h]$.

Alg. 1, the emulation of M with tree encoding is done step by step by the fluent API call chain: If M is in a configuration c after reading the input $\sigma_1\sigma_2 \cdots \sigma_i$, and e is the encoding of c. Then, the type of the fluent API call chain

$$\texttt{\_\_.}\sigma_1\texttt{().}\sigma_2\texttt{().}\cdots\texttt{.}\sigma_i\texttt{()} \tag{7}$$

is precisely the type representation of e: A call to a method named σ`()` in the chain represents the consumption of input letter σ; the type that the method returns is the type encoding of the subsequent consuming configuration. It is the chief duty of the compiler to correctly generate this type. For each generic class t with certain q and α, the compiler examines every $\sigma \in \Sigma$:

- If $D(q, \alpha, \sigma) \neq \bot$, the compiler generates a method σ`()` in t, and uses $D(q, \alpha, \sigma)$ to specify the return type of σ in terms of the type parameters of t. In the example of Fig. 7, the compiler generates in **`q1_γ1_γ1<e0, e1, e2>`** a method `c()` whose return type is the Java representation of the DAG in Fig. 7:
  ```
  q1_γ1_γ1<q0_γ1<e0, e1, e2>, q1_γ1<e0, e1, e2>, e2> c();
  ```
  Examine, e.g., the first type argument of the return type of **`c()`**: The value of this type argument is **`q0_γ1<e0, e1, e2>`** which is exactly the q_0 child of the root of the DAG of the figure.

- Conversely, if $D(q, \alpha, \sigma) = \bot$, the compiler sets the return type of $\sigma()$ to the rejection type $\emptyset$.

After constructing the variable `M` in List. 2 the call `M.compile()` returns a text including Java definitions for the fluent API of `M`. The code is shown in List. 3.

Listing 3 Output of the automata compiler for the DPDA of Fig. 2.

```
interface ø { } // designated type denoting rejection
interface $ { } // designated type denoting acceptance

q0_γ0<$, ø, ø> __ = null; // Initial configuration

interface q0_γ0<e0, e1, e2> { // (1) configurations $\langle q_0, \gamma_0 s' \rangle$
  q1_γ1_γ0<e0, e1, e2> c(); // ‘c’ is the only input letter allowed in this state
 $ $(); // Input may end in this configuration
}
interface q1_γ1_γ0<e0, e1, e2> { // (2) configurations $\langle q_1, \gamma_1 \gamma_1 s' \rangle$
  q1_γ1_γ1<q0_γ0<e0, e1, e2>, q0_γ0<e0, e1, e2>, q0_γ0<e0, e1, e2>> c();
  q0_γ0<e0, e1, e2> ɔ();
  q0_γ0<e0, e1, e2> Ɔ();
}
interface q1_γ1_γ1<e0, e1, e2> { // (3) configurations $\langle q_1, \gamma_1 \gamma_1 s' \rangle$
  q1_γ1_γ1<q0_γ1<e0, e1, e2>, q1_γ1<e0, e1, e2>, e2> c();
  q1_γ1<e0, e1, e2> ɔ();
  e2 Ɔ();
}
interface q0_γ1<e0, e1, e2> extends $ { // (4) Configurations $\langle q_0, \gamma_1 s' \rangle$
  $ $(); // Input may end in this configuration
  // No other input letter is legal here
}
interface q1_γ1<e0, e1, e2> { // (5) configurations $\langle q_1, \gamma_1 s' \rangle$
  q1_γ1_γ1<e0, e1, e2> c();
  e1 ɔ();
  e2 Ɔ();
}
```

Observe in the code the two designated types for acceptance and the start variable whose type is the initial configuration; then follow five parametric state types (interface `q0_γ<e0, e1, e2>` through `q1_γ1<e0, e1, e2>`). Also notice that classes of configurations with state $q_0 \in F$ offer a function `$()` returning the special type \$.

4.3 Correctness

With this construction we can argue

Lemma 6. *The* Java *expression* `__`$.\sigma_1().\sigma_2().\cdots.\sigma_i()$

1. *does not compile, producing a missing method error message, if M aborts on $\sigma_1\sigma_2\cdots\sigma_j$, $1 <= j <= i$, or,*
2. *is of a type that represents the encoding of the configuration of M after reading $\sigma_1\sigma_2\cdots\sigma)_i$, including the special type* `interface ø` *denoting rejection.*

Proof. Mundane, by induction on i: The type of `__` represents the encoding of the initial configuration. The return type of the call to method σ_i is, by construction, the type representation of the encoding of M. ◀

In addition, if $q \in F$, the code generator adds to type t a method `$()` whose return type is `interface $`, the special type denoting acceptance. This method marks the end of input in the emulation.

With this addition, we have that the fluent API chain

$$\_\_.\sigma_1().\sigma_2().\cdots\sigma_n().\$()$$

type checks, if and only if, word $\sigma_1\sigma_2\cdots\sigma_n \in L(M)$.

5 Fluent-API Generation in Fling

The contributions of tree encoding (Sect. 3) and automata compilation strategy (Sect. 4) made it possible to develop Fling– a compiler-compiler in the vein of e.g., YACC and ANTLR: Fling receives its input in a form suitable for clients – an EBNF grammar rather than DPDA specification. It converts the grammar into an automaton, and generates the fluent API type definitions for the language specified by the grammar.

Fling improves on the automata type compiler: Having recognized the fluent API chain as a valid word, Fling also generates code to construct, at runtime, the AST of the chain, and provides clients with means for traversing this tree.

Fling is more limited than the automata compiler, since it can only process LL(1) grammars. The class of languages that can be expressed by such grammars is strictly contained in the class of deterministic languages, which can all be recognized by the automata compiler. However, the restriction to LL(1) grammars is not inherent – extending Fling to support LR(1) grammars (and hence all deterministic languages) is technical (though laborious): One needs to re-implement the classical LR(1) parser generator to produce its output in the format expected by a DPDA compiler. The compiler-compiler features of Fling that we describe here are applicable regardless of the parsing engine.

5.1 Embedding Datalog in Java using Fling

Our open source implementation of Fling[3] includes examples of a dozen or so (small) languages. Here we describe in brief the embedding of Datalog [2] in Java.

Recall that Datalog is a simpler version of Prolog [4] in that predicates cannot be nested. List. 4 is a reminder of the syntax of Datalog, depicting a simple program to manage the ancestral relation, including two facts, three rules, and one query.

Listing 4 A Datalog program managing an ancestral relation.

```
parent(john,bob).
parent(bob,donald).
ancestor(adam,X).
ancestor(A,B) :- parent(A,B).
ancestor(A,B) :- parent(A,C), ancestor(C,B).
ancestor(john,X)?
```

List. 5 is the Java fluent API equivalent of the Datalog program in List. 4. Code comment show the Datalog equivalent of fragments of the call chain.

Listing 5 A fluent API specification of the Datalog program of List. 4.

```
Datalog program = datalog.
  fact("parent").of("john", "bob"). // fluent API of parent(john,bob).
  fact("parent").of("bob", "donald"). // fluent API of parent(bob,donald).
  always("ancestor").of(l("adam"), v("X")). // fluent API of ancestor(adam,X)
  infer("ancestor").of(v("A"), v("B")). // fluent API of ancestor(A,B)
    when("parent").of(v("A"), v("B")). // fluent API of :- parent(A,B).
  infer("ancestor").of(v("A"), v("B")). // fluent API of ancestor(A,B)
    when("parent").of(v("A"), v("C")). // fluent API of :- parent(A,C)
    and("ancestor").of(v("C"), v("B")). // fluent API of , ancestor(C,B).
  query("ancestor").of(l("john"), v("X")); // fluent API of ancestor(john,X)?
```

To create the fluent API demonstrated in List. 5, we start with the names of the methods involved in the chain. These are precisely the terminals (tokens) of the grammar that

[3] https://github.com/OriRoth/fling

generated the fluent API. We therefore write an `enum` definition that enumerates all these methods:

```
enum Terminals implements Fling.Terminals { infer, fact, query, of, and, when, always, v, l }
```

Notice that the methods in fluent API for writing embedded DATALOG code take parameters, `infer("ancestor")` and `of("bob", "donald")`. To understand why, recall that grammars of programming languages such as PASCAL [21] and C++ use two kinds of non-terminals:

- Keywords such as **`begin`**, **`var`**, punctuation such as '`(`', '`;`', '`:`', and operators such as '`;`' can appear in only one form in the program code. We call these *vacuous tokens.* Vacuous tokens can appear in programs in only one way; they serve as parsing aide in the concrete grammar, but are omitted from the abstract syntax tree.
- In contrast, tokens such as *StringLiteral* and *Identifier* carry additional information: A string literal token carries its content, and an identifier literal token carries its name. We call tokens of this sort *informational tokens.*

Grammars of fluent APIs tend to use informational tokens more than vacuous tokens. Compiler compilers such as YACC use a distinct lexical analyzer to specify the many shapes any informational token may take. FLING has no accompanying lexical analyzer. However, since terminals of fluent API are method names, the contents of an informational token is supplied as argument to the method. For example, an identifier token in a fluent API is typically written as `id("fubar")`.

Fig. 8 is the EBNF grammar of the fluent API used for embedding DATALOG in JAVA (e.g., List. 5). The grammar makes frequent use of informational tokens: writing `l("thingy")` is to say that the string parameter is a literal, while `v("thingy")` is to say that it is a literal. Also, the ⟨*Fact*⟩ symbol is composed of two informational tokens: method `fact` in which the DATALOG predicate name is supplied as argument, and method `of` in which the literal parameters are supplied.

$$\begin{aligned}
\langle \mathit{Program}\rangle &::= \langle \mathit{Statement}\rangle^{+}\\
\langle \mathit{Statement}\rangle &::= \langle \mathit{Fact}\rangle \mid \langle \mathit{Query}\rangle \mid \langle \mathit{Rule}\rangle\\
\langle \mathit{Fact}\rangle &::= \texttt{fact}(\langle \mathit{String}\rangle)\ \texttt{of}(\langle \mathit{Literal}\rangle^{*})\\
\langle \mathit{Query}\rangle &::= \texttt{query}(\langle \mathit{String}\rangle)\ \texttt{of}(\langle \mathit{Term}\rangle^{*})\\
\langle \mathit{Rule}\rangle &::= \langle \mathit{Bodyless}\rangle \mid \langle \mathit{WithBody}\rangle\\
\langle \mathit{Bodyless}\rangle &::= \texttt{always}(\langle \mathit{String}\rangle)\ \texttt{of}(\langle \mathit{Term}\rangle^{*})
\end{aligned}$$

$$\begin{aligned}
\langle \mathit{WithBody}\rangle &::= \langle \mathit{RuleHead}\rangle\ \langle \mathit{RuleBody}\rangle\\
\langle \mathit{RuleHead}\rangle &::= \texttt{infer}(\langle \mathit{String}\rangle)\ \texttt{of}(\langle \mathit{Term}\rangle^{*})\\
\langle \mathit{RuleBody}\rangle &::= \langle \mathit{FirstClause}\rangle\ \langle \mathit{AdditionalClause}\rangle^{*}\\
\langle \mathit{FirstClause}\rangle &::= \texttt{when}(\langle \mathit{String}\rangle)\ \texttt{of}(\langle \mathit{Term}\rangle^{*})\\
\langle \mathit{AdditionalClause}\rangle &::= \texttt{and}(\langle \mathit{String}\rangle)\ \texttt{of}(\langle \mathit{Term}\rangle^{*})\\
\langle \mathit{Term}\rangle &::= \texttt{v}(\langle \mathit{String}\rangle) \mid \texttt{l}(\langle \mathit{String}\rangle)
\end{aligned}$$

Figure 8 EBNF grammar of embedding DATALOG in JAVA.

Observe that the same token may take different arguments in different contexts, e.g., token `of` any number of plain strings when it is part of a ⟨*Fact*⟩ , and any number of ⟨*Term*⟩ values when it appears as part of a ⟨*Rulehead*⟩ . Notice that symbol ⟨*Term*⟩ is also defined by the grammar in Fig. 8: In general, the information that accompanies a token is not limited to lexical values, and may be defined by its own grammar.

The specification of the grammar of Fig. 8 requires an `enum` definition of the list of symbols

```
enum Symbols implements Fling.Symbols {
  Program, Statement, Rule, Query, Fact, Bodyless, WithBody,
  RuleHead, RuleBody, FirstClause, AdditionalClause, Term }
```

List. 6 uses `enum Tokens` and `enum Symbols` in a fluent API chain of methods used to describe to FLING the grammar in the Fig. 8.

Listing 6 A fluent API specifying the Datalog grammar of Fig. 8.

```
BNF bnf = bnf(Terminals.class, Symbols.class). // Create a BNF with these terminals and symbols
  start(Program). // ⟨Program⟩ is the start symbol
  derive(Program).to(oneOrMore(Statement)). // ⟨Program⟩ ::= ⟨Statement⟩+
  specialize(Statement).into(Fact, Query, Rule). // ⟨Statement⟩ ::= ⟨Fact⟩ | ⟨Query⟩ | ⟨Rule⟩
  derive(Fact).to(fact.with(String)).and(of.many(String)).
    // ⟨Fact⟩ ::= fact(⟨String⟩) of(⟨Literal⟩*)
  derive(Query).to(query.with(String)).and(of.many(Term)).
  // ⟨Query⟩ ::= query(⟨String⟩)of(⟨Term⟩*)
  specialize(Rule).into(Bodyless, WithBody). // ⟨Rule⟩ ::= ⟨Bodyless⟩ | ⟨WithBody⟩
  derive(Bodyless).to(always.with(String)).and(of.many(Term)).
    // ⟨Bodyless⟩ ::= always(⟨String⟩) of(⟨Term⟩*)
  derive(WithBody).to(RuleHead).and(RuleBody).
    // ⟨WithBody⟩ ::= ⟨RuleHead⟩ ⟨RuleBody⟩
  derive(RuleHead).to(infer.with(String)).and(of.many(Term)).
    // ⟨RuleHead⟩ ::= infer(⟨String⟩) of(⟨Term⟩*)
  derive(RuleBody).to(FirstClause).and(oneOrMore(AdditionalClause)).
    // ⟨RuleBody⟩ ::= ⟨FirstClause⟩ ⟨AdditionalClause⟩*
  derive(FirstClause).to(when.with(String)).and(of.many(Term)).
    // ⟨FirstClause⟩ ::= when(⟨String⟩) of(⟨Term⟩*)
  derive(AdditionalClause).to(and.with(String)).and(of.many(Term)).
    // ⟨AdditionalClause⟩ ::= and(⟨String⟩) of(⟨Term⟩*)
  derive(Term).to(and.with(String)).and(of.many(Term));
    // ⟨Term⟩ ::= l(⟨String⟩)| v(⟨String⟩)
```

Fling offers its own DSL, written in fluent API style, for grammar specification. The following notes, along with code comments in the listing should make this language clear: Token **derive** in Fling's DSL denotes a grammar derivation; the information it carries is the left hand side of a derivation rule. Information attached to token **to** (that follows **derives**) is the first element in the right hand side of the rule. In writing **to(infer.with(String))** we use informational token **to** to specify that the grammar admits an **infer** token with information of type **String**. Similarly, writing **and(of.many(Term)** we use informational token **and** to specify that the grammar admits an **of** token with information that consists of any number of occurrences of symbol **Term**.

5.2 Code generation

Having created a BNF description of Datalog as in List. 6, the execution of **bnf.generate()** will make Fling generate all the definitions required for the code in List. 5 to compile correctly.

Moreover, Fling will generate definitions that make it possible to analyze Datalog programs produced by the fluent API. Variable **program** of type **Datalog** produced in List. 5 is an abstract syntax tree (AST) of the program.

As part of the code generation process, similarly to JAMOOS [7] and SOOP [6] Fling implements an AST class for every symbol in the grammar specification. These classes uses lists for repetitions, omit vacuous tokens, and may stand in inheritance relation: The sub-expression

```
specialize(Statement).into(Fact, Query, Rule)
```

in List. 6 specifies not only grammatical alternation but also inheritance, i.e., class **Fact** inherits from abstract class **Statement**.

Along with these AST classes, Fling generates a template of an AST visitor that clients may use for processing a DSL program supplied in fluent API form. List. 7 demonstrates the use of this visitor to actually run from within Java the Datalog program of List. 4.

Listing 7 `ASTVisitor` running Datalog programs created via fluent API using Jatalog.

```java
ASTVisitor runner() {
  Jatalog j = new Jatalog();
  return new ASTVisitor(DatalogAST.class) {
    public boolean visit(Fact f) throws DatalogException {
      j.fact(f.fact, f.of);
      print(f);
      return true;
    }
    public boolean visit(Bodyless r) throws DatalogException {
      j.rule(Expr.expr(r.always, r.of1));
      print(r);
      return true;
    }
    public boolean visit(WithBody r) throws DatalogException {
      j.rule(Expr.expr(r.infer, r.of1), getExprRightHandSide(r));
      print(r);
      return true;
    }
    public boolean visit(Query q) throws DatalogException {
      print(q);
      print(j.query(Expr.expr(q.query, q.of)));
      return true;
    }
  };
}
```

Our demonstration of Datalog employs Jatalog[4], an open source Datalog engine to make the Datalog embedding complete. The first three methods in the visitor object returned by List. 7 store (and print) facts and rules in the Jatalog engine. The last method asks the engine to compute the query, and then prints the result.

When this visitor is applied to variable **`program`** we obtain the output of shown in List. 8.

Listing 8 Output of List. 7 when run on the embedded Datalog program produced by the fluent API of List. 5.

```
Jatalog:Fling$ parent(john,bob).
Jatalog:Fling$ parent(bob,donald).
Jatalog:Fling$ ancestor(A,B) :- parent(A,B).
Jatalog:Fling$ ancestor(A,B) :- parent(A,C), ancestor(C,B).
Jatalog:Fling$ ancestor(john,X)?
---[{X=bob}, {X=donald}]
```

6 Discussion and Further Work

We showed that any deterministic pushdown automaton can be emulated by a realtime device using the tree encoding data structure. The tree encoding is exponential in the input size, but has a compact DAG representation. An interesting topic in theory of automata is investigation of this data structure, comparing its computational power with that of a stack. For example, one can ask whether PDAs, deterministic or not, can recognize more languages if the pushdown structure is replaced with that of a tree or a DAG.

A crucial observation in our construction is that the tree data structure can be written as a multi-level instantiation of a generic datatype, and that the kind of tree transformations applied in the emulator can also be written as the return type of methods in this data type. This observation was demonstrated in the automata compiler, which in turn made possible the theoretical result of encoding DCFLs in an unbounded polymorphic type system.

The automata compiler was employed in Fling– the first general and practical compiler-compiler of fluent APIs. Fling can be easily extended to support other programming languages, including ML [17], C$^{\#}$, and C++ . With some engineering effort Fling can be extended to LR(1) grammars as supported by YACC does.

[4] `https://github.com/wernsey/Jatalog`

Fling is weaker in its expressive power than the DPDA compiler, supporting only LL(1) grammars. However, we observed the construction in Fling is slightly more useful, in that auto-completion is more accurate. The fluent API problem, as defined here, does not concern itself with auto completion. It would be interesting to study a version of the in which an auto completion feature is required.

In a case study, we showed how Fling is used for embedding Datalog in Java, and how such programs can be written and run from within Java.

Our description of the embedding, Fling itself, and of the automata compiler also demonstrate the usability of fluent API and the specific syntax flavor they induce. See listings 1, 2, 5 and 6. In the context of this style, we coined the term " informational tokens". Unlike traditional lexical analyzers, in Fling information that these tokens carry is not restricted to plain literals and may be defined by their own grammar. In this respect, Fling supports nested and even recursive grammars: The information that a token carries may be defined by the grammar in which the token participates.

We believe that the theoretical result may be also useful in designing extensible languages and languages whose syntax may be extended by their programs. Designers of such languages may choose to use the type system instead of a dedicated parsing engine for the unknown portion of their grammar.

In our implementation we encountered a weakness of the implementation of the to the discussion type system in the javac compiler. As noticed previously by Gil and Levy [5], this compiler represents instantiated generics by value, and admits no sharing.

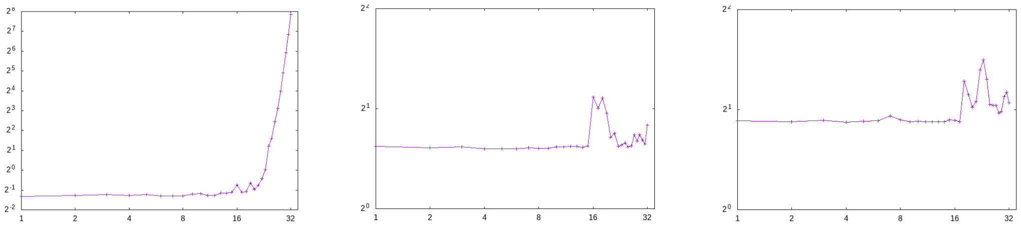
Consider for example the test program in List. 9.

Listing 9 Fluent interfaces exponential type complexity test case.

```java
public class BinaryTreeTypeTest {
  interface Binary<Left,Right> {
    Binary<Binary<Left,Right>,Binary<Left,Right>> b();
  }
  public static void main(String[] args) {
    System.out.println(((Binary<?,?>)null).b().b().b()....b()); // Use chain of length n
  }
}
```

Type **interace Binary** is a generic type. Method **b()** in this type is such that in a given instantiation **Binary**, it returns an instantiation of this type which is twice the size. Thus, a sequence of n fluent calls to **b()**, as in function **main** in the figure, will return an instantiation of **Binary** of size 2^n.

Fig. 9 shows the compile time of List. 9 as a function n on various compilers. Measurements were conducted on Java 1.8.0_131, ECJ 3.11.1 and GCJ 6.3.0.



(a) javac compiler (Oracle). (b) Eclipse Java compiler (ECJ). (c) Gnu Java compiler GCJ.

Figure 9 Compilation time in seconds of BinaryTreeTypeTest List. 9 vs. length of fluent API call chain on various compilers.

As can bee seen in the figure, compilation time grows exponentially on javac. With ECJ and GCJ, these remain constant. Perhaps this work would encourage the makers of javac to use the compact DAG representation of types demonstrated above in Fig. 5.

References

1 Ken Arnold and James Gosling. *The Java Programming Language*. Addison Wesley, 1996.

2 Stefano Ceri, Georg Gottlob, and Letizia Tanca. *Logic programming and databases*. SVNY, 1990.

3 Bruno Courcelle. On jump-deterministic pushdown automata. *Theory of Computing Systems*, 11(1):87–109, 1977.

4 Pierre Deransart, Laurent Cervoni, and AbdelAli Ed-Dbali. *Prolog: The Standard: reference manual*. Springer-Verlag, 1996.

5 Yossi Gil and Tomer Levy. Formal language recognition with the Java type checker. In *30th European Conference on Object-Oriented Programming (ECOOP 2016)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

6 Yossi Gil and David H Lorenz. SOOP – A Synthesizer of an Object-Oriented Parser. *TOOLS'95 Europe*, pages 81–96, 1995.

7 Yossi Gil and Yuri Tsoglin. JAMOOS - A Domain-Specific Language for Language Processing. *Journal of Computing and Information Technology*, 9(4):305–321, 2001.

8 Radu Grigore. Java generics are Turing complete. In Andrew D. Gordon, editor, *(POPL'17)*, pages 73–85, 2017.

9 Zvi Gutterman. Symbolic Pre-computation for Numerical Applications. Master's thesis, Technion, 2004.

10 Matthew Harren, Mukund Raghavachari, Oded Shmueli, Michael G Burke, Rajesh Bordawekar, Igor Pechtchanski, and Vivek Sarkar. XJ: facilitating XML processing in Java. In *Proceedings of the 14th international conference on World Wide Web*, pages 278–287. ACM, 2005.

11 Anders Hejlsberg, Scott Wiltamuth, Peter Golde, and Mads Torgersenby. *The C# Programming Language*. Addison-Wesley Publishing Company, 2nd edition, 2003-10.

12 ISE. *ISE EIFFEL The Language Reference*. ISE, 1997.

13 Donald E Knuth. On the translation of languages from left to right. *Information and control*, 8(6):607–639, 1965.

14 Tomer Levy. A Fluent API for Automatic Generation of Fluent APIs in Java. Master's thesis, Technion, 2016.

15 Erik Meijer and Brian Beckman. Xlinq: Xml programming refactored (the return of the monoids). In *Proceedings of XML*, volume 5, 2005.

16 Erik Meijer, Wolfram Schulte, and Gavin Bierman. Unifying tables, objects, and documents. In *Workshop on Declarative Programming in the Context of Object-Oriented Languages*, pages 145–166. Citeseer, 2003.

17 R. Milner and M. TofteD. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

18 Tomoki Nakamaru, Kazuhiro Ichikawa, Tetsuro Yamazaki, and Shigeru Chiba. Silverchain: a fluent API generator. *Proceedings of the 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences - GPCE 2017*, pages 199–211, 2017. URL: http://dl.acm.org/citation.cfm?doid=3136040.3136041.

19 Bjarne Stroustrup. *The C++ programming language*. Pearson Education India, 2000.

20 Peter Thiemann. A typed representation for HTML and XML documents in Haskell. *Journal of functional programming*, 12(4-5):435–468, 2002.

21 N. Wirth. The programming language Pascal. *Acta informatica*, 1:35–63, 1971.

22 Hao Xu. EriLex: an embedded domain specific language generator. In *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, pages 192–212. Springer, 2010.

NumLin: Linear Types for Linear Algebra

Dhruv C. Makwana
Unaffiliated
https://dhruvmakwana.com
dcm41@cam.ac.uk

Neelakantan R. Krishnaswami
Department of Computer Science and Technology, University of Cambridge, United Kingdom
https://www.cl.cam.ac.uk/~nk480/
nk480@cl.cam.ac.uk

Abstract

We present NumLin, a functional programming language whose type system is designed to enforce the safe usage of the APIs of low-level linear algebra libraries (such as BLAS/LAPACK). We do so through a brief description of its key features and several illustrative examples. We show that NumLin's type system is sound and that its implementation improves upon naïve implementations of linear algebra programs, almost towards C-levels of performance. By doing so, we demonstrate (a) that linear types are well-suited to expressing the APIs of low-level linear algebra libraries accurately and concisely and (b) that, despite the complexity of prior work on it, fractional permissions can actually be implemented using simple, well-known techniques and be used practically in real programs.

2012 ACM Subject Classification Theory of computation → Program specifications

Keywords and phrases numerical, linear, algebra, types, permissions, OCaml

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2019.14

Related Version A full version of the paper is available at https://github.com/dc-mak/NumLin/blob/master/write-up/paper.pdf.

Supplement Material ECOOP 2019 Artifact Evaluation approved artifact available at
https://dx.doi.org/10.4230/DARTS.5.2.3
https://github.com/dc-mak/NumLin

Acknowledgements We would like to thank Stephen Dolan for his advice and support with the implementation and evaluation of NumLin. We would also like to thank the (anonymous) reviewers for their feedback and suggestions.

1 Introduction

Programmers writing numerical software often find themselves caught on the horns of a dilemma. The foundational, low-level linear algebra libraries such as BLAS and LAPACK offer programmers very precise control over the memory lifetime and usage of vector and matrix values. However, this power comes paired with the responsibility to manually manage the memory associated with each array object, and in addition to bringing in the familiar difficulties of reasoning about lifetimes, aliasing and sharing that plague low-level systems programming; this also moves the APIs away from the linear-algebraic, mathematical style of thinking that numerical programmers want to use.

As a result, programmers often turn to higher-level languages such as Matlab, R and NumPy, which offer very high-level array abstractions that can be viewed as ordinary mathematical values. This makes programming safer, as well as making prototyping and verification much easier, since it lets programmers write programs which bear a closer resemblance to the formulas that the mathematicians and statisticians designing these algorithms prefer to work with, and ensures that program bugs will reflect incorrectly-computed values rather than heap corruption.

33rd European Conference on Object-Oriented Programming (ECOOP 2019).
Editor: Alastair F. Donaldson; Article No. 14; pp. 14:1–14:25
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

The intention is that these languages can use libraries BLAS and LAPACK, without having to expose programmers to explicit memory management. However, this benefit comes at a price: because user programs do not worry about aliasing, the language implementations cannot in general exploit the underlying features of the low-level libraries that let them explicitly manage and reuse memory. As a result, programs written in high-level statistical languages can be much less memory-efficient than programs that make full use of the powers the low-level APIs offer.

So in practice, programmers face a trade-off: they can eschew safety and exploit the full power of the underlying linear algebra libraries, or they can obtain safety at the price of unneeded copies and worse memory efficiency. In this work, we show that this trade-off is not a fundamental one.

NumLin is a functional programming language whose type system is designed to enforce the safe usage of the APIs of low-level linear algebra libraries (such as BLAS/LAPACK). It does so by combining linear types, fractional permissions, runtime errors and recursion into a small, easily understandable, yet expressive set of core constructs.

NumLin allows a novice to understand and work with complicated linear algebra library APIs, as well as point out subtle aliasing bugs and reduce memory usage in existing programs. In fact, we were able to use NumLin to find linearity and aliasing bugs in a linear algebra algorithm that was *generated* by another program *specifically designed to translate matrix expressions into an efficient sequence of calls to linear algebra routines.* We were also able to reduce the number of temporaries used by the same algorithm, using NumLin's type system to guide us.

NumLin's implementation supports several syntactic conveniences as well as a *usable* integration with real OCaml libraries.

1.1 Contributions

Our contribution is the idea applying of linear types with fractional permissions to enforce the correct *usage* (as opposed to *implementation*) of linear algebra libraries. We explain the idea in detail and provide evidence for its efficacy. Prior type systems for fractional permissions [11, 9, 8] are quite complex. This is because these type systems typically encode a sophisticated analysis to automatically infer how fractional permissions should be split and rejoined.

In contrast, in NumLin, we made sharing and merging explicit. As a result, we were able to drastically simplify the type system. Thefore, our formal system is very close to standard presentations of linear logic, and the implementation complexity is no worse than that for parametric polymorphism.

In this paper

- we describe NumLin, a linearly typed language for linear algebra programs
- we illustrate that NumLin's design and features are well-suited to its intended domain with progressively sophisticated examples
- we prove NumLin's soundness, using a step-indexed logical relation
- we describe a very simple, unification based type-inference algorithm for polymorphic fractional permissions (similar to ones used for parametric polymorphism), demonstrating an alternative approach to dataflow analysis [9]
- we describe an implementation that is compatible with and usable from existing code
- we show an example of how using NumLin helped highlight linearity and aliasing bugs, and reduce the memory usage of a *generated* linear algebra program

- we show that using NumLin, we can achieve parity with C for linear algebra routines, whilst having much better static guarantees about the linearity and aliasing behaviour of our programs.

2 NumLin Overview and Examples

2.1 Type System and Other Features

The core type theory of NumLin is a nearly off-the-shelf linear type theory [3], supporting familiar features such as linear function spaces $A \multimap B$ and tensor products $A \otimes B$. We adopt linearity – the restriction that each program variable be used exactly once – since it allows us to express purely functional APIs for numerical library routines that mutate arrays and matrices [20]. Due to linearity, values cannot alias and are only used once, which means that linearly-typed updates result in no *observable* mutation.

As a result, programmers can reason about NumLin expressions as if they were ordinary mathematical expressions – as indeed they are! We are merely adopting a stricter type discipline than usual to make managing memory safe.

2.1.1 Intuitionism: ! and `Many`

However, linearity by itself is not sufficient to produce an expressive enough programming language. For values such as booleans, integers, floating-point numbers as well as pure functions, we need to be able to use them *intuitionistically*, that is, more than once or not at all. For this reason, we have the ! constructor at the type level and its corresponding `Many` constructor and `let Many <id> = .. in ..` eliminator at the term level. Because we want to restrict how a programmer can alias pointers and prevent a programmer from ignoring them (a memory leak), NumLin enforces simple syntactic restrictions on which values can be wrapped up in a `Many` constructor (details in Section 3).

2.1.2 Fractional Permissions

There are also valid cases in which we would want to alias pointers to a matrix. The most common is exemplified by the BLAS routine `gemm`, which (rather tersely) stands for *GEneric Matrix Multiplication*. A *simplified* definition of `gemm(`α`, A, B, `β`, C)` is $C := \alpha AB + \beta C$. In this case, `A` and `B` may alias each other but neither may alias `C`, because it is being written to. Related to *mutating* arrays and matrices is *freeing* them. Here, we would also wish to restrict aliasing so that we do not free one alias and then attempt to use another. Although linearity on its own suffices to prevent use-after-free errors when values are *not* aliased (a freed value is *out of scope* for the rest of the expression), we still need another simple, yet powerful concept to provide us with the extra expressivity of aliasing *without* losing any of the benefits of linearity.

Fractional permissions provide exactly this. Concretely, types of (pointers to) arrays and matrices are *parameterised* by a *fraction*. A fraction is either 1 (2^0) or exactly *half* of another fraction (2^{-k}, for natural k). The former represents complete ownership of that value: the programmer may mutate or free that value as they choose; the latter represents read-only access or a *borrow*: the programmer may read from the value but not write to or free it. Creating an array/matrix gives you ownership of it, so too does having one (with a fractional permission of 2^0) passed in as an argument.

In NumLin, we can produce two aliases of a single array/matrix, by *sharing* it. If the original alias had a fractional permission of 2^{-k} then the two new aliases of it will have a fractional permission of $2^{-(k+1)}$ each. Thanks to linearity, the original array/matrix with a fractional permission of 2^{-k} will be out of scope after the sharing. When an array/matrix is shared as such, we can prevent the programmer from freeing or mutating it by making the types of `free` and `set` (for mutation) require a *whole* (2^0) permission.

If we have two aliases *to the same matrix* with *identical* fractional permissions ($2^{-(k+1)}$), we can recombine or *unshare* them back into a single one, with a larger 2^{-k} permission. As before, thanks to linearity, the original two aliases will be out of scope after unsharing.

2.1.3 Runtime Errors

Aside from out-of-bounds indexing, matrix unsharing is one of only *two* operations that can fail at runtime (the other being dimension checks, such as for `gemm`). The check being performed is a simple sanity check that the two aliasing pointers passed to `unshare` point to the same array/matrix. Section 5 contains an overview of how we could remove the need for this by tracking pointer identities statically by augmenting the type system further.

2.1.4 Recursion

The final feature of NumLin which makes it sufficiently expressive is recursion (and of course, conditional branches to ensure termination). Conditional branches are implemented by ensuring that both branches use the same set of linear values. A function can be recursive if it captures no linear values from its environment. Like with `Many`, this is enforced via simple syntactic restrictions on the definition of recursive functions.

2.2 Syntax

NumLin's concrete syntax is inspired by that of OCaml. It desugars (Figure 2) into a smaller, core type and expression grammar (Figure 1).

As described in Section 2.1.2, fractional permissions f are either variables $'fc$, `z` (2^0) or f `s` ($2^{-(f+1)}$).

Types are either simple (**unit**, **bool**, **int**, **elt**), indexed by fractional permission (f **arr**, f **mat**) or compound (! constructor for intuitionism, universally-quantifying over a fractional-permission $'fc$ in t for fraction-polymorphic types, pairs, linear functions).

Expressions are standard with the exception of $'fc.\ t$ introduction (fractional permission abstraction) and elimination (fractional permission specialisation) forms.

The ! annotation on variables is a syntactic convenience for declaring the variable to used intuitionistically; why it desugars the way it does is explained in Section 3.1.

Array/matrix indexing and duplication (non-destructive and destructive) also have special syntax to lessen the syntactic overhead of re-binding identifiers. Furthermore, to aid readability, there is support for using BLAS methods via conventional-looking matrix expressions.

In particular, the syntax `let y <- new (m,n) [| alpha * x1 * x2 |]` is syntactic sugar for first creating a new $m \times n$ matrix (`let y = matrix m n`) and then storing the result of the multiplication in it (`let ((x1, x2), y) = .. in ..`).

Note that, the pattern `let y <- [| x^T * x + beta * y |]` translates to (`syrk true 1. x beta y`), which uses `x` once only.

$$
\begin{array}{lcl}
f & ::= & {'\!f\!c} \mid \mathtt{z} \mid f\ \mathtt{s} \\[1em]
t & ::= & \mathbf{unit} \mid \mathbf{bool} \mid \mathbf{int} \mid \mathbf{elt} \mid f\ \mathbf{arr} \mid f\ \mathbf{mat} \mid\ !t \mid {'\!f\!c}.\ t \mid t \otimes t' \mid t \multimap t' \\[1em]
e & ::= & p \text{ (primitives)} \mid x \text{ (variable)} \mid \mathbf{let}\ x = e\ \mathbf{in}\ e' \mid () \mid \mathbf{let}\ () = e\ \mathbf{in}\ e' \mid \mathbf{true} \mid \mathbf{false} \\
 & & \mathbf{if}\ e\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 \mid k \text{ (integer)} \mid l \text{ (heap location)} \mid el \text{ (array element)} \\
 & & \mathbf{Many}\ v \mid \mathbf{let}\ \mathbf{Many}\ x = e\ \mathbf{in}\ e' \mid \mathbf{fun}\ {'\!f\!c} \to e \text{ (frac. perm. abstraction)} \\
 & & e\ [f] \text{ (frac. perm. specialisation)} \mid (e, e') \mid \mathbf{let}\ (a, b) = e\ \mathbf{in}\ e' \\
 & & \mathbf{fun}\ x : t \to e \mid e\ e' \mid \mathbf{fix}\,(g, x : t, e : t')
\end{array}
$$

Figure 1 Core fraction f, type t and expression e grammar of NumLin. Values v are a subset of the expressions, their full definition and a list of all primitives p is in the Appendix.

$$
\begin{array}{rcll}
x[e] & \Rightarrow & \mathbf{get}\ \_\ x\ (e) & \text{(similarly for matrices)} \\
x[e_1] := e_2 & \Rightarrow & \mathbf{set}\ x\ (e_1)\ (e_2) & \text{(similarly for matrices)} \\[1em]
pat & ::= & () \mid x \mid\ !x \mid \mathbf{Many}\ pat \mid (pat, pat) & \\
\mathbf{let}\ !x = e_1\ \mathbf{in}\ e_2 & \Rightarrow & \mathbf{let}\ \mathbf{Many}\ x = e_1\ \mathbf{in} & \\
 & & \mathbf{let}\ \mathbf{Many}\ x = \mathbf{Many}\ (\mathbf{Many}\ x)\ \mathbf{in}\ e_2 & \\
\mathbf{let}\ \mathbf{Many}\ \langle pat_x \rangle\ = e_1\ \mathbf{in}\ e_2 & \Rightarrow & \mathbf{let}\ \mathbf{Many}\ x = x\ \mathbf{in} & \\
 & & \mathbf{let}\ \langle pat_x \rangle\ = x\ \mathbf{in}\ e_2 & \\
\mathbf{let}\ (\langle pat_a \rangle, \langle pat_b \rangle)\ = e_1\ \mathbf{in}\ e_2 & \Rightarrow & \mathbf{let}\ (a, b)\ = a\_b\ \mathbf{in}\ \mathbf{let}\ \langle pat_a \rangle\ = a\ \mathbf{in} & \\
 & & \mathbf{let}\ \langle pat_b \rangle\ = b\ \mathbf{in}\ e_2 & \\
\mathbf{fun}\ (\langle pat_x \rangle : t) \to e & \Rightarrow & \mathbf{fun}\ (x : t) \to \mathbf{let}\ \langle pat_x \rangle = x\ \mathbf{in}\ e & \\[1em]
arg & ::= & (\langle pat \rangle : t)\ \mid\ ({'x})\ \text{(fractional permission variable)} & \\
\mathbf{fun}\ \langle arg_{1..n} \rangle \to e & \Rightarrow & \mathbf{fun}\ \langle arg_1 \rangle \to ..\ \mathbf{fun}\ \langle arg_n \rangle \to e & \\
\mathbf{let}\ f\ \langle arg_{1..n} \rangle = e_1\ \mathbf{in}\ e_2 & \Rightarrow & \mathbf{let}\ f = \mathbf{fun}\ \langle arg_{1..n} \rangle \to e_1\ \mathbf{in}\ e_2 & \\
\mathbf{let}\ !f\ \langle arg_{1..n} \rangle = e_1\ \mathbf{in}\ e_2 & \Rightarrow & \mathbf{let}\ \mathbf{Many}\ f = \mathbf{Many}\ (\mathbf{fun}\ \langle arg_{1..n} \rangle \to e_1)\ \mathbf{in}\ e_2 & \\
\text{fixpoint} & \equiv & \mathbf{fix}\,(f, x : t, \mathbf{fun}\ \langle arg_{0..n} \rangle \to e_1 : t') & \\
\mathbf{let\ rec}\ f\ (x : t)\ \langle arg_{0..n} \rangle : t' = e_1\ \mathbf{in}\ e_2 & \Rightarrow & \mathbf{let}\ f = \text{fixpoint}\ \mathbf{in}\ e_2 & \\
\mathbf{let\ rec}\ !f\ (x : t)\ \langle arg_{0..n} \rangle : t' = e_1\ \mathbf{in}\ e_2 & \Rightarrow & \mathbf{let}\ \mathbf{Many}\ f = \mathbf{Many}\ \text{fixpoint}\ \mathbf{in}\ e_2 &
\end{array}
$$

Figure 2 Desugaring NumLin.

$$\begin{aligned}
\textbf{let } v \leftarrow x[e] \textbf{ in } e' \quad &\Rightarrow \quad \textbf{let } (x, !v) \ = x[e] \textbf{ in } e' \qquad \text{(similarly for matrices)} \\
\textbf{let } x_2 \leftarrow \textbf{new } [|\ x_1\ |] \textbf{ in } e \quad &\Rightarrow \quad \textbf{let } (x_1, x_2) \ = \textbf{copyM } \_\ x_1 \textbf{ in } e \\
\textbf{let } x_2 \leftarrow [|\ x_1\ |] \textbf{ in } e \quad &\Rightarrow \quad \textbf{let } (x_1, x_2) \ = \textbf{copyM\_to } \_\ x_1\ x_2 \textbf{ in } e
\end{aligned}$$

$$M ::= X \ \mid \ X^T \ \mid \ \mathrm{sym}(X)$$

$$\begin{aligned}
&\textbf{let } Y \leftarrow \textbf{new } (n, k)\ [|\ \alpha M_1 M_2\ |] \textbf{ in } e \Rightarrow \\
&\quad \textbf{let } Y \ = \textbf{matrix } n\ k \textbf{ in let } Y \leftarrow [|\ \alpha M_1 M_2 + 0Y\ |] \textbf{ in } e \\
&\textbf{let } Y \leftarrow [|\ \alpha X X^T + \beta Y\ |] \textbf{ in } e \Rightarrow \\
&\quad \textbf{let } (X, Y) \ = \textbf{syrk false } \alpha\ \_\ X\ \beta\ Y \textbf{ in } e \\
&\textbf{let } Y \leftarrow [|\ \alpha X^T X + \beta Y\ |] \textbf{ in } e \Rightarrow \\
&\quad \textbf{let } (X, Y) \ = \textbf{syrk true } \alpha\ \_\ X\ \beta\ Y \textbf{ in } e \\
&\textbf{let } Y \leftarrow \ [|\ \alpha\, \mathrm{sym}(X_1)\, X_2 + \beta Y\ |] \textbf{ in } e \Rightarrow \\
&\quad \textbf{let } ((X_1, X_2), Y) \ = \textbf{symm false } \alpha\ \_\ X_1\_\ X_2\ \beta\ Y \textbf{ in } e \\
&\textbf{let } Y \leftarrow \ [|\ \alpha\, X_2\, \mathrm{sym}(X_1) + \beta Y\ |] \textbf{ in } e \Rightarrow \\
&\quad \textbf{let } ((X_1, X_2), Y) \ = \textbf{symm true } \alpha\ \_\ X_1\_\ X_2\ \beta\ Y \textbf{ in } e \\
&\textbf{let } Y \leftarrow [|\ \alpha X_1^{T?} X_2^{T?} + \beta Y\ |] \textbf{ in } e \Rightarrow \\
&\quad \textbf{let } ((X_1, X_2), Y) \ = \ \textbf{gemm } \alpha\ \_\ (X_1, \substack{\textbf{true} \\ \textbf{false}})\ \_\ (X_2, \substack{\textbf{true} \\ \textbf{false}})\ \beta\ Y \textbf{ in } e
\end{aligned}$$

Figure 3 Purely syntactic pattern-matching translations of matrix expressions.

2.3 Examples

2.3.1 Factorial

Although a factorial function (Figure 4) may seem like an aggressively pedestrian first example, in a linearly typed language such as NumLin it represents the culmination of many features.

To simplify the design and implementation of NumLin's type system, recursive functions must have full type annotations (non-recursive functions need only their argument types annotated). The body of the factorial function is a closed expression (with respect to the function's arguments), so it type-checks (since it does not capture any linear values from its environment).

The only argument is `!x : !int`. As explained before (Section 2.2), this declares `x` to be used intuitionistically.

The condition for an `if` may or may not use linear values (here, with `x < 0 || x = 0`, it does not). Any linear values used by the condition would not be in scope in either branch of the `if`-expression. Both branches use `x` differently: one ignores it completely and the other uses it twice.

All numeric and boolean literals are implicitly wrapped in a `Many` and all primitives involving them return a `!int`, `!bool` or `!elt` (types of elements of arrays/matrices, typically 64-bit floating-point numbers). The short-circuiting `||` behaves in exactly the same way as a boolean-valued `if`-expression.

```
let rec factorial ( !x : !int ) : !int =
    if x < 0 || x = 0 then
        1
    else
        x * factorial (x - 1) in
factorial ;;
```

Figure 4 Factorial function in NumLin.

```
let rec sum_array (!i : !int) (!n : !int) (!x0 : !elt)
                  ('x) (row : 'x arr) : 'x arr * !elt =
    if i = n then
        (row, x0)
    else
        let (row, !x1) = row[i] in
        sum_array (i + 1) n (x0 +. x1) 'x row in
sum_array ;;
```

Figure 5 Summing over an array in NumLin.

2.3.2 Summing over an Array

Now we can add fractional permissions to the mix: Figure 5 shows a simple, tail-recursive implementation of summing all the elements in an array. There are many new features; first among them is !x0 : !elt, the type of array/matrix elements (64-bit floating point).

Second is ('x) (row: 'x arr) which is an array with a universally-quantified fractional permission. In particular, this means the body of the function cannot mutate or free the input array, only read from it. If the programmer did try to mutate or free row, then they would get a helpful error message (Figure 6).

Alongside taking a row: 'x arr, the function also returns an array with exactly the same fractional permission as the row (which can only be row). This is necessary because of linearity: for the caller, the original array passed in as an argument would be out of scope for the rest of the expression, so it needs to be returned and then rebound to be used for the rest of the function.

An example of this consuming and re-binding is in let (row, !x1) = row[i]. Indexing is implemented as a primitive $\mathbf{get} : \ 'x.\ 'x\ \mathbf{arr} \multimap \mathbf{!int} \multimap\ 'x\ \mathbf{arr} \otimes \mathbf{!elt}$. Although fractional permissions can be passed around explicitly (as done in the recursive call), they can also be *automatically inferred at call sites*: row[i] ⇒ get _ row i takes advantage of this convenience.

2.3.3 One-dimensional Convolution

Figure 7 extends the set of features demonstrated by the previous examples by mutating one of the input arrays. A one-dimensional convolution involves two arrays: a read-only kernel (array of weights) and an input vector. It modifies the input vector *in-place* by replacing each write[i] with a weighted (as per the values in the kernel) sum of it and its neighbours; intuitively, sliding a dot-product with the kernel across the vector.

```
let row = row[i] := x1 in (* or *) let () = free row in
(* Could not show equality:                                     *)
(*     z arr                                                    *)
(* with                                                         *)
(*     'x arr                                                   *)
(*                                                              *)
(* Var 'x is universally quantified                             *)
(* Are you trying to write to/free/unshare an array you don't own? *)
(* In examples/sum_array.lt, at line: 7 and column: 19          *)
```

Figure 6 Attempting to write to or free a read only array in NumLin.

```
let rec simp_oned_conv
    (!i : !int) (!n : !int) (!x0 : !elt)
    (write : z arr) ('x) (weights : 'x arr)
    : 'x arr * z arr =
  if n = i then (weights, write) else
  let !w0 <- weights[0]   in
  let !w1 <- weights[1]   in
  let !w2 <- weights[2]   in
  let !x1 <- write[i]     in
  let !x2 <- write[i + 1] in
  let written = write[i] := w0 *. x0 +. (w1 *. x1 +. w2 *. x2) in
  simp_oned_conv (i + 1) n x1 written _ weights in
 simp_oned_conv ;;
```

Figure 7 *Simplified* one-dimensional convolution.

What's implemented in Figure 7 is a *simplified* version of this idea, so as to not distract from the features of NumLin. The simplifications are:

- the kernel has a length 3, so only the value of `write[i-1]` (prior to modification in the previous iteration) needs to be carried forward using `x0`
- `write` is assumed to have length `n+1`
- `i`'s initial value is assumed to be `1`
- `x0`'s initial value is assumed to be `write[0]`
- the first and last values of `write` are ignored.

Mutating an array is implemented similarly to indexing one – a primitive **set** : **z arr** $\multimap$ **!int** $\multimap$ **!elt** $\multimap$ **z arr**. It consumes the original array and returns a new array with the updated value.

Since `write`: `z arr` (where `z` stands for $k = 0$, representing a fractional permission of $2^{-k} = 2^{-0} = 1$), we may mutate it, but since we only need to read from `weights`, its fractional permission index can be universally-quantified. In the recursive call, we see `_` being used explicitly to tell the compiler to *infer* the correct fractional permission based on the given arguments.

2.3.4 Digression: Types of Primitives

The most pertinent aspect of NumLin *is the types of its primitives* (Figure 8). While the types of operations such as **get** and **set** might be borderline obvious, the types of BLAS/LAPACK routines become an *incredibly useful, automated check for using the API correctly.*

We determine the types for these routines by consulting their documentation. Each routine has a record of the expected aliasing behaviour and whether or not it modifies or

$$\begin{array}{rcl}
\textbf{symm} & : & \textbf{!bool} \multimap \textbf{!elt} \multimap {'x}.\ {'x}\ \textbf{mat} \multimap {'y}.\ {'y}\ \textbf{mat} \multimap \textbf{!elt} \multimap \textbf{z mat} \multimap \\
 & & ({'x}\ \textbf{mat} \otimes {'y}\ \textbf{mat}) \otimes \textbf{z mat} \\
\textbf{gemm} & : & \textbf{!elt} \multimap {'x}.\ {'x}\ \textbf{mat} \otimes \textbf{!bool} \multimap {'y}.\ {'y}\ \textbf{mat} \otimes \textbf{!bool} \multimap \textbf{!elt} \multimap \\
 & & \textbf{z mat} \multimap ({'x}\ \textbf{mat} \otimes {'y}\ \textbf{mat}) \otimes \textbf{z mat} \\
\textbf{gesv} & : & \textbf{z mat} \multimap \textbf{z mat} \multimap \textbf{z mat} \otimes \textbf{z mat} \\
\textbf{posv} & : & \textbf{z mat} \multimap \textbf{z mat} \multimap \textbf{z mat} \otimes \textbf{z mat} \\
\textbf{potrs} & : & {'x}.\ {'x}\ \textbf{mat} \multimap \textbf{z mat} \multimap {'x}\ \textbf{mat} \otimes \textbf{z mat} \\
\textbf{syrk} & : & \textbf{!bool} \multimap \textbf{!elt} \multimap {'x}.\ {'x}\ \textbf{mat} \multimap \textbf{!elt} \multimap \textbf{z mat} \multimap {'x}\ \textbf{mat} \otimes \textbf{z mat}
\end{array}$$

Figure 8 Types of some NumLin primitives.

```
let !square ('x) (x : 'x mat) =
  let (x, (!m, !n)) = sizeM _ x in
  let (x1, x2) = shareM _ x in
  let answer <- new (m, n) [| x1 * x2 |] in
  let x = unshareM _ x1 x2 in
  (x, answer) in
square ;;
```

Figure 9 Squaring a matrix.

consumes its argument in any way. We use that to derive the types in Figure 8. Since most of these low-level routines are very careful not to do any allocation themselves, it is generally very easy to give each a NumLin type – every argument that can modify/consume its argument needs a full permission, and all others can be fraction-polymorphic. Taking Fortran as an example, it has a notion of `in`, `out` and `inout` parameters. The latter two would need full `z` permissions; the first would be fraction-polymorphic.

2.3.5 Squaring a Matrix

Figure 9 shows how a linearly-typed matrix squaring function may be written in NumLin. It is a *non-recursive* function declaration (the return type is inferred). Since we would like to be able to use a function like `square` more than once, it is marked with a `!` annotation (which also ensures it captures no linear values from the surrounding environment).

To square a matrix, first, we extract the dimensions of the argument `x`. Then, because we need to use `x` twice (so that we can multiply it by itself) but linearity only allows one use, we use **shareM** : ${'x}.\ {'x}\ \textbf{mat} \multimap {'x}\ \textbf{s mat} \otimes {'x}\ \textbf{s mat}$ to split the permission `'x` (which represents 2^{-x}) into two halves (`'x s`, which represents $2^{-(x+1)}$).

Even if `x` had type `z mat`, sharing it now enforces the assumption of all BLAS/LAPACK routines that any matrix which is written to (which, in NumLin, is always of type `z mat`) does not alias any other matrix in scope. So if we did try to use one of the aliases in mutating way, the expression would not type check, and we would get an error similar to the one in Figure 6.

By using some simple pattern-matching and syntactic sugar (Figure 3), we can:

- write normal-looking, apparently non-linear code
- use matrix expressions directly and have a call to an efficient call to a BLAS/LAPACK routine inserted with appropriate re-bindings

```
let !lin_reg ('x) (x : 'x mat)
             ('y) (y : 'y mat) =
  let (x, (!_n, !m)) = sizeM _ x in
  let xy <- new (m, 1) [| x^T * y |] in
  let x_T_x <- new (m, m) [| x^T * x |] in
  let (to_del, answer) = posv x_T_x xy in
  let () = freeM to_del in
  ((x, y), answer) in
lin_reg ;;
```

Figure 10 Linear regression (OLS): $\hat{\beta} = (\mathbf{X^TX})^{-1}\mathbf{X^Ty}$.

- retain the safety of linear types with fractional permissions by having the compiler statically enforce the aliasing and read/write rules implicitly assumed by BLAS/LAPACK routines.

2.3.6 Linear Regression

In Figure 10, we wish to compute $\hat{\beta} = (\mathbf{X^TX})^{-1}\mathbf{X^Ty}$. To do that, first, we extract the dimensions of matrix x. Then, we say we would like xy to be a new matrix, of dimension $m \times 1$, which contains the result of $\mathbf{X^Ty}$ (using syntactic sugar for matrix and gemm calls similar to that used in Figure 9, with a ^T annotation on x to set x's "transpose indices"-flag to true).

Note that x can appear *twice* in the *pattern* without any calls to share because the pattern is matched to a BLAS call to syrk true 1. x 0. x_T_x, which uses x once only.

After computing x_T_x, we need to invert it and then multiply it by xy. The BLAS routine **posv** : z **mat** $\multimap$ z **mat** $\multimap$ z **mat** $\otimes$ z **mat** does exactly that: assuming the first argument is symmetric, posv mutates its second argument to contain the desired value. Its first argument is also mutated to contain the (upper triangular) Cholesky decomposition factor of the original matrix. Since we do not need that matrix (or its memory) again, we free it. If we forgot to, we would get a Variable to_del not used error. Lastly, we return the answer alongside the untouched input matrices (x,y).

2.3.7 L1-norm Minimisation on Manifolds

L1-norm minimisation is often used in optimisation problems, as a *regularisation* term for reducing the influence of outliers. Although the below formulation [12] is intended to be used with *sparse* computations, NumLin's current implementation only implements dense ones. However, it still serves as a useful example of explaining NumLin's features.

Primitives like **transpose** : $'x.\ 'x$ **mat** $\multimap$ $'x$ **mat** $\otimes$ z **mat** and **eye** : !**int** $\multimap$ z **mat** allocate new matrices; transpose returns the transpose of a given matrix and eye k evaluates to a $k \times k$ identity matrix.

We also see our first example of re-using memory for different matrices: like with to_del and posv in the previous example, we do not need the value stored in tmp_n_n after the call to gesv (a primitive similar to posv but for a non-symmetric first argument). However, we can re-use its memory much later to store answer with let answer <- [| 0. * tmp_n_n + q_inv_u * inv_u_T |]. Again, thanks to linearity, the identifiers q and tmp_n_n are out of scope by the time answer is bound. Although during execution, all three refer to the same piece of memory, logically they represent different values throughout the computation.

```
let !l1_norm_min (q : z mat) (u : z mat) =
  let (u, (!_n, !k)) = sizeM _ u in
  let (u, u_T) = transpose _ u in
  let (tmp_n_n , q_inv_u ) = gesv q u in
  let i = eye k in
  let to_inv <- [| i + u_T * q_inv_u |] in
  let (tmp_k_k, inv_u_T ) = gesv to_inv u_T in
  let () = freeM tmp_k_k in
  let answer <- [| 0. * tmp_n_n + q_inv_u * inv_u_T |] in
  let () = freeM q_inv_u in
  let () = freeM inv_u_T in
  answer in
l1_norm_min ;;
```

Figure 11 L1-norm minimisation on manifolds: $\mathbf{Q}^{-1}\mathbf{U}(\mathbf{I}+\mathbf{U}^{\mathbf{T}}\mathbf{Q}^{-1}\mathbf{U})^{-1}\mathbf{U}^{\mathbf{T}}$.

2.3.8 Kalman Filter

A *Kalman Filter* [16] is an algorithm for combining prior knowledge of a state, a statistical model and measurements from (noisy) sensors to produce an estimate a more reliable estimated of the current state. It has various applications (navigation, signal-processing, econometrics) and is relevant here because it is usually presented as a series of complex matrix equations.

Figure 12 shows a NumLin implementation of a Kalman filter (equations in Figure 13). A few new features and techniques are used in this implementation:

- sym annotations in matrix expressions: when this is used, a call to symm (the equivalent of gemm but for symmetric matrices so that only half the operations are performed) is inserted
- copyM_to is used to re-use memory by *overwriting* the contents of its second argument to that of its first (erroring if dimensions do not match)
- posvFlip is like posv except for solving $XA = B$
- a lot of memory re-use; the following sets of identifiers alias each other:
  - r_1, r_2 and k_by_k
  - data_1 and data_2
  - mu and new_mu
  - sigma_hT and x.

The NumLin implementation is much longer than the mathematical equations for two reasons. First, the NumLin implementation is a let-normalised form of the Kalman equations: since there a large number of unary/binary (and occasionally ternary) sub-expressions in the equations, naming each one line at a time makes the implementation much longer. Second, NumLin has the additional task of handling explicit allocations, aliasing and frees of matrices. However, it is exactly this which makes it possible (and often, easy) to spot additional opportunities for memory re-use. Furthermore, a programmer can explore those opportunities easily because NumLin's type system statically enforces correct memory management and the aliasing assumptions of BLAS/LAPACK routines.

3 Formal System

3.1 Core Type Theory

The full typing rules are in the Appendix, but the key ideas are as follow.

- A typing judgement consists of $\Theta; \Delta; \Gamma \vdash e : t$.

```
let !kalman
  ('s) (sigma : 's mat) (* n,n *)
  ('h) (h : 'h mat)     (* k,n *)
  (mu : z mat)          (* n,1 *)
  (r_1 : z mat)         (* k,k *)
  (data_1 : z mat)      (* k,1 *) =
  let (h, (!k, !n)) = sizeM _ h in
  (* could use [| sym(sigma) * hT |] but would
     need a (n,k) temporary hT = tranpose _ h *)
  let sigma_hT <- new (n, k) [| sigma * h^T |] in
  let r_2 <- [| r_1 + h * sigma_hT |] in
  let (k_by_k, x) = posvFlip r_2 sigma_hT in
  let data_2 <- [| h * mu - data_1 |] in
  let new_mu <- [| mu + x * data_2 |] in
  let x_h <- new (n,n) [| x * h |] in
  let () = freeM (* n,k *) x in
  let sigma2 <- new [| sigma |] in
  let new_sigma <- [| sigma2 - x_h * sym(sigma) |] in
  let () = freeM (* n,n *) x_h in
  ((sigma, h), (new_sigma, (new_mu, (k_by_k, data_2)))) in
kalman ;;
```

Figure 12 Kalman filter: see Figure 13 for the equations this code implements and the Appendix for an equivalent CBLAS/LAPACKE implementation.

$$\begin{aligned}\mu' &= \mu + \Sigma H^T (R + H \Sigma H^T)^{-1} (H\mu - \text{data})\\ \Sigma' &= \Sigma (I - H^T (R + H\Sigma H^T)^{-1} H \Sigma)\end{aligned}$$

Figure 13 Kalman filter equations (credit: matthewrocklin.com).

- Θ is the environment that tracks which fractional permission variables in scope. Fractional permissions (the Perm judgement) and types (the Type judgement) are *well-formed* if all of their free fractional variables are in Θ.
- Δ is the environment storing non-linearly or *inuitionistically* typed variables.
- Γ is the environment storing linearly typed variables.

Note that rules for typing (), booleans, integers and elements are typed with respect to an *empty* linear environment: this means no linear values are needed to produce a value of those types.

$$\frac{}{\Theta; \Delta; \cdot \vdash () : \mathbf{unit}} \quad \textsc{Ty\_Unit\_Intro}$$

Conversely, whenever two or more subexpressions need to be typed, they must consume a disjoint set of linear values (pairs, let-expressions). In the case of if-expressions, both branches must consume the same set of linear values (disjoint to the ones used to evaluate the condition).

$$\frac{\begin{array}{l}\Theta; \Delta; \Gamma \vdash e : !\mathbf{bool}\\ \Theta; \Delta; \Gamma' \vdash e_1 : t'\\ \Theta; \Delta; \Gamma' \vdash e_2 : t'\end{array}}{\Theta; \Delta; \Gamma, \Gamma' \vdash \mathbf{if}\ e\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 : t} \quad \textsc{Ty\_Bool\_Elim}$$

The `Many` introduction and elimination rules are very important. Producing !-type values may only be done if the expression inside is a syntactic value which is not a location. This allows all safely duplicable resources, including functions which capture non-linear resources from their environments, but prevents producing aliases of (pointers to) arrays and matrices. This is exactly the same as value-restriction from the world of parametric polymorphism; without it, the expression $\textbf{let Many}\ x = \textbf{Many}\ (\textbf{array}\ 5)\ \textbf{in let}\ () = \textbf{free}\ x\ \textbf{in}\ x[0]$ would type-check but error at runtime.

$$\frac{\begin{array}{l}\Theta; \Delta; \cdot \vdash v : t \\ v \neq l\end{array}}{\Theta; \Delta; \cdot \vdash \textbf{Many}\ v : !t}\ \textsc{Ty\_Bang\_Intro}$$

Consuming a variable that refers to a !-type value *moves it* from the linear environment Γ and *into* the intuitionistic environment Δ.

$$\frac{\begin{array}{l}\Theta; \Delta; \Gamma \vdash e : !t \\ \Theta; \Delta, x : t; \Gamma' \vdash e' : t'\end{array}}{\Theta; \Delta; \Gamma, \Gamma' \vdash \textbf{let Many}\ x = e\ \textbf{in}\ e' : t'}\ \textsc{Ty\_Bang\_Elim}$$

Using this, we can explain how the ! annotation on variables – first introduced in the factorial example in 2.3.1 – works. That is, we can explain why the meaning of $\textbf{let}\ !x = e\ \textbf{in}\ e'$ can be expressed using only the rules presented thus far, as $\textbf{let Many}\ x = e\ \textbf{in let Many}\ x = \textbf{Many}\ (\textbf{Many}\ x)\ \textbf{in}\ e'$.[1] The reader is invited to quickly convince themselves that the following meta-rule is provable using Ty_Bang_Intro (twice), Ty_Bang_Elim (twice) and weakening the intuitionistic environment Δ (once).

$$\frac{\begin{array}{l}\Theta; \Delta; \Gamma \vdash e : !t \\ \Theta; \Delta, x : !t; \Gamma' \vdash e' : t'\end{array}}{\Theta; \Delta; \Gamma, \Gamma' \vdash \textbf{let}\ !x = e\ \textbf{in}\ e' : t'}\ \textsc{Meta\_Ty\_Let\_Bang}$$

Rules Ty_Gen and Ty_Spc are for fractional permission generalisation and specialisation respectively. They allow the definition and use of functions that are polymorphic in the fractional permission index of their results and one or more of their arguments.

$$\frac{\Theta, fc; \Delta; \Gamma \vdash e : t}{\Theta; \Delta; \Gamma \vdash \textbf{fun}\ 'fc \to e :\ 'fc.t}\ \textsc{Ty\_Gen} \qquad \frac{\begin{array}{l}\Theta \vdash f\ \mathsf{Perm} \\ \Theta; \Delta; \Gamma \vdash e :\ 'fc.t\end{array}}{\Theta; \Delta; \Gamma \vdash e[f] : t[f/fc]}\ \textsc{Ty\_Spc}$$

Rule Ty_Fix shows how recursive functions are typed. Even though recursive functions are fully annotated, type checking them is interesting for two reasons: to type check the body of the fixpoint, the type of the recusive function is in the *intuitionistic* environment Δ (without this, you would not be able to write a base case) whilst the argument and its type are the *only things in the linear environment* Γ. The latter means that recursive functions can be type checked in an empty environment (thus be wrapped in `Many` and used zero or multiple times).

$$\frac{\Theta; \Delta, g : t \multimap t'; \cdot, x : t \vdash e : t'}{\Theta; \Delta; \cdot \vdash \textbf{fix}\ (g, x : t, e : t') : t \multimap t'}\ \textsc{Ty\_Fix}$$

Lastly, types of almost all NumLin primitives, as embedded in OCaml's type system, are shown in the Appendix, with some similar ones (like those for binary arithmetic operators)

[1] *Why* we have this at all is for the sake of ergonomics when using binary arithmetic operations (e.g. of type $!\textbf{int} \multimap !\textbf{int} \multimap !\textbf{int}$): a programmer should be able to write `let x = 5 + 5 in x - x`, which, although non-linear in `x`, is morally right because integers and operations on them rarely need to be linear. Though it should be possible to handle this using a LNL-style presentation of linear types [5] (using adjoint modalities to distinguish between intrinsically linear and intrinsically intuitionistic types) that is a pretty big digression from the stated goals of this paper.

omitted for brevity. The main difference between the OCaml type of a primitive like `gemm` and its NumLin counterpart (Figure 8) is the inclusion of explicit universal-quantification of fractional permission variables in the latter.

3.2 Dynamic Semantics

The full, small-step transition relation is in the Appendix, but the key ideas are as follow.

Heaps σ are multisets containing triples of an abstract location l, a fractional permission f and sized matrices $m_{n,k}$. The notation $l \mapsto_f m_{k_1,k_2}$ should be read as "location l represents f ownership over matrix m (of size $k_1 \times k_2$)". Each heap-and-expression either steps to another heap-and-expression or a runtime error **err**. In the full grammar definition we see a definition of values and contexts in the language.

We draw the reader's attention to the definitions relating to fractional permissions. Specifically, unlike a lambda, the body of a **fun** $'fc \to v$ must be a syntactic value. The context **fun** $'fc \to [-]$ means expressions can be reduced inside a fractional permission generalisation. This is to emphasize that fractions are merely *compile-time constructs* and do not affect runtime behaviour. Correct usage of fractions is enforced by the type system, so programs do not get stuck. Fractional permissions are specialised using substitution over both the heap and an expression (Op_Frac_Perm).

$$\frac{}{\langle \sigma, (\textbf{fun}\, 'fc \to v)[f]\rangle \to \langle \sigma[f/fc], v[f/fc]\rangle}\ \textsc{Op\_Frac\_Perm}$$

Like with the static semantics, the interesting rules in the dynamic semantics are those relating to primitives. Creating a matrix (**matrix** k_1 k_2) successfully (Op_Matrix) requires non-negative dimensions and returns a (fresh) location of a matrix of those dimensions, extending the heap to reflect that l represents a complete ownership over the new matrix.

$$\frac{\begin{array}{l} 0 \le k_1, k_2 \\ l \text{ fresh} \end{array}}{\langle \sigma, \textbf{matrix}\, k_1\ k_2\rangle \to \langle \sigma + \{l \mapsto_1 M_{k_1,k_2}\}, l\rangle}\ \textsc{Op\_Matrix}$$

Dually, Op_Free, requires a location represent complete ownership before removing it and the matrix it points to from the heap.

$$\frac{}{\langle \sigma + \{l \mapsto_1 m_{k_1,k_2}\}, \textbf{free}\, l\rangle \to \langle \sigma, ()\rangle}\ \textsc{Op\_Free}$$

Choosing a multiset representation as opposed to a set allows for two convenient invariants: multiplicity of a triple $l \mapsto_f m_{k_1,k_2}$ in the heap corresponds to the number of aliases of l in the expression with type f **mat** and the sum of all the fractions for l will always be 1 (for a closed, well-typed expression). With this in mind, the rules Op_Share and Op_Unshare_Eq are fairly natural.

$$\frac{}{\langle \sigma + \{l \mapsto_f m_{k_1,k_2}\}, \textbf{share}[f]\, l\rangle \to \langle \sigma + \{l \mapsto_{\frac{1}{2}f} m_{k_1,k_2}\} + \{l \mapsto_{\frac{1}{2}f} m_{k_1,k_2}\}, (l, l)\rangle}\ \textsc{Op\_Share}$$

$$\frac{\sigma' \equiv \sigma + \{l \mapsto_{\frac{1}{2}f} m_{k_1,k_2}\} + \{l \mapsto_{\frac{1}{2}f} m_{k_1,k_2}\}}{\langle \sigma', \textbf{unshare}[f]\, l\, l\rangle \to \langle \sigma + \{l \mapsto_f m_{k_1,k_2}\}, l\rangle}\ \textsc{Op\_Unshare\_Eq}$$

Combining all of these features, we see that Op_Gemm_Match requires that the location being updated (l_3) has complete ownership of over matrix m_3 and can thus change what value it stores to $m_1 m_2 + m_3$. In particular, this places no restriction on l_2 and l_3: they could be **share**d aliases of the same matrix. Transition rules for other primitives (omitted) follow the same structure: $\mapsto_1$ for any locations that are written to and $\mapsto_{fc}$ for anything else.

$$\frac{\begin{array}{l}\sigma' \equiv \sigma + \{l_1 \mapsto_{fc_1} m1_{k_1,k_2}\} + \{l_2 \mapsto_{fc_2} m2_{k_2,k_3}\}\\ \sigma_1 \equiv \sigma' + \{l_3 \mapsto_1 m3_{k_1,k_3}\}\\ \sigma_2 \equiv \sigma' + \{l_3 \mapsto_1 (m_1\, m_2 + m_3)_{k_1,k_3}\}\end{array}}{\langle \sigma_1, \mathbf{gemm}[fc_1]\, l_1 [fc_2]\, l_2\, l_3\rangle \to \langle \sigma_2, ((l_1, l_2), l_3)\rangle}\ \textsc{Op\_Gemm\_Match}$$

3.3 Logical Relation

First, we define an interpretation of heaps with fractional permissions in the style of Bornat et. al [10] (interpreting the multiset as a partial map from locations to the sum of all its associated fractions and a matrix) as well as the n-fold iteration of $\to$.

$$\mathcal{H}\llbracket\sigma\rrbracket = \bigstar_{(l,f,m)\in\sigma}[l \mapsto_f m]$$

where

$$(\varsigma_1 \star \varsigma_2)(l) \equiv \begin{cases} \varsigma_1(l) & \text{if } l \in \text{dom}(\varsigma_1) \wedge l \notin \text{dom}(\varsigma_2)\\ \varsigma_2(l) & \text{if } l \in \text{dom}(\varsigma_2) \wedge l \notin \text{dom}(\varsigma_1)\\ (f_1 + f_2, m) & \text{if } (f_1, m) = \varsigma_1(l) \wedge (f_2, m) = \varsigma_2(l) \wedge f_1 + f_2 \leq 1\\ \text{undefined} & \text{otherwise}\end{cases}$$

We then define a step-indexed logical relation in the style of Ahmed et. al [2]. $(\varsigma, v) \in \mathcal{V}_k\llbracket t\rrbracket$ means it takes a heap with exactly ς resources to produce a value v of type t in at most k steps. So, something like a **unit** or a $!t$ need no resources, whereas a f **mat** needs exactly f ownership of a some matrix and a pair needs a $\star$ combination of the heaps required for each component.

$$\begin{aligned}\mathcal{V}_k\llbracket\mathbf{unit}\rrbracket &= \{(\emptyset, *)\}\\ \mathcal{V}_k\llbracket f\ \mathbf{mat}\rrbracket &= \{(\{l \mapsto_{2^{-f}} \_\}, l)\}\\ \mathcal{V}_k\llbracket !t\rrbracket &= \{(\emptyset, \mathbf{Many}\ v) \mid (\emptyset, v) \in \mathcal{V}_k\llbracket t\rrbracket\}\\ \mathcal{V}_k\llbracket t_1 \otimes t_2\rrbracket &= \{(\varsigma_1 \star \varsigma_2, (v_1, v_2)) \mid (\varsigma_1, v_1) \in \mathcal{V}_k\llbracket t_1\rrbracket \wedge (\varsigma_2, v_2) \in \mathcal{V}_k\llbracket t_2\rrbracket\}\end{aligned}$$

The definition of $\mathcal{V}_k\llbracket 'fc.\ t\rrbracket$ says a value and heap must be the same regardless of what fraction is substituted into both; the $k-1$ is to take into account fraction specialisation takes one step (Op_Spc).

$$\mathcal{V}_k\llbracket 'fc.\ t\rrbracket = \{(\varsigma, \mathbf{fun}\ 'fc \to v) \mid \forall f.\ (\varsigma[f/fc], v[f/fc]) \in \mathcal{V}_{k-1}\llbracket t[f/fc]\rrbracket\}$$

To understand the definition of $\mathcal{V}_k\llbracket t' \multimap t\rrbracket$, we must first look at $\mathcal{C}_k\llbracket t\rrbracket$, the computational interpretation of types. Intuitively, it is a combination of a frame rule on heaps (no interference), type-preservation and termination (in $j < k$ steps) to either an error or a heap-and-expression. For the case of termination to a heap-and-expression, there is a further condition: if the expression is a value syntactically then it is also one semantically.

$$\begin{aligned}\mathcal{C}_k\llbracket t\rrbracket = \{(\varsigma_s, e_s) \mid \forall j < k, \sigma_r.\ \varsigma_s \star \varsigma_r \text{ defined } \Rightarrow \langle \sigma_s + \sigma_r, e_s\rangle \to^j \mathbf{err}\ \vee \exists \sigma_f, e_f.\\ \langle \sigma_s + \sigma_r, e_s\rangle \to^j \langle \sigma_f + \sigma_r, e_f\rangle \wedge (e_f \text{ is a value } \Rightarrow (\varsigma_f \star \varsigma_r, e_f) \in \mathcal{V}_{k-j}\llbracket t\rrbracket)\}\end{aligned}$$

In this light, $\mathcal{V}_k[\![t' \multimap t]\!]$ simply says that v is a function and that evaluating the application of it to any argument (of the correct type, requiring its own set of resources, bounded by k steps) satisfies all the aforementioned properties.

$$\begin{aligned}\mathcal{V}_k[\![t' \multimap t]\!] = \{(\varsigma_v, v) \mid (v \equiv \textbf{fun } x : t' \to e \vee v \equiv \textbf{fix}(g, x : t', e : t)) \wedge \\ \forall j \leq k, (\varsigma_{v'}, v') \in \mathcal{V}_j[\![t']\!].\ \varsigma_v \star \varsigma'_v \text{ defined } \Rightarrow (\varsigma_v \star \varsigma'_v, v\, v') \in \mathcal{C}_j[\![t]\!]\}\end{aligned}$$

The interpretation of typing environments Δ and Γ are with respect to an arbitrary substitution of fractional permissions θ. Note that only the interpretation of Γ involves a (potentially) non-empty heap.

$$\mathcal{I}_k[\![\Delta, x : t]\!]\theta = \{\delta[x \mapsto v_x] \mid \delta \in \mathcal{I}_k[\![\Delta]\!]\theta \wedge (\emptyset, v_x) \in \mathcal{V}_k[\![\theta(t)]\!]\}$$
$$\mathcal{L}_k[\![\Gamma, x : t]\!]\theta = \{(\varsigma \star \varsigma_x, \gamma[x \mapsto v_x]) \mid (\varsigma, \gamma) \in \mathcal{L}_k[\![\Gamma]\!]\theta \wedge (\varsigma_x, v_x) \in \mathcal{V}_k[\![\theta(t)]\!]\}$$

And so the final semantic interpretation of a typing judgement simply quantifies over all possible fractional permission substitutions θ, linear value substitutions γ, intuitionistic value substitutions δ and heaps σ. Note that, $\varsigma \equiv \mathcal{H}[\![\theta(\sigma)]\!]$.

$$\begin{aligned}{}_k[\![\Theta; \Delta; \Gamma \vdash e : t]\!] = \forall \theta, \delta, \gamma, \sigma.\ \Theta = \mathrm{dom}(\theta) \wedge (\varsigma, \gamma) \in \mathcal{L}_k[\![\Gamma]\!]\theta \wedge \delta \in \mathcal{I}_k[\![\Delta]\!]\theta \Rightarrow \\ (\varsigma, \theta(\delta(\gamma(e)))) \in \mathcal{C}_k[\![\theta(t)]\!]\end{aligned}$$

3.4 Soundness Theorem

▶ **Theorem 1.** *(The Fundamental Lemma of Logical Relations)*

$$\forall \Theta, \Delta, \Gamma, e, t.\ \Theta; \Delta; \Gamma \vdash e : t \Rightarrow \forall k.\ {}_k[\![\Theta; \Delta; \Gamma \vdash e : t]\!]$$

3.4.1 Explanation

If an expression e is syntactically type-checked (against a type t), then

- for an arbitrary number of steps k,
- under any substitution of
  - free fractional permission variables θ,
  - linear variables with a suitable heap (γ, ς) and
  - intuitionistic variables δ,
- the aforementioned suitable heap and expression $(\varsigma, \theta(\delta(\gamma(e))))$
- are in the computational interpretation $\mathcal{C}_k[\![\theta(t)]\!]$ of the type t.

The *computational interpretation* is as defined before (Section 3.3); it identifies executions that do no un- or ill-defined behaviours (e.g. adding a boolean and an integer). Since our operational semantics explicitly models deallocation, we now know no well-typed program will ever try to access deallocated memory, establishing the correctness of our memory management checking.

3.4.2 Proof Sketch

To prove the above theorem, we need several lemmas; the interesting ones are: the moral equivalent of the frame rule, monotonicity for the step-index, splitting up environments corresponds to splitting up heaps and heap-and-expressions take the same steps of evaluation under any substitution of their free fractional permissions; all of these (and more) are stated formally and proved in the Appendix.

The proof proceeds by induction on the typing judgement. The case for Ty_Fix is the reason we quantify over the step-index k in the *conclusion* of the soundness theorem. It allows us to then induct over the step-index and assume exactly the thing we need to prove at a smaller index.

The case for Ty_Gen follows a similar pattern, but has the extra complication of reducing an expression with an arbitrary fractional permission variable in it, and then instantiating it at the last moment to conclude, which is where one of the lemmas (heap-and-expressions take the same steps of evaluation under any substitution of their free fractional permissions) is used.

The rest of the cases are either very simple base cases (variables, unit, boolean, integer or element literals) or follow very similar patterns; for these, only Ty_Let is presented in full and other similar cases simply highlight exactly what would be different. The general idea is to split up the linear substitution and heap along the same split of Γ/Γ', then (by induction) use $\mathcal{C}_k[\![-]\!]$ and one "half" of the linear substitution and heap to conclude the "first" sub-expression either takes $j < k$ steps to **err** or another heap-and-expression.

In the first case, you use Op_Context_Err to conclude the whole let-expression does the same. Similarly we use Op_Context j times in the second case. However, a small book-keeping wrinkle needs to be taken care of in the case that the heap-and-expression turns into a value in $i \leq j$ steps: Op_Context is not functorial for the n-fold iteration of $\rightarrow$. Basically, the following is not quite true:

$$\frac{\langle \sigma, e \rangle \rightarrow^j \langle \sigma', e' \rangle}{\langle \sigma, C[e] \rangle \rightarrow^j \langle \sigma', C[e'] \rangle} \quad \textsc{Op\_Context}$$

because after the i steps, we need to invoke Op_Let_Var to proceed evalution for any remaining $j - i$ steps. After that, it suffices to use the induction hypothesis on the second sub-expression to finish the proof. To do so, we need to construct a valid linear substitution and heap (one in $\mathcal{L}_k[\![\Gamma', x : t]\!]\theta$). We take the other "half" of the linear substitution and heap (from the inital split at the start) and extend it with $[x \mapsto v]$, (where x is the variable bound in the let-expression and v is the value we assume the first sub-expression evaluated to in i steps).

4 Implementation

4.1 Implementation Strategy

NumLin transpiles to OCaml and its implementation follows the structure of a typical domain-specific language (DSL) compiler. Although NumLin's current implementation is not as an embedded DSL, its the general design is simple enough to adapt to being so and also to target other languages.

Alongside the transpiler, a "Read-Check-Translate" loop, benchmarking program and a test suite are included in the artifacts accompanying this paper.

1. **Parsing.** A generated, LR(1) parser parses a text file into a syntax tree. In general, this part will vary for different languages and can also be dealt with using combinators or syntax-extensions (the EDSL approach) if the host language offers such support.
2. **Desugaring.** The syntax tree is then desugared into a smaller, more concise, abstract syntax tree. This allows for the type checker to be simpler to specify and easier to implement.

3. **Matrix Expressions** are also desugared into the abstract syntax tree through pattern-matching.
4. **Type checking.** The abstract syntax tree is explicitly typed, with some inference to make writing typical programs more convenient.
5. **Code Generation.** The abstract syntax tree is translated into OCaml, with a few "optimisations" to produce more readable code. This process is type-preserving: NumLin's type system is embedded into OCaml's (Figure 14), so the OCaml type checker acts as a sanity check on the generated code.

A very pleasant way to use NumLin is to have the build system generate code at *compile-time* and then have the generated code be used by other modules like normal OCaml functions. This makes it possible and even easy to use NumLin alongside existing OCaml libraries; in fact, this is exactly how the benchmarking program and test-suite use code written in NumLin.

4.1.1 Desugaring, Matrix Expressions and Type Checking

As seen earlier (Figure 2), desugaring is conventional. Matrix expressions are translated into BLAS/LAPACK calls via purely syntactic pattern-matching (also seen earlier in Figure 3).

4.1.2 Type checking

Type checking is mostly standard for a linearly typed language, with the exception of fractional permission inference. By restricting fractions to be non-positive integer powers of two, we only need to keep track of the logarithm of the fractions used. Explicit sharing and unsharing removes the need for performing dataflow analysis. As a result, all fractional arithmetic can be solved with unification, and in doing so, fractions become directly usable in NumLin's type-system as opposed to a convenient theoretical tool.

Because all functions must have their argument types explicitly annotated, inferring the correct fraction at a call-site is simply a matter of unification. We believe *full-inference of fractional permissions is similarly just matter of unification* (thanks to an experimental implementation of just this feature), even though the formal system we present here is for an explicitly-typed language.

There are a few differences between the type system as presented in 3.2 and how we implemented it: the environment *changes* as a result of type checking an expression (the standard transformation to avoid a non-deterministic split of the environment for checking pairs); variables are *marked as used* rather than removed for better error messages; variables are *tagged* as linear or intuitionistic in *one* environment as opposed to being stored in *two* separate ones (this allows scoping/variable look-up to be handled uniformly).

4.1.3 Code Generation

Code generation is a straightforward mapping from NumLin's core constructs to high-level OCaml ones. We embed NumLin's type- and term- constructors into OCaml as a sanity check on the output (Figure 14).

This is also useful when using NumLin from within OCaml; for example, we can use existing tools to inspect the type of the function we are using (Figure 15). It is worth reiterating that only the type- and term- constructors are translated into OCaml, NumLin's precise control over linearity and aliasing are not brought over.

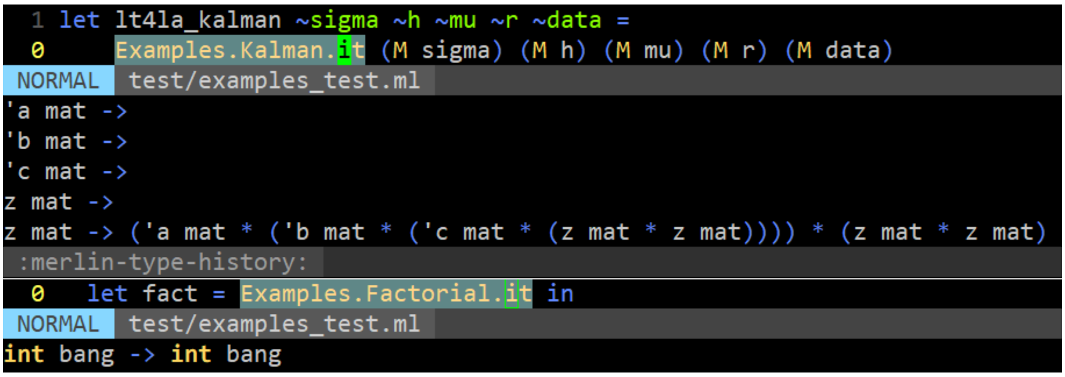
We actually use this fact to our advantage to clean up the output OCaml by removing what would otherwise be redundant re-bindings (Figure 16). Combined with a code-formatter,

$$
\begin{array}{lll}
f & ::= & \\
 & \mid & {}'\!fc \\
 & \mid & \mathbf{z} \\
 & \mid & f\ \mathbf{s} \\
t & ::= & \\
 & \mid & \mathbf{unit} \\
 & \mid & \mathbf{bool} \\
 & \mid & \mathbf{int} \\
 & \mid & \mathbf{elt} \\
 & \mid & f\ \mathbf{arr} \\
 & \mid & f\ \mathbf{mat} \\
 & \mid & !\ t \\
 & \mid & {}'\!fc.\ t \\
 & \mid & t \otimes t' \\
 & \mid & t \multimap t'
\end{array}
$$

```
module Arr =
  Owl.Dense.Ndarray.D

type z = Z
type 'a s = Succ

type 'a arr =
  A of Arr.arr
  [@@unboxed]

type 'a mat =
  M of Arr.arr
  [@@unboxed]

type 'a bang =
  Many of 'a
  [@@unboxed]
```

$$
\begin{aligned}
\llbracket {}'\!fc \rrbracket &= \texttt{'fc} \\
\llbracket \mathbf{z} \rrbracket &= \texttt{z} \\
\llbracket f\ \mathbf{s} \rrbracket &= \llbracket f \rrbracket\ \texttt{s} \\
\llbracket \mathbf{unit} \rrbracket &= \texttt{unit} \\
\llbracket \mathbf{bool} \rrbracket &= \texttt{bool} \\
\llbracket \mathbf{int} \rrbracket &= \texttt{int} \\
\llbracket \mathbf{elt} \rrbracket &= \texttt{float} \\
\llbracket f\ \mathbf{arr} \rrbracket &= \llbracket f \rrbracket\ \texttt{arr} \\
\llbracket f\ \mathbf{mat} \rrbracket &= \llbracket f \rrbracket\ \texttt{mat} \\
\llbracket !\ t \rrbracket &= \llbracket t \rrbracket\ \texttt{bang} \\
\llbracket {}'\!fc.\ t \rrbracket &= \llbracket t \rrbracket \\
\llbracket t \otimes t' \rrbracket &= \llbracket t \rrbracket \texttt{*} \llbracket t' \rrbracket \\
\llbracket t \multimap t' \rrbracket &= \llbracket t \rrbracket \to \llbracket t' \rrbracket
\end{aligned}
$$

Figure 14 NumLin's type grammar (left) and its embedding into OCaml (right).



```
let lt4la_kalman ~sigma ~h ~mu ~r ~data =
      Examples.Kalman.it (M sigma) (M h) (M mu) (M r) (M data)
NORMAL  test/examples_test.ml
'a mat ->
'b mat ->
'c mat ->
z mat ->
z mat -> ('a mat * ('b mat * ('c mat * (z mat * z mat)))) * (z mat * z mat)
 :merlin-type-history:
    let fact = Examples.Factorial.it in
NORMAL  test/examples_test.ml
int bang -> int bang
```

Figure 15 Using NumLin functions from OCaml.

the resulting code is not obviously correct and exactly what an expert would intend to write by hand, but now with the guarantees and safety of NumLin behind it. A small example is shown in Figure 17, a larger one in the Appendix.

4.2 Performance Metrics

We think that using NumLin has two primary benefits: safety and performance. We discuss safety in 5.1, where we describe how we used NumLin to find linearity and aliasing bugs in a linear algebra algorithm that was *generated* by another program.

4.2.1 Setup

For performance, we measured the execution times of four equivalent implementations of a Kalman filter: in C (using CBlas), NumLin (using Owl's low-level CBlas bindings), OCaml (using Owl's intended, safe/copying-by-default interface), and Python (using NumPy, with the interpreter started and functions interpreted). We measured execution time in micro-

```
          let Many x = x in
          let Many x = Many (Many x) in <exp>   ⇒   <exp>

       (* fixp = fix (f, x:t, <exp> : t') *)
       (*1*) let Many f = Many fixp in <body>   ⇒   let rec f x = <exp> in <body>
       (*2*) let f = fixp in <body>

   (*1*) let Many x = <exp> in
   (*-*) let Many x = Many (Many x) in <body>
                                                ⇒   let x = <exp> in <body>
   (*2*) let Many x = Many <exp> in <body>
   (*3*) (fun x : t -> <body>) <exp>
```

Figure 16 Removing redundant re-bindings during translation to OCaml.

```
let rec sum_array i n x0 row =
  if Prim.extract @@ Prim.eqI i n then (row, x0)
  else
    let row, x1 = Prim.get row i in
    sum_array (Prim.addI i (Many 1)) n (Prim.addE x0 x1) row
in
sum_array
```

Figure 17 Recursive OCaml function for a summing over an array, generated (at *compile time*) from the code in Figure 5, passed through `ocamlformat` for presentation.

seconds, against an exponentially (powers of 5) increasing scaling factor for matrix size parameters $n = 5$ and $k = 3$.

For large scaling factors ($n = 5^4, 5^5$), we triggered a full garbage-collection before measuring the execution time of a single call of a function. However, due to the limitations of the micro-benchmarking library we used, for smaller scaling factors ($n = 5^1, 5^2, 5^3$), we measured the execution time of *multiple* calls to a function in a loop, thus including potential garbage-collection effects.

We also measured the execution times of L1-norm minimisation and the “linear-regression” ($(\mathbf{X}^{\mathbf{T}}\mathbf{X})^{-\mathbf{1}}\mathbf{X}^{\mathbf{T}}\mathbf{y}$) similary, but without a C implementation.

4.2.2 Hypothesis

We expected the C implementation to be faster than the NumLin one because the latter has the additional (but relatively low) overhead of dimension checks and crossing the OCaml/C FFI for each call to a Cblas routine, even though the calls and their order are exactly the same. We expected the OCaml and Python implementations to be slower because they allocate more temporaries (so possibly less cache-friendly) and carry out more floating-point operations – the Cblas and NumLin implementations use ternary kernels (coalescing steps), a Cholesky decomposition (of a symmetric matrix, which is more efficient than the LU decomposition used for inverting a matrix in Owl and NumPy) and `symm` (symmetric matrix multiplication, halving the number of floating-point multiplications required).

4.2.3 Results

The results in Figures 18 are as we expected: C is the fastest, followed by NumLin, with OCaml and Python last. Differences in timings are quite pronounced at small matrix sizes, but are still significant at larger ones. Specifically for the Kalman filter, for $n = 625$, Cblas took $112 \pm 35\,ms$, NumLin took $105 \pm 25\,ms$, Owl took $124 \pm 38\,ms$ and NumPy took $112 \pm 12\,ms$; for $n = 3125$, Cblas took $10.8 \pm 0.7\,s$, NumLin took $12.0 \pm 1.2\,s$, Owl took $13.3 \pm 0.2\,s$ and NumPy took $12.7 \pm 0.6\,s$.

Worth highlighting here is the other major advantage of using NumLin is reduced memory usage. Whilst the Owl and NumPy use 11 temporary matrices for the Kalman filter, (*excluding* the 2 matrices which store the results), using $n + n^2 + 4nk + 3k^2 + 2k \approx 4n^2$ (for $k = 3n/5$) words of memory, Cblas and NumLin use only *2* temporary matrices (excluding the *one* matrix which stores one of the results), using only $n^2 + nk \leq 2n^2$ words of memory.

4.2.4 Analysis

As matrix sizes increase, assuming sufficient memory, the difference in the number of floating-point operations ($O(n^3)$) dominates execution times. However for small matrix sizes, since n is small and the measurements were over multiple calls to a function in a loop, the large number of temporaries show the adverse effect of not re-using memory at even quite small matrix sizes: creating pressure on the garbage collector.

5 Discussion and Related Work

5.1 Finding Bugs in SymPy's Output

Prior to this project, we had little experience with linear algebra libraries or the problem of matrix expression compilation. As such, we based our initial NumLin implementation of a Kalman filter using BLAS and LAPACK, on a popular GitHub gist of a Fortran implementation, one that was *automatically generated* from SymPy's matrix expression compiler [18].

Once we translated the implementation from Fortran to NumLin, we attempted to compile it and found that (to our surprise) it did not type-check. This was because the original implementation contained incorrect aliasing, unused variables and unnecessary temporaries, and did not adhere to Fortran's read/write permissions (with respect to `intent` annotations `in`, `out` and `inout`) all of which were now highlighted by NumLin's type system.

The original implementation used 6 temporaries, one of which was immediately spotted as never being used due to linearity. It also contained two variables which were marked as `intent(in)` but would have been written over by calls to "gemm", spotted by the fractional permissions feature. Furthermore, it used a matrix *twice* in a call to "symm", once with a read permission but once with a *write* permission. Fortran assumes that any parameter being written to is not aliased and so this call was not only incorrect, but illegal according to the standard, both aspects of which were captured by linearity and fractional permissions.

Lastly, it contained another unnecessary temporary, however one that was not obvious without linear types. To spot it, we first performed live-range splitting (checked by linearity) by hoisting calls to `freeM` and then annotated the freed matrices with their dimensions. After doing so and spotting two disjoint live-ranges of the same size, we replaced a call to `freeM` followed by allocating call to `copy` with one, in-place call to `copyM_to`. We believe the ability to boldly refactor code which manages memory is good evidence of the usefulness of linearity as a tool for programming.

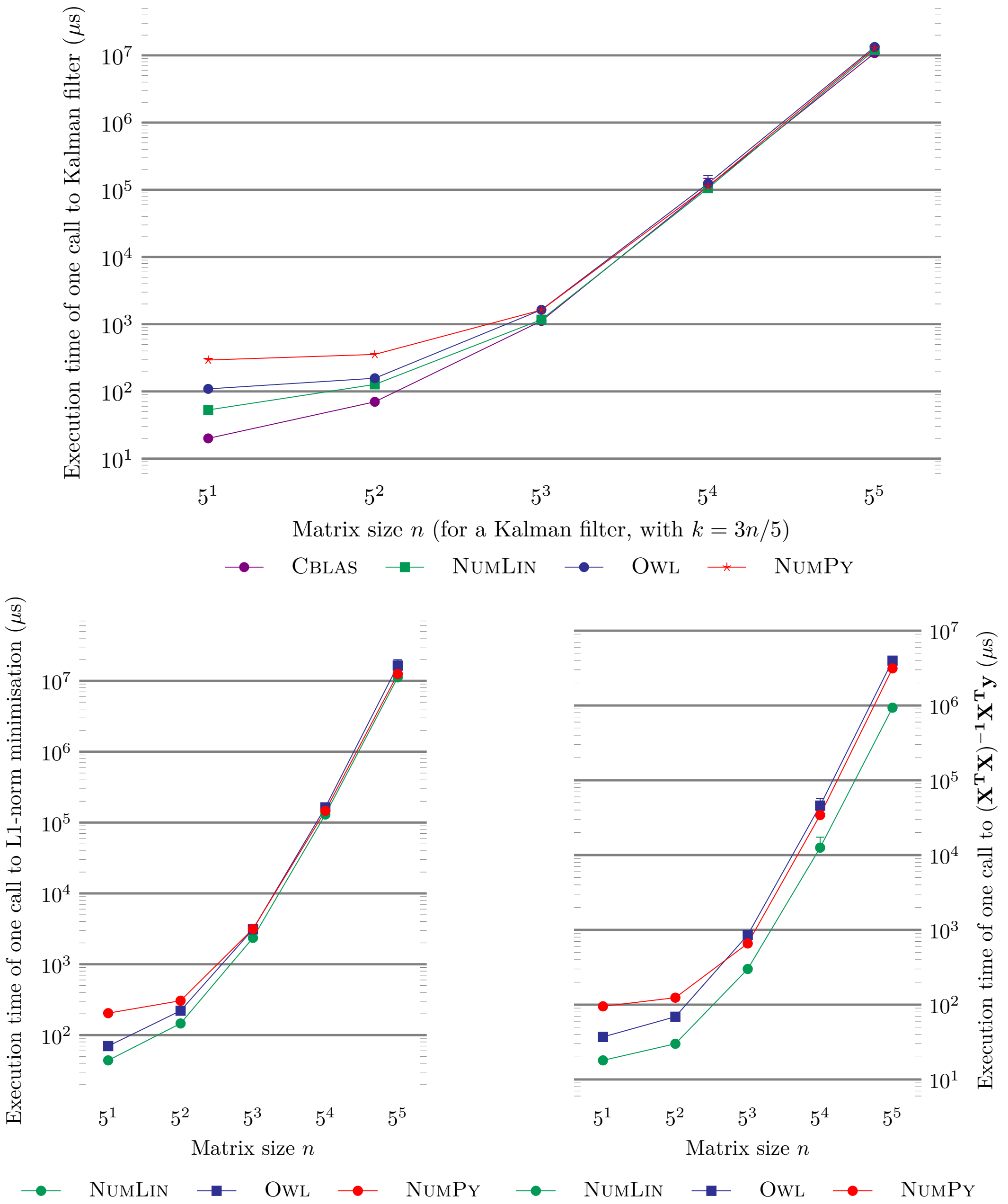


Figure 18 Comparison of execution times (error bars are present but quite small). Small matrices and timings $n \leq 5^3$ were micro-benchmarked with the Core_bench library. Larger ones used Unix's `getrusage` functionality, sandwiched between calls to `Gc.full_major` for the OCaml implementations.

5.2 Related Work

5.2.1 Linear types for *implementing* linear algebra routines

Using linear types for BLAS routines is a particularly good domain fit (given the implicit restrictions on aliasing arguments), and as a result the idea of using substructural types to express array computations is not a particularly new one [19, 14, 7]. However, many of these designs have been focused on building languages to *implement* the kernel linear algebra functions, and as a result, they tend to add additional limitations on the language design. Both Futhark [14] and Single Assignment C [19] omit higher-order functions to facilitate compilation to GPUs. The work of [7] forbids term-level recursion, in order to ensure that all higher-order computations can be statically normalized away and thereby maximize opportunities for array fusion.

5.2.2 Our contribution: linear types for enforcing correct *usage* of linear algebra routines

In contrast, our approach is to begin with the assumption that we can take existing efficient BLAS-like libraries, and then enforce their correct *usage* using a linear type discipline with fractional permissions.

5.2.3 Traditionally complex approaches to sharing

Our approach is similar to the one taken in linear algebra libraries for Rust – these libraries typically take advantage of the distinction that Rust's type system offers between mutable views/references to arrays. The work of [21] and [15] suggest that Rust's borrow-checker *can be expressed in simpler terms* using fractional permissions, though to our knowledge the programmer-visible lifetime analysis in Rust has never been formalized.

Working explicitly with fractional permissions has two main benefits. First, our type system demonstrates that type systems for fractional permissions can be dramatically simpler than existing state-of-the-art approaches, including both industrial languages like Rust, as well as academic (such as those developed by [9]). Bierhoff *et al*'s type system, much like Rust's, builds a complex dataflow analysis into the typing rules to infer when variables can be shared or not. This allows for more natural-looking user programs, but can create the impression that using fractional permissions requires a heavy theoretical and engineering effort going well beyond that needed for supporting basic linear types.

5.2.4 Our contribution: a simpler approach to sharing

Instead, our approach, of requiring sharing to be made explicit, lets us demonstrate that the existing unification machinery already in place for ordinary ML-style type inference can be reused to support fractions. Basically, we can view sharing a value as dividing a fraction by two, and after taking logarithms all fractions are Peano numbers, whose equality can be established with ordinary unification.

5.2.5 Implications

This fact is important because there are major upcoming implementations of linear types such as Linear Haskell [6], which do not have built-in support for fractional permissions. Instead, Linear Haskell takes a slightly different definition of linearity, one based on *arrows* as opposed to *kinds*: for $f : a \multimap b$, if $f\,u$ is used exactly once *then* u is used exactly once. Whilst this has the advantage of being backwards-compatible, it also means that the type system has no built-in support for the concurrent reader, exclusive writer pattern that fractional permissions enable.

However, since our type system shows unification is "all one needs" for fractions, it should be possible to *encode* NumLin's approach to fractional permissions in Linear Haskell by adding a GADT-style natural number index to array types tracking the fraction, which should enable supporting high-performance BLAS bindings in Linear Haskell. Actually implementing this is something we leave for future work, as there remains one issue which we do not see a good encoding for. Namely, only having support for linear functions makes it a bit inconvenient to manipulate linear values directly – programs end up taking on a CPS-like structure. This seems to remain an advantage of a direct implementation of linear types over the Linear Haskell style.

5.3 Simplicity and Further Work

We are pleasantly surprised at how simple the overall design and implementation of NumLin is, given its expressive power and usability. So simple in fact, that fractions, a convenient theoretical abstraction until this point, could be implemented by restricting division and multiplication to be by 2 only [11], thus turning any required arithmetic into unification.

Indeed, the focus on getting a working prototype early on (so that we could test it with real BLAS/LAPACK routines as soon as possible) meant that we only added features to the type system when it was clear that they were absolutely necessary: these features were !-types and value-restriction for the `Many` constructor.

Going forwards, one may wish to eliminate even more runtime errors from NumLin, by extending its type system. For example, we could have used existential types to statically track pointer identities [2], or parametric polymorphism.

We could also attempt to catch mismatched dimensions at compile time as well. While this could be done with generative phantom types [1], using dependent types may offer more flexibility in *partitioning* regions [17] or statically enforcing dimensions related constraints of the arguments at compile-time. ATS [13] is an example of a language which combines linear types with a sophisticated proof layer. But although it provides BLAS bindings, it does not aim to provide aliasing restrictions as demonstrated in this paper.

Taking this idea one step even further, since matrix dimensions are typically fixed at runtime, we could *stage* NumLin programs and compile matrix expressions using more sophisticated algorithms [4]. However, it is worth noting that without care, such algorithms [18], usually based on graph-based, ad-hoc dataflow analysis, can produce erroneous output which would not get past a linear type system with fractions.

We also think that this concept (and the general design of its implementation) need not be limited to linear algebra: we could conceivably "backport" this idea to other contexts that need linearity (concurrency, single-use continuations, zero-copy buffer, streaming I/O) or combine it with dependent types to achieve even more expressive power to split up a single block of memory into multiple regions in an arbitrary manner [17].

References

1 Akinori Abe and Eijiro Sumii. A simple and practical linear algebra library interface with static size checking. *arXiv preprint*, 2015. `arXiv:1512.01898`.

2 Amal Ahmed, Matthew Fluet, and Greg Morrisett. L^ 3: a linear language with locations. *Fundamenta Informaticae*, 77(4):397–449, 2007.

3 Andrew Barber and Gordon Plotkin. *Dual intuitionistic linear logic*. University of Edinburgh, Department of Computer Science, Laboratory for . . . , 1996.

4 Henrik Barthels, Marcin Copik, and Paolo Bientinesi. The generalized matrix chain algorithm. *arXiv preprint*, 2018. `arXiv:1804.04021`.

5 P Benton. A mixed linear and non-linear logic: Proofs, terms and models. In *Computer Science Logic*, pages 121–135. Springer, 1995.

6 Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R Newton, Simon Peyton Jones, and Arnaud Spiwack. Linear Haskell: practical linearity in a higher-order polymorphic language. *Procedings of the ACM on Programming Languages*, 2(POPL):5, 2017.

7 Jean-Philippe Bernardy, Vıctor López Juan, and Josef Svenningsson. Composable efficient array computations using linear types. *Unpublished Draft*, 2016.

8 Kevin Bierhoff and Jonathan Aldrich. PLURAL: checking protocol compliance under aliasing. In *Companion of the 30th international conference on Software engineering*, pages 971–972. ACM, 2008.

9 Kevin Bierhoff, Nels E Beckman, and Jonathan Aldrich. Fraction Polymorphic Permission Inference.

10 Richard Bornat, Cristiano Calcagno, Peter O'Hearn, and Matthew Parkinson. Permission accounting in separation logic. In *ACM SIGPLAN Notices*, volume 40 (1), pages 259–270. ACM, 2005.

11 John Boyland. Checking interference with fractional permissions. In *International Static Analysis Symposium*, pages 55–72. Springer, 2003.

12 Alex Bronstein, Yoni Choukroun, Ron Kimmel, and Matan Sela. Consistent discretization and minimization of the l1 norm on manifolds. In *3D Vision (3DV), 2016 Fourth International Conference on*, pages 435–440. IEEE, 2016.

13 Sa Cui, Kevin Donnelly, and Hongwei Xi. Ats: A language that combines programming with theorem proving. In *International Workshop on Frontiers of Combining Systems*, pages 310–320. Springer, 2005.

14 Troels Henriksen, Niels GW Serup, Martin Elsman, Fritz Henglein, and Cosmin E Oancea. Futhark: purely functional GPU-programming with nested parallelism and in-place array updates. *ACM SIGPLAN Notices*, 52(6):556–571, 2017.

15 Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. RustBelt: securing the foundations of the rust programming language. *PACMPL*, 2(POPL):66:1–66:34, 2018. `doi:10.1145/3158154`.

16 Rudolph Emil Kalman. A new approach to linear filtering and prediction problems. *Journal of basic Engineering*, 82(1):35–45, 1960.

17 Conor McBride. Code Mesh London 2016, Keynote: SpaceMonads. `https://www.youtube.com/watch?v=QojLQY5H0RI`. Accessed: 08/05/2018.

18 Matthew Rocklin. Mathematically informed linear algebra codes through term rewriting, 2013.

19 Sven-Bodo Scholz. Single Assignment C: efficient support for high-level array operations in a functional setting. *Journal of functional programming*, 13(6):1005–1059, 2003.

20 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, April 1990. North-Holland.

21 Aaron Weiss, Daniel Patterson, and Amal Ahmed. Rust Distilled: An Expressive Tower of Languages. *arXiv preprint*, 2018. `arXiv:1806.02693`.

Deep Static Modeling of `invokedynamic`

George Fourtounis
University of Athens, Department of Informatics and Telecommunications, Greece
gfour@di.uoa.gr

Yannis Smaragdakis
University of Athens, Department of Informatics and Telecommunications, Greece
smaragd@di.uoa.gr

Abstract

Java 7 introduced programmable dynamic linking in the form of the `invokedynamic` framework. Static analysis of code containing programmable dynamic linking has often been cited as a significant source of unsoundness in the analysis of Java programs. For example, Java lambdas, introduced in Java 8, are a very popular feature, which is, however, resistant to static analysis, since it mixes `invokedynamic` with dynamic code generation. These techniques invalidate static analysis assumptions: programmable linking breaks reasoning about method resolution while dynamically generated code is, by definition, not available statically. In this paper, we show that a static analysis can predictively model uses of `invokedynamic` while also cooperating with extra rules to handle the runtime code generation of lambdas. Our approach plugs into an existing static analysis and helps eliminate all unsoundness in the handling of lambdas (including associated features such as method references) and generic `invokedynamic` uses. We evaluate our technique on a benchmark suite of our own and on third-party benchmarks, uncovering all code previously unreachable due to unsoundness, highly efficiently.

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, invokedynamic

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2019.15

Supplement Material ECOOP 2019 Artifact Evaluation approved artifact available at
`https://dx.doi.org/10.4230/DARTS.5.2.6`

Funding We gratefully acknowledge funding by the European Research Council, grants 307334 (SPADE) and 790340 (PARSE), a Facebook Research and Academic Relations award, and an Oracle Labs collaborative research grant.

1 Introduction

Object-oriented and functional programming have combined in recent years to produce hybrid programming languages. Some of these, such as Scala [45], are new languages, designed from the ground up to incorporate features of both programming paradigms. Others, for instance Java [23] and C# [25], have adapted to the demand for functional features by carefully adding them in an existing language design; examples of this evolution are lambdas [49] and the streams API [74] in the Java platform and the Language Integrated Query (LINQ) facility in the .NET ecosystem [36].

On another axis, programming languages occupy different places in the spectrum between static and dynamic typing. At the extremes, programming languages either have to supply static ("type") information for every entity in the program, or do away with all such types, in a completely dynamic coding style. In practice, most programming languages are closer to the middle, having a fundamental static or dynamic design, while mixing elements from

33rd European Conference on Object-Oriented Programming (ECOOP 2019).
Editor: Alastair F. Donaldson; Article No. 15; pp. 15:1–15:28
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

the opposite approach. For example, the Java Virtual Machine (JVM), the best-established language runtime system, supports dynamic facilities, such as reflection and dynamic class loading, that offer significant flexibility, outside the control of the static type system.

A recent dynamic facility added to the JVM, in order to combine flexibility with highly optimized performance, is that of programmable method resolution and dynamic linking, in the form of the `invokedynamic` instruction [62]. The `invokedynamic` instruction and its accompanying `java.lang.invoke` framework permit the expression of fully dynamic behavior, in much the same way as traditional Java reflection. However, whereas reflection can be thought of as dynamically *interpreting* dispatch logic, programmable linking can be thought of as dynamically *compiling* dispatch logic, transforming call sites at load time with decisions possibly cached and subsequently executed at full speed. This facility enables the JVM to support dynamic language patterns with great efficiency. As a result, the framework has also been used to implement Java lambdas – the newly-added functional feature of the language.[1]

Dynamic features are welcome by many programmers as they offer a needed flexibility. However, they come at a cost: static reasoning is greatly hindered. For instance, static analysis tools for Java are largely ineffective when faced with `invokedynamic` code, although static analysis has long dealt with (statically-typed) dynamic dispatch (a.k.a. *virtual dispatch*) facilities. Virtual method resolution in statically-typed bytecode is much easier to analyze, compared to purely dynamic code that lacks explicit method signatures. (Virtual dispatch in standard object-oriented languages performs a dynamic lookup of the function, based on its name, signature, and the type hierarchy. This is still significantly friendlier to static reasoning than completely dynamic calls, of functions with possibly statically-unknown names or types.)

These problems of static reasoning for the dynamic features of the JVM (and, by extension, its functional lambdas) have been well identified. In recent work, *Reif et al.* [60] and *Sui et al.* [68] describe the unsoundness in the construction of call graphs for Java, caused by features such as lambdas and `invokedynamic`. These features are not going away: in a recent study, Mazinanian et al. [35] "found an increasing trend in the adoption rate of lambdas." Also, Holzinger et al. found method handles, a core part of the `invokedynamic` framework, to pose "a risk to the secure implementation of the Java platform" [26]. This is a design problem: to control performance overhead, method handles are less secure by design, compared to the core reflection API [65].

In this paper, we propose a static analysis that can successfully analyze both the `invokedynamic` framework and its particular combination with generated code in Java lambdas. Our analysis cooperates with an existing points-to analysis and an existing reflection analysis (when needed), in mutually recursive fashion. The analysis also simulates parts of the Java API that either do dynamic code generation or call native code, to maintain soundness. Finally, we supply a special static analysis extension that can analyze lambdas and method references, without any reflection support. This last feature permits the static analysis of large Java code bases without paying the performance overhead of reflection reasoning.

[1] We emphasize again that the concepts of lambdas (a functional language feature) and programmable linking (a dynamic language implementation technique) are orthogonal. Lambdas could be implemented via front-end class generation, dynamic code generation plus traditional virtual dispatch, or other similar techniques. They are implemented using programmable linking in Oracle's JDK only as a matter of choice, since the mechanism is flexible, powerful, and efficient.

In more detail, our work makes the following contributions:

- We offer the first static analysis that handles general-purpose `invokedynamic` – the basis of modern dynamic features of Java. The static analysis operates at a deep level: it includes full modeling of the underlying `java.lang.invoke` framework: a DSL-like facility for capturing and manipulating methods as values.
- We present a static modeling of Java lambdas – the main functional feature of Java. Although lambdas and `invokedynamic` are conceptually orthogonal, in practice lambdas are implemented using `invokedynamic`, making the analyses of the two features closely interrelated. Still, the analysis of lambdas is not a mere client of the general-purpose `invokedynamic` analysis, since it both needs extra modeling (for generated code) and admits more efficient implementation, due to its specialized use of `invokedynamic`, eschewing the need for complex reflection reasoning.
- The analysis is accompanied by a micro-benchmarking suite covering many patterns found in realistic uses of lambdas and `invokedynamic`. The suite is independently usable for validation of static support of these features.
- The analysis is evaluated on the third-party suite of *Sui et al.* [68], which was designed for showcasing the unsoundness of call-graph construction under dynamic and functional Java features. Our analysis models all general-purpose uses of `invokedynamic` and fully models uses of lambdas.

This paper is structured as follows: we first present a set of examples that explain how the dynamic and functional features of Java work (Section 2) and proceed to give a more technical background of these features (Section 3). We then present our technique for the static analysis of these features, in a declarative analysis framework (Section 4). We evaluate our model (Section 5), connect with related work (Section 6), and conclude (Section 7).

2 Motivation and Illustration

This section introduces `invokedynamic` and Java lambdas with the help of examples.

2.1 Motivating Example 1: Late Linking

A common use of dynamic linking is for breaking dependencies between pieces of code so that they do not have to be compiled together. An example of Java code using `invokedynamic` to break a compile-time dependency is shown in Figure 1. Since Java does not permit `invokedynamic`-equivalent expressions at the source level,[2] we use in the example an INVOKEDYNAMIC pseudo-intrinsic that contains the following information:

- a dynamic name (`print`),
- a method type (`(A)V`),
- a list of arguments (just `this.obj` here),
- a bootstrap method signature (here: `<A: CallSite bootstrap(MethodHandles.Lookup, String, MethodType)>`), and
- a list of bootstrap arguments (empty in this example).

While the code without `invokedynamic` has to explicitly state which version to call (and thus store an immutable signature in an `invokestatic` in the bytecode), the code using `invokedynamic` looks up the method programmatically, via a "bootstrap" method, which initializes the call site. (This lookup could be arbitrarily complex, although in this

[2] A proposal is underway to allow such expressions via intrinsics [21].

example the outcome is always the same.) Here we note that the programmer could also use classic Java reflection to do a similar lookup-and-invoke (retrieving a `Method` metaobject and calling an `invoke` method on it), but that would be inefficient, since standard Java reflection contains an interpretive layer of introspection. In contrast, `invokedynamic` can be compiled away: the bootstrap method is executed at *load time*, not run time (i.e., not when method `run` is invoked, but when it is loaded). The bootstrap method effectively acts as a load-time macro, accepting as arguments load-time constants (e.g., string constants) or fragments of uninterpreted expression syntax. This bootstrap method returns a "constant call site", which the JVM can inline in place of the `invokedynamic` call as needed, similar to having the `invokestatic` call that is missing from the bytecode.

2.2 Motivating Example 2: Lambdas

For a simple program that creates and uses a lambda, we can take the following example (adapted from the dynamic benchmark of Sui et al. [68]):

```
import java.util.function.Consumer;
public class LambdaConsumer {

  public void source() {
    Consumer<String> c = (input) -> target(input);
    c.accept("input");
  }

  public void target(String input) { }

}
```

Here, method `source()` creates a lambda that consumes a string value. The lambda takes an `input` parameter and calls method `target()` in its body, passing the parameter to the callee.

The arrow syntax declares a lambda function, which is rather a mismatch for object orientation: it looks like a bare method, without an instance or declaring type. However, that syntax behind the scenes constructs an object of type `Consumer`, as shown by the static type of variable `c`. This type is one of the "functional interfaces" [16] provided by Java, which are interface types that have a functional flavor, i.e., declare a single method. Generic typing helps with annotating uses of such instances (as with the type parameter of `Consumer` here).

Indeed, the Consumer type declares a single `accept` method that takes a `String`. Calling that method on a lambda should then evaluate the body of the lambda with the appropriate parameter passed to it. If we were to inline the code in the body of `source()` to eliminate the lambda, it would read:

```
public void source() {
  target("input");
}
```

However, such inlining cannot happen in the general case: lambdas are often passed to code or returned by it, to be applied in a location remote to their origin. Reasoning about the code above is thus based on non-local (possibly whole-program) reasoning about the "functional object" that was created and assigned to variable `c`.

In Figure 2, we see the bytecode generated for the two statements in the body of `source` in our example. For presentation purposes, instead of stack-based bytecode, we use the friendlier

Code without invokedynamic

```
class C implements Runnable {
  A obj;

  C(A obj) {
    this.obj = obj;
  }

  void run() {
    A.print(this.obj);       // Direct call
  }
}

class A {
  public static void print(A a) { }
}

(new C(new A())).run();
```

Code using invokedynamic

```
class C implements Runnable {
  A obj;

  C(A obj) {
    this.obj = obj;
  }

  void run() {
    INVOKEDYNAMIC "print" "(A)V" [this.obj]
      <A: CallSite bootstrap(MethodHandles.Lookup,String,MethodType)>
      []
  }
}

class A {
  public static void print(A a) { }
  public static CallSite bootstrap(MethodHandles.Lookup caller,
                                   String name, MethodType type) {
    MethodType mt = MethodType.methodType(Void.TYPE, A.class);
    MethodHandles.Lookup lookup = MethodHandles.lookup();
    MethodHandle handle = lookup.findStatic(A.class, name, mt);
    return new ConstantCallSite(handle);
  }
}

(new C(new A())).run();
```

Figure 1 Example: using `invokedynamic` to postpone linking of a method call.

```
l0 := @this: LambdaConsumer;

l1 = dynamicinvoke "accept" <java.util.function.Consumer (LambdaConsumer)>(l0)
       <invoke.LambdaMetafactory: invoke.CallSite metafactory(
         invoke.MethodHandles$Lookup,String,invoke.MethodType,
         invoke.MethodType,invoke.MethodHandle,invoke.MethodType)>
       (class "(Ljava/lang/Object;)V",
        handle: <LambdaConsumer: void lambda$source$0(String)>,
        class "(Ljava/lang/String;)V");

interfaceinvoke l1.<java.util.function.Consumer: void accept(Object)>("input");
```

Figure 2 The `invokedynamic` behind a lambda creation (Jimple syntax for example in Section 2.2). The package prefix "`java.lang`" has been removed from all types – e.g., `invoke.MethodType` is `java.lang.invoke.MethodType`.

3-address Jimple intermediate language [75]. (In the Jimple syntax, the `invokedynamic` JVM instruction is denoted `dynamicinvoke`.) We observe that the call that generates the lambda does the following:

- It invokes as a bootstrap method (i.e., the method to execute at load-time over the site of `invokedynamic`) a special "lambda metafactory" method. Again, this is a method executing at load time (i.e., akin to a macro). It processes the call site directly and returns a `CallSite` value, not the `Consumer` value of the user code.
- It passes to the lambda metafactory enough information to specify what kind of lambda needs to be generated: one with an "accept" method, implementing interface `Consumer` and capturing from its environment parameter `l0`. The `l0` capture means that the current value of `this` escapes to the new code that will construct the lambda. This is to be expected, as the lambda body needs a receiver to resolve the call to `target`.
- A method handle pointing to a compiler-generated method `lambda$source$0` is also passed as an argument. This method encodes the body of the lambda expression.

Note that `invokedynamic` is used at the site of lambda generation, not lambda invocation. The latter (in the final line of Figure 2) is a regular interface call.

From the perspective of a static analysis, the only method call that can be resolved in the `invokedynamic` instruction is the call to the metafactory but analysis of that cannot complete: the metafactory does load-time code generation. The compiler-metafactory synergy (of generating methods at compile time, yet leaving other code generation and call-site transformation to load time) is a design that cannot be penetrated by a conventional static analysis. When, in the next instruction, the static analysis tries to analyze the interface call on the object returned in the `invokedynamic` instruction, it cannot resolve the target method and analysis of this call fails.

2.3 Motivating Example 3: Method References

Java 8 introduced lambdas due to popular demand for the feature but also because they were needed for scaling stream processing over multicore hardware [17]. Streams were another new functional feature added to Java, that supported combinator functions over series of data ("streams"), enabling function composition and higher-order programming idioms. An example of streams and lambdas is the following snippet from Urma's streams tutorial [74]:

```
List<Integer> transactionsIds = transactions
  .stream()
  .filter(t -> t.getType() == Transaction.GROCERY)
  .sorted(comparing(Transaction::getValue).reversed())
  .map(Transaction::getId)
  .collect(toList());
```

Here, we see that function `filter` takes a lambda using the arrow syntax. We also see another higher-order feature added in Java 8: method references, such as `Transaction::getValue` and `Transaction::getId`. These pass regular methods as function parameters to combinator functions `comparing` and `map`.

While the syntax of method references is different compared to lambdas, these expressions are also implemented by the lambda metafactory in a similar way. Method references may be a simplified version of lambdas but they still have semantic complexities as they can capture a value from the environment for their receiver.

2.4 Motivating Example 4: SAM Conversion

The use of lambdas (and, by extension, `invokedynamic`) in Java is not limited to pure functional programming patterns. Lambdas are backwards compatible with pre-Java-8 code. In the following example, we see two `Runnable` objects being constructed, both with the same functionality:

```
public class Main {
  public static void main(String[] args) {

    // Use anonymous class.
    Runnable a = new Runnable() {
                   public void run() {
                     System.out.println("Hello.");
                   }
                 };
    a.run();

    // Use a lambda.
    Runnable b = (() -> System.out.println("Hello."));
    b.run();

  }
}
```

The `Runnable` interface is a standard type of the Java platform that happens to have a single method. It is, thus, a "single abstract method" ("SAM") type[3] and the lambda syntax can be used to generate an instance of it, which can be passed to code compiled with older Java versions. This approach makes pre-Java-8 code "forward-compatible to lambdas" [17] by viewing all existing single-method interfaces as lambdas ("SAM conversion" [14, 49]). In practice, this ease of constructing many types as lambdas means that, even in a simple "hello world" Java program, several `invokedynamic` calls to the lambda metafactory take place.

[3] Or a "functional interface" [19].

This has caused a regression in the power of static analysis tools on bytecode: *unless it supports lambdas, an analysis may find fewer facts for the same program under Java 8, compared to Java 7.* Java has become more dynamic and functional under the hood.

3 Technical Background

This section gives a basic background on the technology behind method handles, the `invokedynamic` framework, lambdas, and method references. We show as much as needed for the needs of the model of the static analysis that will follow.

3.1 Method Handles and Method Types

Two important kinds of values that are used in the rest of this section are method handles and method types.

Method handles are the equivalent of type-safe function pointers [64] and a lightweight alternative to standard reflective method objects [41]. They represent targets for invocation that can point to methods, constructors, fields, or other parts of an object [64]. There are three basic kinds of method handles: direct method handles are very similar to pointers; bound method handles are partial applications of methods [46, 63], and adapter method handles perform various adjustments of method parameters (e.g., from a flat list of arguments to a single argument array) [63].

Method types are type descriptors that help method handle invocations guarantee run-time type safety. A method type describes the parameter types and the return type that a method handle can accept. Method types can be modified to produce new method types: for example, their return type can be changed and types can be dropped, changed, or appended [54].

A method handle can be invoked via two methods called on it:

- `invokeExact()` calls the method handle directly, matching its types against the handle method type.
- `invoke()` is more permissive: it permits conversions of arguments and return type during the method handle invocation. Such conversions must be compatible with appropriate conversions of its method type [52].

The general `java.lang.invoke` API [47], offers ways to compose method handles, convert, fill in, or rearrange their arguments, perform conditional logic on them, or manipulate them in other ways. In practice, the method handles API is an embedded domain-specific language (DSL), which has the flavor of a combinatorial language over functional types. This DSL does deep embedding [69], i.e. the API creates an intermediate representation that reflects the semantics of the intended method handle.

The method handles are translated to an intermediate representation called *lambda forms* [48, 27]. (Not to be confused with the synonymous “lambda” high-level functional language feature of the language that we discuss extensively in this paper.) The lambda form representations can be cached and reused, interpreted, or compiled (using Just-in-Time technology). *This aspect of method handles argues for a static analysis to model them as primitive concepts: since they eventually do dynamic code generation, their semantics are impenetrable to a conventional static analysis.*

The compilation of lambda forms creates dynamically-generated bytecode of a special form, called *anonymous classes* [61]. This is bytecode that is not even visible to the runtime system class dictionary and is used for fast lightweight code generation [41]. Not only are these classes hidden; they also violate the read-only invariant of loaded classes in the VM, as they can patch other classes on the fly.

This design introduces a complete embedded mini-language on top of bytecode, together with a small implementation (intermediate representation, interpreter, and compilation back-end). For static analysis tools to reason about custom dynamic behavior, they must, thus, reason about this small language, from its front-end API embedding, through the implementation, to the generated bytecode.

3.2 The `invokedynamic` Instruction

The JVM was initially used to implement only the Java language. As the virtual machine became a state-of-the art optimizing Just-in-Time (JIT) compiler and the underlying platform grew, other statically-typed object-oriented languages (such as Scala [45] and Fortress [1]) chose to reuse it by having a compiler front-end from their syntax to bytecode. At the same time, the rise of dynamic languages, combined with the desire of their implementers to reuse the Java platform, led to a proliferation of dynamic languages implemented on top of the JVM, both existing ones such as Ruby (JRuby [44]), Python (Jython [56]), and JavaScript (Rhino/Nashorn [5]), and new ones such as Groovy and BeanShell. In the meantime, functional features entered the mainstream, influencing the object-oriented programming paradigm; functional languages gained enough traction to warrant implementations on top of the JVM. Examples of functional languages on the JVM are Clojure [24], the Haskell-inspired Eta [73] and Frege [76], and the Erjang version of Erlang [72]. Finally, Java itself had to evolve and incorporate functional features (we describe them in detail in Section 3.3).

To become multi-lingual in an efficient way, the JVM design had to gain two new powers: the capability to implement diverse dynamic behaviors; and native support for the basic building block of functional programming, lambdas. In this subsection, we give an overview of `invokedynamic`, while on the next subsection, we will see how the functional features are supported under the hood as an instance of dynamic behavior (Section 3.3).

A classic characteristic of dynamically-typed languages is their reliance on runtime optimization for performance, since there are no statically-available types to use for optimization. Naive implementations of dynamically-typed languages are slow, since they are usually interpreters that constantly query metadata to discover the runtime types of objects in order to perform safe operations on them. Runtime optimization systems come to the rescue: modern high-performance dynamic languages profile the running program and optimize it, often generating good code at runtime, when more information is known about the behavior of the program (the “Just-in-Time” or “JIT” approach).

JIT optimization has a long history, for instance one of its techniques to speed up method calls, “inline caching”, appears in the classic implementation of the Smalltalk object-oriented dynamic language [8]. Today, the JIT approach forms the basic technology behind successful implementations as diverse as the cutting-edge Java Virtual Machine [33] or the browser runtimes of JavaScript that enabled the Web 2.0 wave of applications.

As dynamic languages on the JVM were pushing for more performance on the JVM, Java 7 introduced a new bytecode opcode, `invokedynamic` [62], together with an API around it, that could offer the programmer the capability to completely customize dynamic program behavior. The program could now implement its own method dispatch semantics, for example perform linking, unlinking, and relinking of code on the fly, add or remove fields and methods in objects, or implement inline caching using plain Java code. The crucial advantages of this approach, compared to writing adapter code by hand, are not only in saving engineering effort through a friendly API, but also in informing the JIT optimizer so that better optimizations (such as inlining) can happen across dynamic dispatch borders.

Oracle offers this as motivation: *“The invokedynamic instruction simplifies and potentially improves implementations of compilers and runtime systems for dynamic languages on the*

JVM. The invokedynamic instruction does this by allowing the language implementer to define custom linkage behavior. This contrasts with other JVM instructions such as invokevirtual, in which linkage behavior specific to Java classes and interfaces is hard-wired by the JVM.”[4]

Dynamic languages on the JVM were naturally the first users of this new functionality (JRuby [40, 77], Jython [3], the Nashorn JavaScript engine [31], Groovy [71], Redline Smalltalk [43], and a significant subset of PHP [12]), as they could improve their performance [55]. The `invokedynamic` instruction even inspired the creation of at least one new JVM-based language [58]. Moreover, this new capability was used for other applications, such as live code modification [59], aspect-oriented programming [39], context-oriented programming [2, 34], multiple dispatch/multi-methods (a generalization of object-oriented dynamic dispatch to take more than one method arguments into consideration when choosing the target method of an invocation) [42], lazy computations [42, 15], generics specialization [20], implementation of actors [38], and dynamically adaptable binary compatibility via cross-component dynamic linking [28]. This new low-level functionality also became available for programmable high-level dynamic linking and metaobject protocol implementation via the Dynalink library [70].

Informally, `invokedynamic` can be seen as configurable initialization (and possible reconfiguration) of invocations in Java bytecode. When the JVM loads a class, it resolves every `invokedynamic` instruction in it. For every `invokedynamic` instruction:

1. A special *bootstrap method* is called. The method reads information either embedded in the instruction or coming from the constant pool of the class.
2. The bootstrap method returns a *call site* object. That object belongs to the instruction location in the bytecode and contains a method handle.
3. Since the call site contains a method handle, the invocation is resolved now and the call site has been linked. The method handle can be thus invoked (see Section 3.1).
4. The call site is a Java object, so the program can access it and can later mutate its method handle so that the invocation is effectively re-linked to resolve to another method. This is essential for modeling fully dynamic behavior (e.g., making an object support an extra method during run time).

The model above means that the program can now control the linking of method calls. Moreover, this framework makes dynamically-linked invocations efficient. Since the JVM internally supports `invokedynamic`, it can optimize such invocations. For example, if the call site is a constant call site,[5] the invocation can be inlined. The efficiency of `invokedynamic` invocations has been confirmed by Kaewkasi [29] and Ortin et al. [55].

3.3 Method References and Lambdas

As seen in the examples of Section 2, method references and lambdas are functional programming features added to Java for more expressive power. Eventually, Java 8 implemented these two features with `invokedynamic` [15]. A crucial motivation for this implementation choice has been compatibility, i.e., to avoiding a commitment to a single bytecode-visible implementation of lambdas (e.g., as classes). Describing the implementation of lambdas in terms of `invokedynamic` gives the Java compiler developers the freedom to later change the

[4] https://docs.oracle.com/javase/7/docs/technotes/guides/vm/multiple-language-support.html#invokedynamic

[5] https://docs.oracle.com/javase/8/docs/api/java/lang/invoke/ConstantCallSite.html

underlying implementation, without breaking binary compatibility [15, 17]. The only trace of the translation of lambdas inside the bytecode is an `invokedynamic` call to a specific lambda metafactory, but the code emitted by that may later change.

Both lambdas and method references use the same implementation technique: `invokedynamic` sites that use special bootstrap methods, the lambda metafactories [50]. A lambda metafactory initializes a call site so that it contains a lambda factory, i.e., it can generate functional objects. The Java 8 lambda metafactory generates an inner class that implements the functional interface.[6] The functional objects created can either be stateless or access values from their enclosing environment [18]. The implementation of lambdas is a thin layer of code that only uses small pieces of dynamically-generated code as glue.

In practice, the Java compiler creates appropriate methods for the bodies of lambdas (*implementing methods*) and registers method handles of them in the constant pool. These method handles are then used in `invokedynamic` invocations to the lambda metafactories, together with any values captured from the environment. The lambda metafactories can then create new anonymous classes that can be used to instantiate the functional objects and forward method calls to the implementing methods.

4 Static Analysis

We next present our model for handling the `java.lang.invoke` API (i.e., method handles), the `invokedynamic` instruction (in general), as well as Java lambdas. We offer a declarative set of inference rules that appeal to relations defined and used by an underlying value-flow/points-to static analysis. Our implementation is on the declarative Doop framework [6], so it is to a great extent isomorphic to the analysis model presented.

The essence of our analysis approach is threefold:

- Our baseline model gives semantics to method handles. (This is also the main novelty of our approach: the deep modeling of the `java.lang.invoke` API at its most fundamental level.) This requires appealing to an existing value flow analysis, since method handles have no hard-coded signatures in the bytecode: they offer `invoke` operations that are "signature-polymorphic". Therefore, any resolution of method handles requires a static model of all possible signature arguments to `invoke` instructions. Modeling the semantics of method handles is necessary since their implementation is un-analyzable, relying on run-time code generation (via the aforementioned "lambda forms"). Furthermore, this model requires static analysis of Java reflection, since method handles can also be looked up via reflection operations (e.g., by method types generated via reflective class values, or by "unreflecting" method objects into method handles).
- Based on the modeling of method handles, we straightforwardly model `invokedynamic` as an invocation of a method handle computed by a bootstrap method.
- Reasoning about lambdas appeals to a part of the `invokedynamic` reasoning. However, modeling lambdas both requires extra reasoning (because of dynamic code generation) and can avoid the need for expensive reflection analysis, since the method handles computed for lambdas do not employ reflection.

[6] `https://bugs.openjdk.java.net/browse/JDK-8000806`

4.1 Model Basics

We assume the following domains and (meta)variables, also listing some simple convenience predicates along the way:

- $s \in S$ are strings.
- $n, k \in \mathcal{N}$ are numbers.
- The symbol $*$ denotes arguments that can be ignored.
- $v \in V$ are variables, $\mathit{val} \in \mathit{Val}$ are values, $\lambda \in \mathit{Val}$ are functional objects.
- $t \in T$ are types while $t^i \in T^I \subset T$ are interface types. Constructor $\mathit{mock}_c(t, i)$ creates a mock object of type t that corresponds to an instruction i. The Class metaobject of a type t is given by $\mathit{Reified}_C(t)$ and is a value.
- $m \in M$ are methods. The formal of m at position n is represented as F_n^m. The special "this" variable of an instance method m is represented as $m/\texttt{this}$. The Method metaobject of a method m is given by $\mathit{Reified}_C(m)$ and is a value. We use the following predicates:
  - $\mathit{Constr}(m)$: m is a constructor method.
  - $\mathit{Static}(m)$: m is a static method.
  - $m \in t$: m is declared in type t.
  - The return variable v of m is represented as $\mathit{RetVar}(v, m)$.
- $m_t = \{t, [t_0, \ldots, t_{n-1}]\} \in M^T, n \geq 0$ are method types, which are pairs of a return type t and a (possibly empty) list of parameter types. Predicate $\mathit{AsType}(m_t^1, m_t^2)$ holds when method type m_t^2 has the same arity as m_t^1, and for every pair t, t' of m_t^1 and m_t^2 (at the same position), it holds that the two types are compatible: $t \leftrightarrows t'$. ($t \leftrightarrows t'$ is one of the analysis's main input predicates from Figure 3.) This type compatibility represents the `asType` rules of the specification [52]. Function $\mathit{MethodMT}(m)$ maps a method m to its method type.
- $i \in I$ are invocation instructions. Predicate $i \in m$ means that instruction i belongs to method m. The actual parameter that is passed at invocation i in position n is represented as A_n^i. For `invokedynamic` instructions, these are the non-bootstrap parameters of the bytecode instruction. If instruction i returns a value, $\mathit{Ret}(i)$ is the variable that will hold the returned value.
- $h \in \mathit{MH}$ are method handles. A method handle h has the form $\langle m, m_t \rangle$, which is a pair of a method m and a method type m_t. We also assume predicate $\mathit{DMHLookup}(t, s, m_t)$, which returns the direct method handle that corresponds to a method with name s, declared in type t, with method type m_t. Constructor $\mathit{mock}_h(t, h)$ creates a mock object of type t that corresponds to method handle h.
- $c \in C$ are call site identifiers. (These are different from mere instructions: because of the dynamic nature of calls, the same instruction can play the role of distinct call sites.)
- We assume lookup objects $\mathcal{L}_t$, one for each type t. These are opaque objects in the `java.lang.invoke` API that are used as intermediate values in a lookup: to retrieve, e.g., a method handle, first one retrieves a lookup object over a type, and subsequently uses it with method-identifying information.[7]

The table in Figure 3 lists the main relations that will be used in the analysis rules (i.e., all relations other than convenience predicates described earlier). We annotate each relation with IN if it is consumed by our rules and OUT if our rules inform it. Relation $v \mapsto \mathit{val}$

[7] Maintaining a distinct lookup object for each type also shows that our technique can potentially track access restrictions per type, as mandated by the specification of method lookup objects: `https://docs.oracle.com/javase/8/docs/api/java/lang/invoke/MethodHandles.Lookup.html`.

Relation	Description	Use
$v \mapsto val$	Variable v points to val.	IN, OUT
$f \mapsto val$	Field f points to val.	IN
$v[*] \mapsto val$	Variable v is an array and $v[i]$ points to val for some i.	OUT
$i \xrightarrow{h} m$	Instruction i calls method m using method handle h.	OUT
$i \xrightarrow{\lambda} m$	Instruction i calls method m using functional object λ.	OUT
$t \leftrightarrows t'$	Types t and t' are either subtypes of each other or can be converted to each other via boxing or unboxing.	IN
$CSite(c, i, t)$	Instruction i creates call site c with dynamic return type t.	INTER
$CSite_C(c, h, m)$	Call site c contains meth. handle h pointing to method m.	INTER
$MetafactoryInvo(i, s, t^i)$	Lambda metafactory invocation at instruction i, with dynamic method name s and functional interface t^i.	INTER
$Lambda(\lambda, m, s, i)$	Functional object λ with implementing method m, dynamic method name s and `invokedynamic` source instruction i.	INTER
$Capture(i, n, val)$	Instruction i captures environment value val at position n.	INTER
$InstanceImpl(i, m, \lambda)$	Functional object λ, generated at instruction i, uses non-static method m as implementing method.	INTER

Figure 3 Analysis relations.

is both IN and OUT, since our analysis is mutually recursive with the existing points-to analysis. Relations annotated with INTER are intermediate relations used in the analysis, that may not be externalized.

4.2 Model: Method Types and Method Handles

We show how the analysis can understand the APIs of method types and method handles. This includes handling the *polymorphic signatures* of Java bytecode.

A fundamental problem in the static analysis of method handles is that they contain native code, for example their "invoke" methods that must be used to do the method call are native.

The basic relation in this model is $i \xrightarrow{h} m$ which is a call-graph edge from instruction i to method m annotated with a method handle h. This relation is both created by rules (that discover method handle invocations) and consumed by rules (that handle argument passing and value returns).

The method handle invocation rules are shown in Figure 4 while Figure 5 shows the rules that simulate part of the method handles API. For clarity, we omit packages from qualified types (e.g., we write `MethodHandle` instead of `java.lang.invoke.MethodHandle`).

The rules of Figure 4 are relatively straightforward, capturing regular calling semantics for method handle invocations, once a method handle value has been determined. Interesting elements include the mutual recursion with an existing points-to analysis, as well as the construction of new (mock) objects, per the API specification, when a method handle that corresponds to a constructor is invoked.

$$\frac{m_t = \mathit{MethodMT}(m)}{\langle m, m_t \rangle}\ \textsc{MHMethod}$$

$$\frac{i = v.\texttt{<MethodHandle.invokeExact>}(\ldots) \qquad v \mapsto h \qquad h = \langle m, * \rangle}{i \xrightarrow{h} m}\ \textsc{MHCGE}$$

$$\frac{i \xrightarrow{h} m \qquad A^i_n \mapsto \mathit{val}}{F^m_k \mapsto \mathit{val}}\ \textsc{MHArgs} \qquad \frac{i \xrightarrow{h} m \qquad \mathit{RetVar}(v, m) \qquad v \mapsto \mathit{val} \qquad \mathit{Ret}(i) = v'}{v' \mapsto \mathit{val}}\ \textsc{RetH}$$

$$\frac{i \xrightarrow{h} m \qquad \mathit{Constr}(m) \qquad \mathit{val} = \mathit{mock}_h(t, h) \qquad \mathit{Ret}(i) = v}{m/\texttt{this} \mapsto \mathit{val} \qquad v \mapsto \mathit{val}}\ \textsc{MHConstr}$$

Figure 4 Rules for handling method handle invocations.

Rule MHMethod. This rule creates a method handle h and a method type m_t for every method found in the program.

Rule MHCGE. This rule informs the method handles call graph relation that an invocation i calls method m using method handle h (notation: $i \xrightarrow{h} m$).

Rules RetH and MHArgs. These rules pass arguments and return parameters.

Rule MHConstr. For method handles that correspond to constructors, a mock value is constructed and both the `this` variable in it and the return value of the invocation point to this value.

The rules of Figure 5 are a bit more demanding, since they capture precisely the semantics of the `java.lang.invoke` API, including lookup objects, using reflection to retrieve method handles, and more.

Rule AsType. This rule models the `asType()` method of the `MethodHandle` API using predicate $\mathit{AsType}(m_t^1, m_t^2)$.

Rule MHLookup. This rule models the per-type lookup object needed to find method handles. The `lookup()` method modeled in this rule is caller-sensitive [53], thus the caller type t characterizes the returned lookup object and is available for future uses of the object.

Rule MHLookupC. This rule models the connection between a lookup object and its type (e.g., to be used in the code for accessibility checks).

Rule Unreflect. This rule models the API methods that bridge the Reflection API with the `java.lang.invoke` API. These methods convert reified methods/constructors to method handles.

Rule Find. This rule models the API methods that look up a virtual or static method via a lookup object, returning a method handle.

Rule MType. This rule models the two-argument method `methodType()` of class `MethodHandle`. The other overloaded versions of `methodType()` are modeled similarly. The rule needs access to reflection support, since it takes advantage of points-to information that points to reified Class objects.

Reflection Support. A useful subset of these rules does not need reflection support in the analysis. For some programs, method types and method handles may come from the constant

$$\frac{\begin{array}{c} i = v.\texttt{<MethodHandle.asType>(}v'\texttt{)} \qquad v \mapsto \langle m, m_t^1 \rangle \\ v' \mapsto m_t^2 \qquad \mathit{AsType}(m_t^1, m_t^2) \qquad \mathit{Ret}(i) = v'' \end{array}}{v'' \mapsto \langle m, m_t^2 \rangle}\ \textsc{AsType}$$

$$\frac{i = \texttt{<MethodHandles.lookup>()} \qquad i \in m \qquad m \in t \qquad \mathit{Ret}(i) = v}{v \mapsto \mathcal{L}_t}\ \textsc{MHLookup}$$

$$\frac{i = v.\texttt{<MethodHandles.Lookup.lookupClass>()} \qquad v \mapsto \mathcal{L}_t \qquad \mathit{Ret}(i) = v'}{v' \mapsto \mathit{Reified}_C(t)}\ \textsc{MHLookupC}$$

$$\frac{\begin{array}{c} \mathit{Ret}(i) = v' \qquad \mathit{MethodMT}(m) = m_t \qquad i = \texttt{<MethodHandles.Lookup.}s\texttt{>(}v\texttt{)} \\ v \mapsto \mathit{Reified}_M(m) \qquad s \in \{\texttt{unreflect}, \texttt{unreflectSpecial}, \texttt{unreflectConstructor}\} \end{array}}{v' \mapsto \langle m, m_t \rangle}\ \textsc{Unreflect}$$

$$\frac{\begin{array}{c} i = v.\texttt{<MethodHandles.Lookup.}s\texttt{>(}v_0\texttt{, } v_1\texttt{, } v_2\texttt{)} \qquad s \in \{\texttt{findVirtual}, \texttt{findStatic}\} \qquad v \mapsto \mathcal{L}_t \\ \mathit{Ret}(i) = v' \qquad v_0 \mapsto \mathit{Reified}_C(t') \qquad v_1 \mapsto s \qquad v_2 \mapsto m_t \qquad \mathit{DMHLookup}(t', s, m_t) = h \end{array}}{v' \mapsto h}\ \textsc{Find}$$

$$\frac{\begin{array}{c} i = v.\texttt{<MethodType.methodType>(}v_0\texttt{, } v_1\texttt{)} \\ v_0 \mapsto \mathit{Reified}_C(t_0) \qquad v_1 \mapsto \mathit{Reified}_C(t_1) \qquad \mathit{Ret}(i) = v \end{array}}{v \mapsto \{t_0, [t_1]\}}\ \textsc{MType}$$

Figure 5 Rules for handling part of the method handles API.

pool instead of being looked up by the `java.lang.invoke` API; for such code, our rules do not require reflection support.

`invoke()` vs. `invokeExact()`. As mentioned in Section 3.1, the method handle API offers two different ways to invoke a method handle. The most fundamental is `invokeExact()`, which assumes the arguments and the return value have types that exactly match the method type of the method handle. In contrast, `invoke()` permits conversions in arguments and return values, as if the method handle could successfully change its method type via the `asType()` method. For presentation purposes, we only show the rules for `invokeExact` in Figure 4 and the rules for `asType()` in Figure 5. The handling of `invoke()` follows directly from these rules, accounting for autoboxing in the case of primitive conversions. The handling of `invokedynamic` (shown in Section 4.3) is not affected, since that only needs the functionality of invocations via `invokeExact` [33].

Generalized method handles. Method handles are also able to represent fields; we don't model this behavior here since it is not important for the `invokedynamic` analysis (that follows in the next section) but it is a simple extension of our model.

4.3 Generic Handling of `invokedynamic`

We next discuss the static modeling of `invokedynamic` instructions. The model effects the dynamic linking that eventually computes a method handle and invokes it. The key concept employed is call sites ($c \in C$). These are the return objects of `invokedynamic` bootstrap methods (as determined by regular points-to analysis) and internally use method handles to determine the calling behavior.

Our rules model the invokedynamic framework in order to discover the method handles contained in each call site. When a method handle h that maps to a method m is discovered to be contained in the call site of instruction i, a new call-graph edge $i \xrightarrow{h} m$ is created and the rules of the previous section analyze the method handle invocation. The rules for handling invokedynamic invocations are shown in Figure 6. Evaluation-wise, these rules *precede* the earlier rules that give semantics to method handles: The purpose of the rules in Figure 6 is to express what an invokedynamic does in terms of method handles, so that the earlier reasoning can take over.

We extend the earlier domains and predicates with:

- $I^d \subset I$ are invokedynamic instructions. Predicate $i \dashrightarrow_b m$ holds when an invokedynamic instruction i calls bootstrap method m.[8] We also assume the following invokedynamic projections:
  - $Boot : I^d \to M$ returns the bootstrap method.
  - $B^p : I^d \to n \to V$ returns the bootstrap parameter at position n.
  - $Dyn : I^d \to (S \times M^T)$ returns the dynamic method name / method type pair.

The rules are explained below:

Rules Bargs, Bargs0, and BargsV. The first rule passes arguments to the boot method, shifted by three positions, since the first three arguments are filled in by the JVM (and handled by rule Bargs0). Boot methods such as the alt metafactory may also take varargs that require special handling by the JVM, thus we also have rule BargsV. Note the introduction of an artificial (mock) array object to maintain the vararg values.

Rule RetB. This is the standard rule that returns value from a method call. It is adapted here for completeness, for the case of bootstrap method invocations.

Rule CSite. This rule stores information about a call site object computed at an invokedynamic instruction.

Rules CSite1 and CSite2. These two rules relate a call site object with its method handle and the method it points to.

Rule MHCGE$_{\mathsf{Dyn}}$. This rule relates the invokedynamic call site and its method handle to create call-graph edges with method handle semantics. From this point on, the rules in the previous section take over and complete the method handle invocation.

Reflection Support. The rules presented in the subsection do not require reflection. For example, a program which contains an invokedynamic instruction that passes a method handle constant (read from the class constant pool) to its bootstrap method, can be analyzed without reflection support. In practice, however, bootstrap methods often employ reflective reasoning to compute the method handle that will be returned in the call site return value, and thus reflection support should be provided.

4.4 Model: Method References and Lambdas

Both method reference expression and lambdas are implemented by the same machinery, a "lambda metafactory" [50]. At a very high level, the metafactory takes two arguments,

[8] We assume that all invokedynamic instructions call their bootstrap methods when their containing type is loaded.

$$\frac{i \dashrightarrow_b m \qquad \mathit{Dyn}(i) = \langle s, m_t \rangle}{F_0^m \mapsto \mathcal{L}_t \qquad F_1^m \mapsto s \qquad F_2^m \mapsto m_t}\ \textsc{BArgs0} \qquad \frac{i \dashrightarrow_b m \qquad B^p(i,n) \mapsto \mathit{val}}{F_{n+3}^m \mapsto \mathit{val}}\ \textsc{BArgs}$$

$$\frac{i \dashrightarrow_b m \qquad \mathit{val}' = \mathit{mock}_c(\texttt{java.lang.Object[]}, i) \qquad B^p(i,n) \mapsto \mathit{val} \qquad n > 2}{F_3^m \mapsto a \qquad \mathit{val}'[*] \mapsto \mathit{val}}\ \textsc{BArgsV}$$

$$\frac{i \dashrightarrow_b m \qquad \mathit{RetVar}(v,m) \qquad v \mapsto \mathit{val} \qquad \mathit{Ret}(i) = v'}{v' \mapsto \mathit{val}}\ \textsc{RetB}$$

$$\frac{\mathit{Dyn}(i) = \langle *, m_t \rangle \qquad m_t = \{t, *\} \qquad \mathit{Boot}(i) = m \qquad \mathit{RetVar}(v,m) \qquad v \mapsto c}{\mathit{CSite}(c,i,t)}\ \textsc{CSite}$$

$$\frac{\mathit{CSite}(c,*,t) \qquad c.\texttt{target} \mapsto h \qquad h = \langle m, \{t,*\} \rangle}{\mathit{CSite}_C(c,h,m)}\ \textsc{CSite1}$$

$$\frac{\mathit{CSite}(c,*,t) \qquad c.\texttt{target} \mapsto h \qquad h = \langle m, * \rangle \qquad \mathit{Constr}(m) \qquad m \in t}{\mathit{CSite}_C(c,h,m)}\ \textsc{CSite2}$$

$$\frac{\mathit{CSite}(c,i,t) \qquad \mathit{CSite}_C(c,h,m) \qquad h = \langle *, \{t,*\} \rangle}{i \xrightarrow{h} m}\ \textsc{MHCGE}_{\textsc{Dyn}}$$

Figure 6 Rules for generic handling of `invokedynamic`.

(1) a method handle pointing to a method m and (2) a SAM type t, and returns a functional object implementing t whose (single) method calls m.

The functional object may be an instance of a new dynamically-generated class, thus a naive points-to analysis cannot penetrate the object to analyze calls on it. Our analysis understands the semantics of the functional objects created by the dynamic linking and method resolution of the metafactory, and creates a mock value in place of the functional object. That value can be propagated in the program as usual by the underlying points-to analysis. Appropriate metadata on the value help the analysis compute intended semantics such as the invocation target or the captured values of the environment.

The Three Phases of `invokedynamic` for Lambdas. When used for lambdas, functional object creation by the lambda metafactory works in three phases [50]:

1. **Linkage.** The bootstrap method is called and a call site object is returned, at the location of the `invokedynamic` instruction. The bootstrap method being the “metafactory”, the call site is then a “lambda factory”, which must be invoked to produce a functional object.
2. **Capture.** The method handle in the call site object is invoked, possibly with some arguments. This permits different behavior for different contexts by capturing values of the enclosing environment. The result is the functional object.
3. **Invocation.** The functional object can then be passed around in the code and the method of its functional interface can be eventually called.

The rules that enable analysis of method references are shown in Figure 7. The basic idea is to create mock values in the analysis for functional objects and simulate all three phases so that calls are correctly resolved. We assume the following domains, (meta)variables, and predicates:

- The L^λ constant stands for the lambda metafactory [50] of the OpenJDK.

- $\lambda \in \mathit{Val}$ ranges over functional objects.
- $\#i$ returns the arity of instruction i (the number of actual parameters passed to the functional object).

The rules are explained below:

Rule Metafactory. This rule marks an `invokedynamic` invocation as a lambda metafactory invocation.

Rule Lambda. This rule creates the mock functional object λ that will propagate in the program and behave (in the analysis) as if it was an object created by the metafactory. The object keeps related metadata in relation $\mathit{Lambda}(\lambda, m, s, i)$: its implementing method m (found in a constant method handle argument of the metafactory), the name of the functional interface method it implements, and the `invokedynamic` instruction i that created the functional object.

Rule Capture. This rule records possibly captured values from the enclosing environment. (All arguments are eagerly recorded as possible captured values and the appropriate capture arguments are recognized in later rules CaptArgsand LambdaThis.)

Rule CGE$_\mathsf{L}$. This rule creates call-graph edges to the actual implementing method of the functional object. Following these edges bypasses the dynamically-generated classes and lets the static analysis discover the code of method references and lambdas.

Rule Ret$_\mathsf{L}$. This is the standard rule for return values from methods.

Rule InstImpl. This rule records that a functional object is implemented by a non-static method. This means that further rules should discover the receiver and pass it to the method.

Rules Shift1, Shift2, and Shift3. These rules populate relation $\mathit{Shift}(\lambda, m, n, k)$, which records if the arguments passed to the functional object must be shifted to make room for a receiver. This is because instance methods may implicitly consume one of the actual arguments of the `invokedynamic` or of the functional interface invocation, to use as the receiver. Static methods take all `invokedynamic`-actual arguments before the ones passed to the functional object during method invocation.

Rule LArgs. This passes arguments to the implementing method, from the method invocation on the functional object. The shifting of parameters addresses a number of patterns that the metafactory follows to capture and pass values from the environment.

Rule CaptArgs. This rule passes captured arguments to the implementing method.

Rule LambdaThis. This rule handles the pattern of captured receiver parameters.

Rule MRefThis. This rule handles the pattern where a method reference to an instance method has not captured a receiver, but will receive it during invocation as an extra argument.

Rule CCall. This handles the special case where a method reference points to a constructor (is thus a “constructor reference”). Since constructor methods are `void` and assume an already constructed (but not initialized) object, this rule creates such an object and binds it both to the ’this’ variable of the constructor and the return variable of the invocation.

Additional Features. The JDK also has a second metafactory, the “alt metafactory”: a generalization of the lambda metafactory that provides additional features, such as bridging, support for multiple interfaces, and serializability. We do not model such extra properties of its lambdas here, but these features are type-based so they are amenabe to handling in a similar way to the rules we already present.

Reflection Support. The method handles passed to the metafactory are statically known: either the programmer provided them as method references or the compiler generated them for lambdas. Thus our rules for handling lambdas and method references do not need reflection support; the only method handles used come from the constant pool. This means that our approach can integrate with the baseline configuration of a points-to analysis, in order to analyze programs without overhead due to reflection support.

Context sensitivity. The analysis, as presented, has a context-insensitive formulation, to avoid unnecessary complication of the rules. Careful (but conceptually standard) addition of context elements to predicates (as shown, e.g., in reference [66]) produces a context-sensitive version. Our implementation is fully context sensitive.

5 Evaluation

We evaluate our analysis on two test suites: a microbenchmark suite of our own (Section 5.1) and the test suite of *Sui et al.* [68] (Section 5.2).

Our analysis is implemented in the declarative static analysis framework Doop [6]. All analyses are run on a 64-bit machine with an Intel Xeon CPU E5-2667 v2 3.30GHz with 256 GB of RAM. We use the Soufflé compiler (v.1.4.0), which compiles Datalog specifications into binaries via C++ and run the resulting binaries in parallel mode using four jobs. Doop uses the Java 8 platform as implemented in Oracle JDK v1.8.0_121. All running times and precision numbers are for Doop's default context-insensitive analysis. (Context sensitivity adds no precision to the high-level metrics shown.) For benchmarks of generalized `invokedynamic` features (i.e., not lambdas and method references), we enable reflection support in Doop.

5.1 Microbenchmark Suite

To evaluate our technique, we have built our own suite of microbenchmarks. These benchmarks capture a large number of idioms found in realistic uses of method references (Section 5.1.1), lambdas (Section 5.1.2), and method handles combined with `invokedynamic` (Section 5.1.3), including most of the patterns shown in the examples of Section 2. (Other patterns are captured in the Sui et al. suite, discussed later.) The suite is freely available.

Analysis times for the three component benchmarks are shown in Figure 8. As can be seen, enabling reflection analysis, for fully general handling of `invokedynamic`, incurs higher cost.

Our static analysis fully models all behavior in the microbenchmark suite. Although the suite was developed in tandem with the analysis, it still provides partial validation of analysis completeness, given the effort to encode many variations of operations, as detailed next.

5.1.1 Microbenchmark: Method References

This benchmark includes Oracle's tutorial code MethodReferencesTest [51]. We capture the behavior of all four kinds of methods references (found in the tutorial table): to static methods, to instance methods of a particular object, to instance methods of an arbitrary object of a particular type, and to constructors.

The microbenchmark also contains code that showcases the following features:

1. Construction of functional objects directly from method references.
2. Use of functional objects together with Java 8 stream API methods.

LINKAGE

$$\frac{i \dashrightarrow_b m \qquad m \in L^{\lambda} \qquad \mathit{Dyn}(i) = \langle s, m_t \rangle \qquad m_t = \{t^i, *\}}{\mathit{MetafactoryInvo}(i, s, t^i)}\ \textsc{Metafactory}$$

$$\frac{\mathit{MetafactoryInvo}(i, s, t^i) \qquad B^p(i, 1) \mapsto \langle m, * \rangle \qquad \mathit{Ret}(i) = v \qquad \lambda = \mathit{mock}_c(t^i, i)}{v \mapsto \lambda \qquad \mathit{Lambda}(\lambda, m, s, i)}\ \textsc{Lambda}$$

CAPTURE

$$\frac{\mathit{MetafactoryInvo}(i, *, *) \qquad A^i_n \mapsto \mathit{val}}{\mathit{Capture}(i, n, \mathit{val})}\ \textsc{Capture}$$

INVOCATION

$$\frac{\mathit{Lambda}(\lambda, m, s, *) \qquad v \mapsto \lambda \qquad i = v.\texttt{<}s\texttt{>}(\ldots)}{i \xrightarrow{\lambda} m}\ \mathrm{CGE}_{\mathrm{L}}$$

$$\frac{i \xrightarrow{\lambda} m \qquad R^m_n = v \qquad v \mapsto \mathit{val} \qquad \mathit{Ret}(i) = v'}{v' \mapsto \mathit{val}}\ \textsc{Ret}_{\mathrm{L}}$$

$$\frac{* \xrightarrow{\lambda} m \qquad \neg \mathit{Static}(m) \qquad \mathit{Lambda}(\lambda, *, *, i)}{\mathit{InstanceImpl}(i, m, \lambda)}\ \textsc{InstImpl}$$

$$\frac{\mathit{Lambda}(\lambda, m, *, *) \qquad \mathit{Static}(m)}{\mathit{Shift}(\lambda, m, 0, 0)}\ \textsc{Shift1}$$

$$\frac{\mathit{InstanceImpl}(i, m, \lambda) \qquad \#i = 0}{\mathit{Shift}(\lambda, m, 0, 1)}\ \textsc{Shift2} \qquad \frac{\mathit{InstanceImpl}(i, m, \lambda) \qquad \#i > 0}{\mathit{Shift}(\lambda, m, 1, 0)}\ \textsc{Shift3}$$

$$\frac{\begin{array}{l} i \xrightarrow{\lambda} m \qquad \mathit{Shift}(\lambda, m, k, n) \qquad \mathit{Lambda}(\lambda, m, *, i) \\ A^i_{n'} = v' \qquad F^m_{n''} = v \qquad n'' = \#i - (k + n) + n' \qquad v' \mapsto \mathit{val} \end{array}}{v \mapsto \mathit{val}}\ \textsc{LArgs}$$

$$\frac{* \xrightarrow{\lambda} m \qquad \mathit{Shift}(\lambda, m, k, *) \qquad \mathit{Lambda}(\lambda, m, *, i) \qquad \mathit{Capture}(i, n, \mathit{val}) \qquad k + n \leq \#i}{F^m_{n-k} \mapsto \mathit{val}}\ \textsc{CaptArgs}$$

$$\frac{\mathit{Shift}(\lambda, m, 1, 0) \qquad \mathit{InstanceImpl}(i, m, \lambda) \qquad \mathit{Capture}(i, 0, \mathit{val})}{m/\texttt{this} \mapsto \mathit{val}}\ \textsc{LambdaThis}$$

$$\frac{i \xrightarrow{\lambda} m \qquad \mathit{Shift}(\lambda, m, 0, 1) \qquad A^i_0 \mapsto \mathit{val}}{m/\texttt{this} \mapsto \mathit{val}}\ \textsc{MRefThis}$$

$$\frac{i \xrightarrow{\lambda} m \qquad \mathit{Constr}(m) \qquad m \in t \qquad \mathit{Ret}(i) = v \qquad \mathit{val} = \mathit{mock}_c(t, i)}{v \mapsto \mathit{val} \qquad m/\texttt{this} \mapsto \mathit{val}}\ \textsc{CCall}$$

Figure 7 Rules for handling method references and lambdas.

Benchmark	Time (sec)
Method References	27
Lambdas	23
Method Handles and `invokedynamic`	378

Figure 8 Microbenchmark times.

3. Auto-boxing conversions.

5.1.2 Microbenchmark: Lambdas

This benchmark shows the handling of the following features:

1. Creating lambdas with arrow notation. This includes nested lambdas.
2. Creating lambdas that can access values of the outside environment (forming closures).

5.1.3 Microbenchmark: Method Handles and `invokedynamic`

Java currently does not support the direct representation of `invokedynamic` in source code, although such a feature is considered for inclusion in future versions of the language [21]. For this reason, this benchmark uses the ASM bytecode manipulation library[9] to dynamically generate and load a class with `invokedynamic` invocations.

The benchmark captures the following patterns:

1. Lookup of a `MethodHandles.Lookup` object via `MethodHandles.lookup()`.
2. Construction of method type values via `MethodType.methodType()` methods.
3. Look-up of virtual and static methods via `MethodHandles.Lookup.findVirtual()` and `MethodHandles.Lookup.findStatic()`.
4. Calling method handles with `MethodHandle.invokeExact()`.
5. Passing a receiver for non-static methods (thus handling places where the signature of the target method differs from the signature of the `MethodHandle.invokeExact()` signature found in the bytecode).
6. Bootstrapping calls to another class in a manner similar to the motivating example in Section 2.1.

5.2 Sui et al. Test Suite

We also evaluate our technique using the dynamic features test suite of *Sui et al.* [68]. This is a test suite that examines the soundness of call-graph construction and is written to specifically test the static analysis of features such as lambdas and `invokedynamic`, by authors with extensive experience in systematic Java benchmarking efforts (e.g., XCorpus [9]).

The benchmark suite contains three benchmarks for lambdas, plus a benchmark for `invokedynamic` in general (Dynamo). Dynamo is a realistic software artifact [28] that has been configured in the benchmark suite to specifically evaluate the analysis of `invokedynamic`. The Dynamo library exercises all features of dynamic invocation sites (static vs. non-static, constructors, signature adaptation, interaction with plain Java reflection). It injects `invokedynamic` calls in unsuspecting code to address cross-component linking errors. Thus, if these `invokedynamic` sites are not analyzed, then the static analysis cannot find calls from code to a library.

The Dynamo test program in the suite contains two `invokedynamic` sites:

[9] `https://asm.ow2.io/`

Benchmark	Reachable		Unreachable		Time (sec)
	expected	analysis	expected	analysis	
LambdaConsumer	1	✓	1	✓	21
LambdaFunction	1	✓	2	✓	21
LambdaSupplier	1	✓	1	✓	22
Dynamo	1	✓	1	–	242

Figure 9 Dynamic benchmark results.

1. A site that looks up a constructor method and creates an object. Since it is a constructor method handle, the analysis also recognizes that an object must also be allocated for this invocation.
2. A site that looks up an instance method and calls it. The original signature of the method accepts an object and is adapted to also accept the receiver.

In both cases, Dynamo retrieves the method via reflection and then proceeds to "unreflect" it. The test program does not test lookup of static methods.

For every benchmark, the following ground truth is provided: one or more methods are expected to be found reachable, while one or more different methods are expected to be found unreachable. The results of applying our analysis to these benchmarks are shown in Figure 9.

Notably:

- All lambda benchmarks are analyzed precisely: the expected methods are found reachable or unreachable.
- For Dynamo, our analysis over-approximates reachability. Dynamo uses `invokedynamic` as a layer between components to ensure binary compatibility with evolving code. As seen in Figure 9, our analysis over-approximates reachability: it discovers the expected method as reachable but also discovers the expected unreachable method. This problem is not fundamental to the technique that we present, but is caused by the lack of flow sensitivity in the underlying points-to analysis, provided by the Doop framework. Dynamo code creates method handles by gathering reflectively all members of classes and then selectively filtering out the ones that do not match; Doop's flow insensitivity causes it to ignore this filtering. Coupled with flow sensitivity, our technique should be able to ignore the expected unreachable method.
- The efficiency of a lambda-specialized analysis vs. a general-purpose `invokedynamic` analysis that requires reflection support is again demonstrated in the running times.

6 Related work

Static Analysis of Java Lambdas and Dynamic Calls. Some recent work has attempted to treat lambdas and their static analysis, mostly in isolation, as another high-level feature for practical tools. Cifuentes et al. [7] perform a pattern-based vulnerability analysis (i.e., not a full low-level analysis of value flow) and recognize code patterns containing lambdas. There has also been work on dynamic analyses that understand Java-style lambdas [11].

Reflection and programmable dynamic calls are subtle features that should be formalized in order to be addressed. However, the bibliography is lacking: we only know of the work of Landman et al. [30], who give a syntax of the DSL behind the standard Java Reflection API. They do not treat its semantics, as they did not need to (their work was on mining big codebases for the existence of specific patterns).

To the best of our knowledge, no formal semantic model of `invokedynamic` and its API exists. Other Java APIs that cannot be easily analyzed statically have also been candidates for static semantic modeling. Smaragdakis et al. model the reflection API [67] and Fourtounis et al. model dynamic proxies [13]. Our approach differs in two aspects: (a) we do not necessarily incur performance overheads (our handling of functional objects does not require expensive reflection support) and (b) we model the lower-level `java.lang.invoke` API, which requires handling of JVM features such as signature polymorphism, caller sensitivity, and reasoning about code running at class-loading time.

The IBM WALA static analysis framework [10] has limited support for `invokedynamic`, specifically for call-graph edges over lambdas by generating synthetic classes.[10] WALA also lacks full support for constructor method references [60].

Transforming Away `invokedynamic`. Lambdas are not easy to work with; Soot, a popular Java manipulation and analysis framework, even considers statically transforming them away [4], since `invokedynamic` has been too difficult to analyze: "Soot does not fully support dynamic invokes ... could not find an easy workaround and instead decided that it would be best to change Schaapi such that dynamic invokes (and thus lambdas) are ignored completely."[11]

Along the same lines, but more completely, the OPAL bytecode rectifier[12] removes instances of `invokedynamic` as used in Java lambdas. This is a general alternative static treatment of lambdas, but not of other instances of `invokedynamic`. Similar removal of stylized uses of `invokedynamic`, without handling the general case, are performed by RetroLambda[13] and Google's D8.[14] These tools cannot, e.g., make the Dynamo benchmark analyzable by analyses that do not understand `invokedynamic`.

Other Platforms. Apart from the popular OpenJDK and its VM, used on servers or desktops, the other mainstream Java platform is Android. The implementation of `invokedynamic` on Android posed some complications because dynamic code generation is restricted on Android due to resource constraints [57, 64]. `invokedynamic` was prototyped for Android [64] and, eventually, became officially supported when the latest "Android N" switched to Java 8. Our work is, thus, applicable to Android as well. Android is a platform that commands special attention due to its popularity. `invokedynamic` enables new optimizations and analyses [78, 79, 80]. However, the instruction is also a security threat, since it is so powerful that it can, for example, hide method calls and make malware undetectable (as demonstrated by the DexProtector tool [32] or the survey of Gorenc and Spelman [22]) and provides less security by-design compared to classic reflection [65].

The .NET platform also has functionality similar to method handles and anonymous classes, called "dynamic methods" [37]. We, thus, expect that our approach can be ported to other runtimes and to their implementation of dynamic features.

7 Conclusion

We presented a static analysis modeling of programmable dynamic linking in Java, i.e., the `invokedynamic` instruction and accompanying framework. The approach addresses the most

[10] https://groups.google.com/forum/#!topic/wala-sourceforge-net/omsGtp_ow7I, https://github.com/wala/WALA/blob/f2b1e9fec0627e221427404cb7ba194c4a89cd9e/com.ibm.wala.core/src/com/ibm/wala/ipa/summaries/LambdaSummaryClass.java#L42

[11] https://github.com/cafejojo/schaapi/pull/295

[12] http://www.opal-project.de/DeveloperTools.html

[13] https://github.com/luontola/retrolambda

[14] https://jakewharton.com/androids-java-8-support/

fundamental level of the language feature, fully modeling method handles, while at the same time it maintains high efficiency and completeness for common uses of `invokedynamic` in Java lambdas. This is the first thorough handling of the `invokedynamic` feature, which had so far resisted static analysis.

References

1 Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L Steele Jr, Sam Tobin-Hochstadt, Joao Dias, Carl Eastlund, et al. The Fortress language specification. *Sun Microsystems*, 139:140, 2005.

2 Malte Appeltauer, Michael Haupt, and Robert Hirschfeld. Layered Method Dispatch with INVOKEDYNAMIC: An Implementation Study. In *Proceedings of the 2nd International Workshop on Context-Oriented Programming*, COP '10, pages 4:1–4:6, New York, NY, USA, 2010. ACM. doi:10.1145/1930021.1930025.

3 Shashank Bharadwaj. Optimizing Jython using invokedynamic and Gradual Typing. Master's thesis, University of Colorado at Boulder, 2012.

4 Eric Bodden. Develop transformer that gets rid of indy calls for lambda capture #226, 2014. URL: https://github.com/Sable/soot/issues/226.

5 Norris Boyd et al. Rhino: Javascript for Java. *Mozilla Foundation*, 2007.

6 Martin Bravenboer and Yannis Smaragdakis. Strictly Declarative Specification of Sophisticated Points-to Analyses. In *Proceedings of the 24th Annual ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, OOPSLA '09, New York, NY, USA, 2009. ACM.

7 Cristina Cifuentes, Andrew Gross, and Nathan Keynes. Understanding Caller-sensitive Method Vulnerabilities: A Class of Access Control Vulnerabilities in the Java Platform. In *Proceedings of the 4th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, SOAP 2015, pages 7–12, New York, NY, USA, 2015. ACM. doi:10.1145/2771284.2771286.

8 L. Peter Deutsch and Allan M. Schiffman. Efficient Implementation of the Smalltalk-80 System. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '84, pages 297–302, New York, NY, USA, 1984. ACM. doi:10.1145/800017.800542.

9 Jens Dietrich, Henrik Schole, Li Sui, and Ewan D. Tempero. XCorpus - an executable corpus of Java programs. *Journal of Object Technology*, 16(4):1:1–24, 2017. doi:10.5381/jot.2017.16.4.a1.

10 Julian Dolby, Stephen J. Fink, and Manu Sridharan. T.J. Watson libraries for analysis (WALA). http://wala.sourceforge.net.

11 Sebastian Erdweg, Vlad Vergu, Mira Mezini, and Eelco Visser. Finding Bugs in Program Generators by Dynamic Analysis of Syntactic Language Constraints. In *Proceedings of the Companion Publication of the 13th International Conference on Modularity*, MODULARITY '14, pages 17–20, New York, NY, USA, 2014. ACM. doi:10.1145/2584469.2584474.

12 Rémi Forax. JSR 292 / PHP.reboot. https://www.lrde.epita.fr/dload/seminar/2010-12-08/forax.pdf, 2010.

13 George Fourtounis, George Kastrinis, and Yannis Smaragdakis. Static Analysis of Java Dynamic Proxies. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, pages 209–220, New York, NY, USA, 2018. ACM. doi:10.1145/3213846.3213864.

14 Brian Goetz. One VM, Many Languages. https://gotocon.com/dl/jaoo-aarhus-2010/slides/BrianGoetz_OneVMManyLanguages.pdf, 2010. GOTO Aarhus 2010 Conference.

15 Brian Goetz. From lambdas to bytecode. http://wiki.jvmlangsummit.com/images/1/1e/2011_Goetz_Lambda.pdf, 2011. JVM Language Summit.

16 Brian Goetz. Implementing lambda expressions in Java. http://wiki.jvmlangsummit.com/images/7/7b/Goetz-jvmls-lambda.pdf, 2012. JVM Language Summit.

17 Brian Goetz. Lambda: A peek under the hood. `https://www.slideshare.net/jaxlondon2012/lambda-a-peek-under-the-hood-brian-goetz`, 2012. JAX London 2012.

18 Brian Goetz. Translation of Lambda Expressions, April 2012. Accessed: June 11, 2019. URL: `http://cr.openjdk.java.net/~briangoetz/lambda/lambda-translation.html`.

19 Brian Goetz. State of the Lambda, September 2013. Accessed: June 11, 2019. URL: `http://cr.openjdk.java.net/~briangoetz/lambda/lambda-state-final.html`.

20 Brian Goetz. Project Valhalla Update. `https://www.oracle.com/technetwork/java/jvmls2016-goetz-3126134.pdf`, 2016. JVM Language Summit.

21 Brian Goetz. JEP 303: Intrinsics for the LDC and INVOKEDYNAMIC Instructions, 2018. URL: `https://openjdk.java.net/jeps/303`.

22 Brian Gorenc and Jasiel Spelman. Java Every-Days – Exploiting Software Running on 3 Billion Devices. `https://media.blackhat.com/us-13/US-13-Gorenc-Java-Every-Days-Exploiting-Software-Running-on-3-Billion-Devices-WP.pdf`. HP Security Research Zero Day Initiative.

23 James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. The Java Language Specification, Java SE 8 Edition (Java Series), 2014.

24 Stuart Halloway. *Programming Clojure*. Pragmatic Bookshelf, 1st edition, 2009.

25 Anders Hejlsberg, Scott Wiltamuth, and Peter Golde. *C# Language Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

26 Philipp Holzinger, Stefan Triller, Alexandre Bartel, and Eric Bodden. An In-Depth Study of More Than Ten Years of Java Exploitation. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 779–790, New York, NY, USA, 2016. ACM. `doi:10.1145/2976749.2978361`.

27 Vladimir Ivanov. Invokedynamic: Deep Dive. `http://cr.openjdk.java.net/~vlivanov/talks/2015-Indy_Deep_Dive.pdf`. Accessed: June 11, 2019.

28 Kamil Jezek and Jens Dietrich. Magic with Dynamo – Flexible Cross-Component Linking for Java with Invokedynamic. 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 12:1–12:25, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. `doi:10.4230/LIPIcs.ECOOP.2016.12`.

29 Chanwit Kaewkasi. Towards Performance Measurements for the Java Virtual Machine's Invokedynamic. In *Virtual Machines and Intermediate Languages*, VMIL '10, pages 3:1–3:6, New York, NY, USA, 2010. ACM. `doi:10.1145/1941054.1941057`.

30 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.

31 Jim Laskey. Adventures in JSR-292 or How To Be A Duck Without Really Trying. `http://wiki.jvmlangsummit.com/images/c/ce/Nashorn.pdf`, 2011. JVM Language Summit.

32 Licel. DexProtector. URL: `https://dexprotector.com/docs`.

33 Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java Virtual Machine Specification, Java SE 8 Edition*. Addison-Wesley Professional, 1st edition, 2014.

34 Baptiste Maingret, Frédéric Le Mouël, Julien Ponge, Nicolas Stouls, Jian Cao, and Yannick Loiseau. Towards a Decoupled Context-Oriented Programming Language for the Internet of Things. In *Proceedings of the 7th International Workshop on Context-Oriented Programming*, COP'15, pages 7:1–7:6, New York, NY, USA, 2015. ACM. `doi:10.1145/2786545.2786552`.

35 Davood Mazinanian, Ameya Ketkar, Nikolaos Tsantalis, and Danny Dig. Understanding the Use of Lambda Expressions in Java. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):85:1–85:31, October 2017. `doi:10.1145/3133909`.

36 Erik Meijer, Brian Beckman, and Gavin Bierman. LINQ: Reconciling object, relations and XML in the .NET framework. In *Proceedings of the 2006 ACM SIGMOD International*

Conference on Management of Data, SIGMOD '06, pages 706–706, New York, NY, USA, 2006. ACM. doi:10.1145/1142473.1142552.

37 Microsoft. DynamicMethod Class. URL: https://msdn.microsoft.com/en-us/library/system.reflection.emit.dynamicmethod(v=vs.110).aspx.

38 Behrooz Nobakht and Frank S. de Boer. Programming with Actors in Java 8. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Specialized Techniques and Applications: 6th International Symposium, ISoLA 2014, Imperial, Corfu, Greece, October 8-11, 2014, Proceedings, Part II*, pages 37–53. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014. doi:10.1007/978-3-662-45231-8_4.

39 S. Nopnipa and C. Kaewkasi. Aspect-aware bytecode combinators for a dynamic AOP system with invokedynamic. In *The 2013 10th International Joint Conference on Computer Science and Software Engineering (JCSSE)*, pages 246–251, May 2013. doi:10.1109/JCSSE.2013.6567353.

40 Charles Nutter. A First Taste of InvokeDynamic. URL: http://blog.headius.com/2008/09/first-taste-of-invokedynamic.html.

41 Charles Nutter. The Power of the JVM. URL: http://blog.headius.com/2008/05/power-of-jvm.html.

42 Charles Nutter. invokedynamic: You Ain't Seen Nothing Yet, 2012. Proceedings of the JAX Conference (JAX 12). URL: https://www.slideshare.net/CharlesNutter/jax-2012-invoke-dynamic-keynote.

43 Charles Nutter. GOTO Night with Charles Nutter Slides, 2014. GOTO 2014. URL: https://www.slideshare.net/AlexandraMasterson/goto-night-with-charles-nutter-slides.

44 Charles O. Nutter, Thomas Enebo, Nick Sieger, Ola Bini, and Ian Dees. *Using JRuby: Bringing Ruby to Java.* Pragmatic Bookshelf, 1st edition, 2011.

45 Martin Odersky and Tiark Rompf. Unifying Functional and Object-oriented Programming with Scala. *Communications of the ACM*, 57(4):76–86, April 2014. doi:10.1145/2591013.

46 OpenJDK Compiler Team. Bound method handles - HotSpot - OpenJDK Wiki. URL: https://wiki.openjdk.java.net/display/HotSpot/Bound+method+handles.

47 Oracle. java.lang.invoke (Java Platform SE 7). URL: https://docs.oracle.com/javase/7/docs/api/java/lang/invoke/package-summary.html.

48 Oracle. JEP 160: Lambda-form representation for method handles. URL: http://openjdk.java.net/jeps/160.

49 Oracle. JSR 335: Lambda Expressions for the Java™ Programming Language. URL: https://jcp.org/en/jsr/detail?id=335.

50 Oracle. LambdaMetafactory (Java Platform SE 8). URL: https://docs.oracle.com/javase/8/docs/api/java/lang/invoke/LambdaMetafactory.html.

51 Oracle. Method References (The Java™ Tutorials > Learning the Java Language > Classes and Objects), 2017. URL: https://docs.oracle.com/javase/tutorial/java/javaOO/methodreferences.html.

52 Oracle. MethodHandle (Java Platform SE 8), 2018. URL: https://docs.oracle.com/javase/8/docs/api/java/lang/invoke/MethodHandle.html.

53 Oracle. MethodHandles (Java Platform SE 8), 2018. URL: https://docs.oracle.com/javase/8/docs/api/java/lang/invoke/MethodHandles.html.

54 Oracle. MethodType (Java Platform SE 8), 2018. URL: https://docs.oracle.com/javase/8/docs/api/java/lang/invoke/MethodType.html.

55 F. Ortin, P. Conde, D. Fernandez-Lanvin, and R. Izquierdo. The Runtime Performance of invokedynamic: An Evaluation with a Java Library. *IEEE Software*, 31(4):82–90, July 2014. doi:10.1109/MS.2013.46.

56 Samuele Pedroni and Noel Rappin. *Jython Essentials: Rapid Scripting in Java.* O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1 edition, 2002.

57 Jerome Pilliet, Remi Forax, and Gilles Roussel. DualStack: Improvement of invokedynamic implementation on Android. In *Proceedings of the 13th International Workshop on Java*

Technologies for Real-time and Embedded Systems, JTRES '15, pages 4:1–4:8, New York, NY, USA, 2015. ACM. doi:10.1145/2822304.2822310.

58 Julien Ponge, Frédéric Le Mouël, and Nicolas Stouls. Golo, a Dynamic, Light and Efficient Language for Post-invokedynamic JVM. In *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, PPPJ '13, pages 153–158, New York, NY, USA, 2013. ACM. doi:10.1145/2500828.2500844.

59 Julien Ponge and Frédéric Le Mouël. JooFlux: Hijacking Java 7 invokedynamic to support live code modifications. *CoRR*, abs/1210.1039, 2012. arXiv:1210.1039.

60 M. Reif, F. Kübler, M. Eichberg, and M. Mezini. Systematic evaluation of the unsoundness of call graph construction algorithms for Java. In *Proceedings of SOAP 2018*. ACM, 2018.

61 John R. Rose. Anonymous classes in the VM, January 2008. URL: https://blogs.oracle.com/jrose/entry/anonymous_classes_in_the_vm.

62 John R. Rose. Bytecodes Meet Combinators: Invokedynamic on the JVM. In *Proceedings of the Third Workshop on Virtual Machines and Intermediate Languages*, VMIL '09, pages 2:1–2:11, New York, NY, USA, 2009. ACM. doi:10.1145/1711506.1711508.

63 John R. Rose. Method Handles and Beyond... Some basis vectors. http://wiki.jvmlangsummit.com/images/8/88/Rose-2011-FutureDirections.pdf, 2011. JVM Summit.

64 Gilles Roussel, Remi Forax, and Jerome Pilliet. Android 292: Implementing Invokedynamic in Android. In *Proceedings of the 12th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '14, pages 76:76–76:86, New York, NY, USA, 2014. ACM. doi:10.1145/2661020.2661032.

65 Security Explorations. Security Vulnerabilities in Java SE. http://www.security-explorations.com/materials/se-2012-01-report.pdf. Technical Report.

66 Yannis Smaragdakis and George Balatsouras. Pointer Analysis. *Foundations and Trends in Programming Languages*, 2(1):1–69, 2015. doi:10.1561/2500000014.

67 Yannis Smaragdakis, George Balatsouras, George Kastrinis, and Martin Bravenboer. More Sound Static Handling of Java Reflection. In *Proceedings of the Asian Symposium on Programming Languages and Systems*, APLAS '15. Springer, 2015.

68 L. Sui, J. Dietrich, M. Emery, S. Rasheed, and A. Tahir. On the Soundness of Call Graph Construction in the Presence of Dynamic Language Features - A Benchmark and Tool Evaluation. https://sites.google.com/site/jensdietrich/publications/preprints/On%20the%20Soundness%20of%20Call%20Graph%20Construction%20in%20the%20Presence%20of%20Dynamic%20Language%20Features.pdf?attredirects=0&d=1. Accepted for APLAS'18.

69 Josef Svenningsson and Emil Axelsson. Combining Deep and Shallow Embedding of Domain-specific Languages. *Computer Languages, Systems and Structures*, 44(PB):143–165, December 2015. doi:10.1016/j.cl.2015.07.003.

70 Attila Szegedi. Dynalink - Dynamic Linker Framework for JVM Languages. http://medianetwork.oracle.com/video/player/1113272541001, July 2011. JVM Language Summit.

71 The Apache Groovy Project. Invoke dynamic support. URL: http://groovy-lang.org/indy.html.

72 Trifork. erjang. URL: https://github.com/trifork/erjang/wiki.

73 TypeLead. The Eta Programming Language. URL: http://eta-lang.org/.

74 Raoul-Gabriel Urma. Processing Data with Java SE 8 Streams, Part 1. *Java Magazine*, 2014. URL: http://www.oracle.com/technetwork/articles/java/ma14-java-se-8-streams-2177646.html.

75 Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a Java Bytecode Optimization Framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '99, pages 13–. IBM Press, 1999. URL: http://dl.acm.org/citation.cfm?id=781995.782008.

76 Ingo Wechsung. The Frege Programming Language (Draft). `http://www.frege-lang.org/doc/Language.pdf`, 2014.

77 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, New York, NY, USA, 2013. ACM. `doi:10.1145/2509578.2509581`.

78 Shijie Xu, David Bremner, and Daniel Heidinga. Mining Method Handle Graphs for Efficient Dynamic JVM Languages. In *Proceedings of the Principles and Practices of Programming on The Java Platform*, PPPJ '15, pages 159–169, New York, NY, USA, 2015. ACM. `doi:10.1145/2807426.2807440`.

79 Shijie Xu, David Bremner, and Daniel Heidinga. MHDeS: Deduplicating method handle graphs for efficient dynamic JVM language implementations. In *Proceedings of the 11th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*, ICOOOLPS '16, pages 4:1–4:10, New York, NY, USA, 2016. ACM. `doi:10.1145/3012408.3012412`.

80 Shijie Xu, David Bremner, and Daniel Heidinga. Fusing Method Handle Graphs for Efficient Dynamic JVM Language Implementations. In *Proceedings of the 9th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages*, VMIL 2017, pages 18–27, New York, NY, USA, 2017. ACM. `doi:10.1145/3141871.3141874`.

Reasoning About Foreign Function Interfaces Without Modelling the Foreign Language

Alexi Turcotte
Northeastern University, Boston, MA, USA

Ellen Arteca
Northeastern University, Boston, MA, USA

Gregor Richards
University of Waterloo, Waterloo, ON, Canada

Abstract

Foreign function interfaces (FFIs) allow programs written in one language (called the *host* language) to call functions written in another language (called the *guest* language), and are widespread throughout modern programming languages, with C FFIs being the most prevalent. Unfortunately, reasoning about C FFIs can be very challenging, particularly when using traditional methods which necessitate a full model of the guest language in order to guarantee anything about the whole language. To address this, we propose a framework for defining whole language semantics of FFIs without needing to model the guest language, which makes reasoning about C FFIs feasible. We show that with such a semantics, one can guarantee some form of soundness of the overall language, as well as attribute errors in well-typed host language programs to the guest language. We also present an implementation of this scheme, Poseidon Lua, which shows a speedup over a traditional Lua C FFI.

2012 ACM Subject Classification Software and its engineering → Interoperability; Software and its engineering → Semantics

Keywords and phrases Formal Semantics, Language Interoperation, Lua, C, Foreign Function Interfaces

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2019.16

Supplement Material ECOOP 2019 Artifact Evaluation approved artifact available at `https://dx.doi.org/10.4230/DARTS.5.2.9`

Acknowledgements The authors would like to thank Rafi Turas for writing the implementation of these techniques in Poseidon Lua. We'd also like to thank Hugo Musso Gualandi for his valuable discussions/feedback. This work was partially funded by NSERC.

1 Introduction

Often, programming languages are designed with a specific purpose or task in mind. For example, domain specific languages (DSLs) exist for a variety of domains (e.g., querying databases), and a programmer will often choose a DSL when solving a problem that falls in its domain. But, when a programmer wants to write code which touches on several domains, they turn to more general-purpose languages (e.g., Java) to give them the tools they need to do everything they need to do, even though the language might be worse at any one given task as compared to a DSL written specifically for it. With so many programming languages to choose from, not only is picking the right language non-trivial, picking the "wrong" language may come back to haunt you.

To make choosing a language easier, many programming languages are equipped to interoperate with other languages, and one of the most common forms of interoperation is the foreign function interface (FFI). FFIs allow code written in one language (called the

33rd European Conference on Object-Oriented Programming (ECOOP 2019).
Editor: Alastair F. Donaldson; Article No. 16; pp. 16:1–16:32
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

host language) to call functions written in another language (called the *guest* language), and also interface with data from the guest language, typically accomplished with wrapper code surrounding guest language values and regulating access to them. By and large the most common form of language interoperation is the C FFI, since C is so fast; C FFI's are available for Python, Lua and many other dynamic languages.

Semantically, interfacing with C exposes one to all of C's foibles and irregularities: Memory accesses can fail, return values of an incorrect type, or cause system-specific undefined behavior. As such, FFI's are usually avoided in language semantics, and assumed to be either benign or absent. Unfortunately, proving properties of the behavior of a C FFI using conventional techniques is challenging:

Of the existing body of work on formal specification of language interoperation, some are designed with a very specific use case in mind [6][1], and others propose general frameworks [16] which are difficult to use when reasoning about interoperation with C; these general approaches rely on fully defined semantics for all interoperating languages, which is usually infeasible when one of those languages is C.

In this paper, we aim to describe what behavioral guarantees remain true in the presence of an FFI, how a language hosting an FFI can guarantee its own type correctness at the interface, and how that can motivate the implementation of an FFI. We propose a framework which allows typed languages with a C FFI to be formalized and easily reasoned about *without a full model of C*. Our approach relies on a merger of the guest and host language's type systems, which allows us reason statically about the whole language and the host language's use of the FFI. Additionally, without a model of C, our semantics is *nondeterministic* – as there's no telling what an arbitrary C function might do – and we develop a novel method to reason about these nondeterministic semantics. In principle, this approach works well for interoperation with other languages too, though our model of C's memory and C's types in the host language make languages with similar memory behavior to C's most suitable.

As an example of our framework in action, we also present both the semantics and implementation of Poseidon Lua, a Typed Lua C FFI. In Poseidon Lua, Typed Lua interfaces with C by holding direct pointers to C data, and is equipped to dereference these pointers, cast them, allocate C data directly, as well as call arbitrary C functions. We prove conditional soundness of Poseidon Lua, and prove that if anything "goes wrong" in well-typed Poseidon Lua programs, C code is at fault for the error. Interestingly, merging the type systems of the constituent languages eliminates the need for wrapper code around guest language values, which contributes to improved overall performance.

The main contributions of this paper are:

- a framework for merging type systems of guest and host language to allow interoperation that can be easily reasoned about;
- a semantics for Poseidon Lua, a Typed Lua C FFI, implemented with our framework;
- an implementation, Poseidon Lua;
- improved performance results over the previously existing Lua C FFI.

2 Background

In this section, we will provide requisite background for understanding our proposed framework, as well as our prototype implementation, Poseidon Lua. We will begin with an overview of foreign function interfaces, as we are describing a framework for reasoning about them.

We will also discuss taint analysis, since the concept of taint features prominently in our semantics. We will then discuss Lua, Typed Lua, and Featherweight Lua, as all are crucial to understanding our language Poseidon Lua. We end the section with a quick highlight of some related work.

2.1 Foreign Function Interfaces

A foreign function interface (FFI) is a framework in which code written in one language (called the *host* language) may call code written in another language (called the *guest* language) as well as interface with data from that guest language. In an FFI, the guest language typically exports an API of available functions to the host language, and the host language calls said functions through the function interface. In addition to this function interface, a *data interface* is required to manage the use of one language's data in the other language.

FFIs are prevalent in modern programming. They date back to Common Lisp [11], which first introduced the concept of calling functions written in another language. Many dynamic languages, such as Python [23] and Perl [19], have easy-to-use C FFIs, allowing programmers to quickly and easily call functions written in C, a language known for its speed. In fact, C FFIs are very common, particularly in systems where performance is critical: Scientific computing environments, such as MATLAB [15] and Julia [9], carry out intensive numeric computations and simulations, and often programmers turn to external C functions available through an FFI to speed up the running time of their computationally intensive programs. This provides the user with an easy-to-use scripting language front end which may not be very performant, but with the ability to call fast functions when speed becomes an issue.

Most C FFIs interface with C in environments where C has access to all memory, including that of the host language, but there are exceptions where C is an embedded language with restricted access. One popular such system is Emscripten [29]. Emscripten is a source-to-source compiler from LLVM to JavaScript; its goal is to provide a way to run code on the web which can be compiled with LLVM but not natively run in browsers. Since JavaScript can run in essentially any web setting, compiling a language such as C to JavaScript would enable it to run reliably on a browser. With Emscripten, this can be done by first compiling the original source code down to LLVM, and then translating this to JavaScript. In terms of semantics, C is isolated to its own heap, and cannot interfere with JavaScript's; we use this style of isolation in our own semantics.

Idiomatic FFI usage is to minimize the data interface between the languages to the point where only primitive, scalar values are passed between the languages, as sharing actual structured data has unfortunate behavior: Often, if the FFI even has the capability to allow the host language to store pointers to guest structures, they are mediated through a wrapper. This wrapper problem is insidious: Consider, for example a list. With each access to the next element of a list, a new wrapper must be allocated, and the old wrapper discarded, so a series of simple accesses instead becomes a series of allocations. If the FFI has no capability to access structured guest data, as in Lua's built-in C FFI, the programmer has to write a C accessor for every member they want to access. While the definition of these accessors can be automated, they still incur the FFI to actually access the data, as the accessors are written in C.

Even in systems which generate the data interface statically, such as JNI [20] and SWIG [26], you still need wrappers. Imagine that we are using JNI or SWIG to interface with C. The problem there is that because no type system resembling C's is actually integrated into the host's type system, some layer is needed to make a working data interface, and that layer works exactly the same as in a fully-dynamic FFI. (Note that SWIG *can* be a partial exception depending on the host language: If the host language lets you hold raw pointers, it just generates a bunch of wrapper *functions*, instead of wrapper *objects*.) Our scheme, on the other hand, avoids using wrappers at all, with our strategy of integrating the guest type system into that of the host; this is discussed in more detail Section 3.

There has been some previous work on formally specifying FFIs, and language in general. One example is early work by M. Abadi and coauthors [1], which explores dynamic typing in a statically typed language, a mixing of two very different language paradigms. Other work by K. Gray [6] tackles the problem of multi-language object extension, and presents a sound calculus modelling the language interoperability and the semantics of objects written in one language being extended in another. Additional work by J. Matthews and R. B. Findler [16] realizes whole language semantics by defining full semantics for host and guest languages, and uses *boundaries* to explicitly regulate value conversions. For our purposes, these approaches are either too specific [1][6], or do not generalize to reasoning about languages with a C FFI [16]. One particular work has a similar motivation to ours and has a fairly generalizable approach: linking types presented by Patterson and Ahmed [22]. This is discussed below in Section 2.4.

2.2 Dynamic Taint Analysis

Introduced by Newsome and Song in their paper [18], dynamic taint analysis is a technique initially developed for tracing potential error propagation through a system, in order to detect exploits on commodity software. The idea is that some data sources are considered *untrusted*, and data which originates from these sources is labelled with taint. This allows for the tracking of potential errors, and also can be used to restrict what the tainted data can be used for. In addition, if there is an error in the program that involves some of the tainted data, information on what potentially caused the error is all available as taint information.

The idea of dynamic taint analysis can be generalized to the tracking or propagation of any tagged (tainted) data in a program. In this work, we adapt the concept of taint to reasoning about a C FFI without modelling C: when a C call occurs, we cannot say what *will* happen, but we can reason about what *could* happen. We can model arbitrary C calls by tagging any data which could have been modified by the call with taint information identifying it, and should an error occur involving any of this data, the taint can point to the call which tampered with the data. Note that this is a property of the semantics for the purpose of proofs; we do not demand that an implementation track dynamic taint. This is explained in detail in Sections 3.2 and 4.

2.3 The Base for Poseidon Lua

Later in this work, we will be presenting Poseidon Lua, a Typed Lua C FFI. In this section, we present variants of Lua, the host language in Poseidon Lua. First we discuss the Lua language itself, before turning our attention to its variants and extensions.

2.3.1 Lua

Lua is a lightweight dynamic imperative scripting language with lexical scoping and first class functions. Lua is extensible, and offers many metaprogramming mechanisms to facilitate adaptation of the language. Its main data structure is an associative array known as the table, which can stand in for most common data structures, such as arrays, records, and objects. The functionality of tables can be further augmented through metamethods, which are essentially hooks for the Lua compiler. Classic object-oriented programming patterns, such as methods and constructors, can be easily encoded in Lua with these table extensions. A C FFI was developed for Lua by Facebook [3]: called luaffifb, it is a standard C FFI which wraps C data for use by Lua. Note that we did not implement Poseidon Lua on top of LuaJIT [21], as the implementation merely serves as a demonstration of our semantics, and JIT compilers are less amenable to such modifications. Also, LuaJIT offers the same sort of data interface that we do, but without types and with boxed references to C structures – our techniques would thus apply to it for better performance.

Our approach to reasoning about FFIs involves embedding the type system of the guest into the host language, but Lua has no type system to embed into! For this reason, Lua is not the host language in Poseidon Lua – as we need a type system, we chose Typed Lua as a base.

2.3.2 Typed Lua

Lua is a dynamic language, and as is often the case with these languages (see TypeScript [17] and Typed Racket [27]), there have been a few attempts at adding types in some form. One such example with Lua specifically is Tidal Lock [12], a static analyzer relying on simple type annotations. Another is Typed Lua, an optional type system for Lua [14].

In their design of Typed Lua, Maidl et al. performed an automated analysis of existing Lua programs to obtain a clear picture of how programmers use the language; they paid close attention to idiomatic Lua code to ensure that their design aligned with conventional language use. Typed Lua is optionally typed, which means that the type annotations are removed when code is compiled. Typed Lua accounts for a large subset of Lua, but a few parts are omitted, namely polymorphic functions and table types, and certain uses of the **`setmetatable`** function. The type system of Poseidon Lua largely matches Typed Lua's, and a full discussion will appear in Section 4.1.

Like other optionally and gradually typed languages, a program written in Typed Lua has an initial stage of *type compilation*. First, the Typed Lua code gets translated (i.e., compiled) to its corresponding Lua program, and it's during this first phase of compilation that the type information is used. At "type compile" time, typed code can be checked statically for type errors before being translated. The type information has no effect on the generated Lua code; Typed Lua programs are type checked by the compiler, and if they are well-typed, the compiler simply erases the types, generating plain Lua. Then, this Lua code is compiled to bytecode and run on the Lua virtual machine.

This multistage process means that there are two distinct versions of Lua involved in running a Typed Lua program. For clarity, in our discussion of Poseidon Lua we will use the following terminology: Typed Lua will be referred to as the *typed language* or the *user language*, since this is the language in which the programmer will be writing programs. Then, the *untyped language* or the *run-time language* refers to the subset of Lua resulting from the compilation of user language programs and additional expressions needed to deal with C. Both of these languages' grammar and operational semantics are given in Section 4.

In giving a prototype using our framework we needed to develop a formal representation of Poseidon Lua. Poseidon Lua is formalized using a *core calculus* based on Featherweight Lua (FWLua) [10], itself a core calculus of Lua (discussed next).

2.3.3 Featherweight Lua

There have been a few formal specifications of Lua. First, a semantics was developed by M. Soldevila and coauthors [25] to gain a deeper understanding of Lua programs; it was mechanized in PLT Redex [4] using reduction semantics with evaluation contexts. Another semantics, not unlike Featherweight Java [8] and LambdaJS [7], proposes a core calculus for Lua. Called Featherweight Lua (FWLua) [10], this semantics focuses on formalizing what authors deem to be the essential features of Lua: first-class functions, tables, and metatables. Remaining Lua features, including expression sequencing and control structures, are shown to reduce into FWLua through an extensive desugaring process. The FWLua specification [10] also provides a reference interpreter written in Haskell.

The principle goal of FWLua is to capture core Lua idioms, and a crucial aspect of the Lua language is its table construct. Under the hood, Lua handles table access and table write with `rawsget` and **`rawset`** functions, respectively; these are not typically written by the programmer, but are part of how Lua drives table functionality. In their design of FWLua, the authors modelled table access and table write wholly with these **`rawget`** and **`rawset`** operations, and together with other basic semantic constructs (e.g., functions and binary operations) propose functions which mimic the semantics of full-fledged Lua. For example, to capture Lua's scoping rules, FWLua reserves certain tables to be so-called "scope tables": the `_local` table is one such example and is always accessible, and changes whenever a new scope is entered while keeping a reference to its outer scope in its `_outer` member. This way, variable access (say, of `x`) is desugared into a function which first searches through `_local`, and if `x` is not present in `_local`, then it searches recursively through `_local._outer`, and so on until `x` is located, producing `nil` if `x` is not found. This proved challenging to reason about, so we chose to promote variables to first-class language members.

To contrast Lua and FWLua, consider the following, which illustrates table construction in Lua:

```
local t = {}
t.x -- nil, uninitialized table members are nil
t.x = 42 -- t.x is now 42
t[0] = "hello" -- tables may be indexed like arrays
t["hi"] = 3.14 -- equivalent to t.hi
```

As you can see, tables can be accessed in a variety of ways in Lua, and have syntax which specifically supports different access styles, be it array-style or record-style. Tables are incrementally constructed, and can be extended at any time, much like dynamic object extension in JavaScript or other dynamic languages. In FWLua, the above translates (with a line-by-line correspondence) to:

```
rawset(_local, "t", {})
rawget(rawget(_local, "t"), "x")
rawset(rawget(_local, "t"), "x", 42)
rawset(rawget(_local, "t"), 0, "hello")
rawset(rawget(_local, "t"), "hi", "hello")
```

Here we see the **`rawset`** and **`rawget`** functions are used to write and read from a table, respectively. As we mentioned earlier, FWLua desugars variables into special table members: The table `_local` deals with local variables, and the table `_ENV` deals with global variables.

2.4 Related Work: Linking Types

Linking types, presented by Patterson and Ahmed [22], consider a different approach to reasoning about language interoperation. This work considers the languages working together as components within a larger language, which itself encompasses behavior of one language as well as the added behavior of making calls to the other language. Linking types themselves are designed to allow programmers to express and reason about one language's features in another (possibly) less expressive language which has no concept of those features. With linking types, the programmer can annotate a program to indicate where it interfaces with more expressive code in the linked language. Then, with these types, reasoning about the behavior of the whole program becomes possible.

Although both their work and ours are motivated by the same essential problem, they require modelling of both languages and focus more on the language of types than on semantics or proofs. In our work, we take a notably different approach in deciding not to model the behavior of the guest language, and instead work with the semantics of the point of intersection (i.e. the boundary between host and guest), using nondeterminism to consider the potential outcomes of the guest language calls. We believe that our types could be expressed in terms of linking types with no meaningful change to our semantics or proofs, but have not investigated this.

3 The Problem

FFIs are ubiquitous in programming, and C FFIs are by far the most common, but they are usually excluded from formal treatments of programming languages. Unfortunately, traditional methods of reasoning about FFIs necessitate a full semantic model of the guest language to show anything about the overall system: Defining a formal semantics for C is very involved, and, any such semantics will be compiler-dependent. For example, while the CompCert [2] project was groundbreaking in their implementation of a formally verified C compiler, their guarantees are limited to C programs compiled with this compiler, and do not hold for C programs compiled on other compilers (such as `gcc`).

Hypothetically, if we had a whole language semantics for a system with a C FFI, what might we be interested to show? One result of interest would be some form of type soundness for the host language, to ensure that the inclusion of the FFI in the semantics didn't cause any strange issues. Additionally, we might like to show that if any failures occur in a well-typed program calling a C FFI, then C is in some way *at fault* for the failure. In this work, we show that we can get these results *even without a full model of C!*

To achieve this, we will need to be able to reason statically about *use* of the FFI (i.e., the host's interface with the guest). The function interface of an FFI exports function handles, so we can at least check that functions are being called and used correctly, even if we don't know exactly what they do. However, the data interface of FFIs is typically built up *dynamically*, and cannot be reasoned about statically. Indeed, in a conventional FFI, wrappers are built up at run-time as values flow from one language to another, and dynamically regulate access to underlying data.

In order to fully guarantee that the host language's use of the C FFI is correct, we need the data interface to be static, and we can achieve this by embedding C's type system into the type system of the host language. This way, the host language can express C types and statically check its own use of C data instead of relying on run-time wrapper code like in traditional approaches. As it happens, with this scheme wrappers are no longer necessary, and their removal results in improved performance; this is discussed further in Section 5.

It's not enough to have a system in place to statically reason about the host language's use of the C FFI, as we still need to consider how we can model calls to C when we have no model of the C code, and how we can reason about the resulting semantics.

3.1 Taint and Nondeterminism

With no model of foreign C code, a well-typed call to a C function exhibits *nondedeterminism*. Without analyzing the C code we cannot reason statically about what exactly the function does (e.g., a C function could dereference a null pointer or otherwise crash the program). To account for this, at least two semantic rules for guest language calls are required: one modelling a *successful* call where the function didn't crash and returned something to the host language execution, and another modelling *failure*, where the function failed to do so (or, more generally, failed to successfully pass execution back to the host). Note that the rule for failure must have strictly more permissive preconditions that any rule modelling a successful call, as failure must always be an option.

Unfortunately, this simple model of nondeterministic success and failure of a particular call does not fully account for all effects that C can have. For instance, executing a C function could free some memory that the host program has access to while still terminating and returning successfully, and the next dereference of a pointer to that memory would fail or return unexpected values. To fully account for this case where a successful C call has detrimental side effects, we need some additional mechanism to indicate to subsequent reductions that the function may have tampered with some data.

To model the fact that black-box C code may arbitrarily modify data, we use the concept of *taint* as described in 2.2; here, even successful calls to C functions will taint the memory locations which may have been modified (i.e. all the memory C has access to). The presence of taint at a memory location indicates that use of the location is nondeterministic: the next use of the location could either succeed, indicating that no fatal modification was made, or it could fail, indicating that the location was fatally modified by the call which placed the taint. Note that success in accessing a tainted location does *not* mean that the value at that location is the value that was there before it became tainted, it just means that the access did not crash; C could still have changed the value in a way that was not fatal to the program. Crucially, successfully using a tainted location will *clean* or *remove* the taint, as from that moment until the next C call we are sure that the location is not somehow broken, and that its value will not change (unless overwritten by Lua).

In summary, nondeterminism and taint together enable us to express the effects that C may have on the host language program *without* modelling C. Note that since we use a nondeterministic semantics for C and thus avoid modelling its behavior, in principle this approach works well with other languages. However, our model of C's memory and C's types in the host language make languages with similar memory behavior to C's most suitable.

To demonstrate this framework, we will present the semantics of Poseidon Lua, a Typed Lua C FFI. A high-level description of Poseidon Lua will be given in the next section.

3.2 Overview of Poseidon Lua

Essentially, Poseidon Lua is Typed Lua with a C FFI. It is fine-grained relative to standard FFIs: Unlike traditional FFIs, in Poseidon Lua the type systems of Lua and C are merged through a Lua pointer type, and the language has syntax with which the Lua programmer can allocate and manipulate these pointers. Specifically, Poseidon Lua allows you to: allocate and use C data, cast said pointers, and call C functions. The formal semantics are discussed fully in Section 4.

In our semantics of Poseidon Lua, Lua directly holds C values through a *pointer* to some location in a C store, which is separate from Lua's store. Structs are laid out in the C store as they would be in C, taking up space proportional to the number of struct members;

these members can then be accessed with an offset equal to its position in the list of struct members (like accessing elements in an array). As explained, with no model of C, C function calls are nondeterministic, with successful calls taint everything in the C store – for this reason, our formalization includes optional taint information in the C store. Access to clean (i.e., taint-free) locations in the C store are deterministic, while accesses to tainted locations are not, and in the event of successful access to a tainted location the taint can be removed and future accesses to that same location become deterministic (at least, until the next call to a C function).

Another interesting application of taint is in modeling C's undefined behaviour, of which one classic example is casting pointers. In Poseidon Lua, as in C, pointers to C values may be downcast. To model this in our formal semantics, we include types in the C store, alongside taint and the values themselves – the C store is thus a list of triples of (*value*, *type*, *optional taint*). This way, we can model the cast of a Lua pointer (to a C value) to some type T by changing the type held at the pointer's location in the C store to T. But that's not quite enough, as casting pointers is undefined behavior in C, and we can use taint to cleanly capture this: Once cast, the location becomes tainted, and the next access to that location is nondeterministic. In this scenario, taint indicates the cast location's potential for undefined behavior when it is accessed.

Another use of taint in Poseidon Lua is in our modelling of allocation of C pointers. In C, the `calloc` function initializes the allocated memory with `0`s, so in allocating a pointer to a pointer, one is actually allocating a pointer to a `0` (which is to be treated as a pointer)! Indeed, if one were to dereference the second pointer, one would be dereferencing `0` which leads to a segmentation fault in most circumstances (`0`, of course, is `NULL` in C). To achieve this in our semantics, we taint the allocated memory location when a (Lua pointer to a) C pointer is being allocated, to indicate the potential failure of the next access to this location.

Even though we don't model C, we do make some assumptions about C's behavior: For one, we assume that C does not touch Lua's memory, and that its effects are contained to an explicitly defined C store: in other words, the shared memory has clearly defined bounds. This mirrors reality in most other FFIs, where guest code and data is not aware of host code and data. However, it is technically possible for C code to violate this assumption. We also make a simplifying assumption that all allocation and access is by word, which reduces the complexity of C data accesses without loss of generality. We require that C doesn't write new or mutate existing Lua code, otherwise we would have to scrutinize existing expressions that have yet to be reduced and would be unable to prove anything. We additionally make no explicit mention of the stack pointer, which would needlessly complicate function calls and returns for no real benefit. Further, C functions cannot call Lua functions in our formalization, so as to package all of C's effects into one black box; this is possible through callbacks, but would again be very complex without meaningfully improving the semantics. Finally, we disregard threads, which avoids needing to reason about the effects of concurrency on top of the effect of C, a layer of complexity which is outside of the scope of this project.

4 Semantics

Poseidon Lua is our proof-of-concept for the ideas discussed in Section 3. Having highlighted some of the stranger corners of our formal specification of Poseidon Lua in Section 3.2, we will now discuss the C FFI in its entirety.

In Poseidon Lua, Lua primarily interacts with C by calling C functions, and our merger of the two languages necessitates that C values be a part of the broader language. To represent these C values, Typed Lua has a concept of a Lua pointer to a C value, which is Lua's

$$\begin{array}{rcll}
T & ::= & \textbf{nil} & \textit{nil type} \\
 & | & \textbf{value} & \textit{top type} \\
 & | & \textbf{ref}\, T & \textit{reference type} \\
 & | & T_1 \cup T_2 & \textit{union type} \\
 & | & L & \textit{literal type} \\
 & | & B & \textit{base type} \\
 & | & T_1 \rightarrow_L T_2 & \textit{function type} \\
 & | & \{f_1, ..., f_n\} & \textit{table type} \\
 & | & \textbf{ptr}_\textbf{L}\; T_C & \textit{Lua pointer type} \\
T_C & ::= & \textbf{int} & \textit{C integer type} \\
 & | & T_C^1 \rightarrow_C T_C^2 & \textit{C function type} \\
 & | & \textbf{ptr}_\textbf{C}\; T_C & \textit{C pointer type} \\
 & | & \{s_1 : T_C^1, ..., s_n : T_C^n\} & \textit{C struct type}
\end{array}$$

$$\begin{array}{rcll}
f & ::= & s : T & \textit{field} \\
 & | & \textbf{const}\; s : T & \textit{const field} \\
L & ::= & b & \textit{boolean literal} \\
 & | & n & \textit{numberliteral} \\
 & | & s & \textit{stringliteral} \\
B & ::= & \textbf{boolean} & \textit{base types} \\
 & | & \textbf{number} & \\
 & | & \textbf{string} &
\end{array}$$

Figure 1 The Poseidon Lua type system.

window to accessing C data. This means that Lua never deals directly with C values per se, and instead deals with pointers to these values. As mentioned previously, we implement the additional functionality of allocating C data as well as downcasting C pointers, both directly from Lua code without needing to call C.

We start by describing the type system in detail, and follow with a presentation of a core calculus which models the language. Then, we discuss the typing and reduction relations before concluding with a discussion of soundness and other interesting proven results.

4.1 Type Systems

Poseidon Lua's type system is a combination of Typed Lua's [14] and C's type systems. For illustrative purposes, we chose a subset of C's type system which highlights some of C's interesting features without getting bogged down in the low-level details; we only formalized integers, pointers, structs, and functions. These are not limitations of the concept, merely simplifications made to the formalization. The story is similar with Typed Lua's type system; our function type only has a single argument type, and multivariate functions are curried to repeated application of single variable functions, by which a single argument function type suffices. In fleshing out this type system for our core calculus, we found no need for Typed Lua's type variables, recursive types, and projection types, and were able to greatly simplify their table type. Further, to simplify reasoning about Lua, we only allow string indexing in tables. Again, these are not limitations of the language, and are only simplifications for the purposes of formalization.

Our types are given in Figure 1, and explained in detail throughout this section. Type ordering is as follows:

- **value** is a supertype of all types;
- **nil** is the type of Lua's `nil` value, and is a subtype of all base types;
- union types are supertypes of their members;
- literal types are the types of literals (e.g. the literal type of `5` is *5*), and base types are the more general typical types of these literals (e.g. the base type of `5` is *number*) – that said, literal types are subtypes of their corresponding base types;
- function types are contravariant in their argument types, and covariant in their return types;
- table types have **width subtyping**: A table type T is a supertype of a table type T' which has a superset of all of the fields of T (in other words, adding extra fields preserves the subtyping relationship);

- table types have **depth subtyping** only on **const** fields: If a table type T has a **const** field x with type T_x, and a table type T' has all the same fields as T except that field x has type T'_x, where $T'_x <: T_x$, then $T' <: T$ (in other words, **const** field types may be specialized while preserving the subtyping relationship)

C's types are included in the Typed Lua type system (and made accessible to the user) via the "Lua pointer" C type $\mathbf{ptr_L}\ T_C$; here, $\mathbf{ptr_L}$ denotes a Lua pointer type, and T_C is the C type being pointed to (e.g., $\mathbf{ptr_L}$ **int** is a Lua pointer to a C integer). As explained above, Lua only ever deals with *pointers* to C values, and not C values themselves: the only access to C values is through this pointer. C's type system is consequently entirely self contained, and is a strict subset of Lua's with no ability to reference Lua types. In some sense, C is "plugged" in to Lua through the $\mathbf{ptr_L}\ T_C$ type.

While we don't formally model C, we do need some information on C functions in order to ensure that everything shakes out properly at run-time. For example, in our semantics we model C functions as black boxes with no function body, and we ask for parameter and return types for these functions to ensure that they are called with correctly-typed arguments, even though the function bodies themselves are not modeled. What this means is that we can make sure that the functions are called correctly, but are not responsible for their internal behavior. Indeed, FFIs typically export function types as part of their API and may not always export their code – this is the situation modeled by our semantics. This is also analogous to a user calling a library for which the source code is not provided, even when the library is written in the same language as the "library host" language.

4.2 The Language

In this section, we present a core calculus modelling Poseidon Lua, akin to FWLua [10]. We will discuss the language of expressions, both typed and untyped, before moving on to the typing judgment and reduction relation.

We present *two* languages (in the same manner as Typed Lua, recall from Section 2.3.2): The language of *untyped expressions* E, also known as the language of *run-time expressions*, is the language that will actually reduce at run-time, and the language of *typed expressions* TE is the language that programmers will interface with and program in, with a few minor caveats which will be discussed in time. Roughly, the typed language corresponds to Typed Lua with our added C FFI, and the untyped language corresponds to a subset of Lua with additional expressions for C interoperation. We begin with the typed language TE.

4.2.1 Typed Language

Figure 2 presents the language of typed expressions, representing the language that the programmer will be interfacing with, with some notable exceptions. The *Lua dereference* and *location update* expressions, and the *Lua location* value are not explicitly written by the programmer; they are artifacts of our typing judgment which will be presented in Section 4.3. We sometimes refer to the aforementioned expressions as *intermediate expressions*; the typed language without these is the *user language*.

These expressions largely describe a core calculus of Typed Lua, with the exception of the following C expressions:

- *C downcast* denotes the cast of expression te to C type T_C;
- *C allocation* allocates a *C pointer* to a value of C type T_C;
- *C deref* is used to dereference the C pointer expression te;

te	$::=$	v_t	*value*
	$\mid$	$\{s_1 = v_1, ..., s_n = v_n\}$	*table*
	$\mid$	$\mathbf{let}\, x : T := te_1 \,\mathbf{in}\, te_2$	*let binding*
	$\mid$	$x := te$	*variable update*
	$\mid$	$\mathbf{loc}\, n := te$	*location update*
	$\mid$	$\mathbf{deref}\, te$	*Lua dereference*
	$\mid$	$te_1 \, op \, te_2$	*binary operation*
	$\mid$	$te_1(te_2)$	*function call*
	$\mid$	x	*variable*
	$\mid$	$te_1.te_2$	*dot access*
	$\mid$	$te_1.te_2 := te_3$	*dot update*
	$\mid$	$\mathbf{cast}\, te\, T_C$	*C downcast*
	$\mid$	$\mathbf{calloc}\, T_C$	*C allocation*
	$\mid$	$\mathbf{deref_C}\, te$	*C deref*
	$\mid$	$te_1;\ te_2$	*sequence*

v_t	$::=$	$\mathbf{nil}$	*nil value*
	$\mid$	r	*register*
	$\mid$	c	*constant*
	$\mid$	$\mathbf{loc}\, n$	*Lua location*
	$\mid$	$\lambda x : T.te$	*Lua function*
	$\mid$	$\mathbf{cfun}\, T_C$	*C function*
	$\mid$	$\mathbf{ptr}\, n\, T_C$	*C pointer*
r	$::=$	$\mathbf{reg}\, n$	*table store loc*
c	$::=$	n	*number*
	$\mid$	b	*boolean*
	$\mid$	s	*string*
op	$::=$	$+, -, *, /$	*arithmetic*
	$\mid$	$\leq, <, \geq, >$	*order*
	$\mid$	$\wedge, \vee$	*boolean*
	$\mid$	$..$	*concatenation*
	$\mid$	$==$	*equality*

Figure 2 The language of typed expressions.

- *C function* describes a C function with type signature T_C. The type T_C is required by the type transformation to type these functions, as it cannot leverage the function body (as is the case with traditional functions);
- *C pointer* is a pointer to location n in the C store, with expected C type T_C;
- Access to C structs is done through the *dot access* and *dot update* expressions (so long as *te₁* is a C struct), and calling C functions is done through the *function call* expression (so long as *te₁* is a C function).

Besides the C expressions, the typed language is standard or otherwise directly analogous to some untyped expression, which we will discuss in more detail shortly.

Typed expressions will all compile into equivalent run-time expressions where the types have been erased. We explore this run-time language next.

4.2.2 Untyped Language

The untyped language describes the expressions which will reduce/evaluate at run-time. Generally speaking, they are analogous to some equivalent typed expression where the types have been erased. This language essentially describes a core calculus of Lua, based on FWLua (described in Section 2.3.3), though we added sequencing, let bindings, variables, table literals, and of course C interoperability. The full language can be found in Figure 3.

FWLua is a core calculus of Lua, and a number of minor modifications were required when adapting FWLua to describe Typed Lua, particularly with tables. Recall that tables are the principle data structure in Lua; as discussed previously, FWLua desugars all of Lua's table manipulation into the dual **rawget** and **rawset** constructs. For the purposes of formalization, we needed to relax FWLua's extreme desugaring; one example of this being the table literal (*table*) expression. FWLua handles table construction incrementally: an empty table is first created and stored, and then it is populated with the values at the programmer's discretion. Unfortunately, this scheme fails in typed languages, as the empty table is not a subtype of any non-empty tables, so we include a table literal to allow the expression of a full table when needed for assignments.

$$
\begin{array}{lcll}
e & ::= & v & \textit{value} \\
 & | & \{s_1 = v_1, ..., s_n = v_n\} & \textit{table} \\
 & | & \mathbf{rawget}\, e_1\, e_2 & \textit{table select} \\
 & | & \mathbf{rawset}\, e_1\, e_2\, e_3 & \textit{table update} \\
 & | & e_1\, op\, e_2 & \textit{binary operation} \\
 & | & e_1(e_2) & \textit{Lua fun. appl.} \\
 & | & x & \textit{variable} \\
 & | & x := e & \textit{var. assignment} \\
 & | & \mathbf{loc}\, n := e & \textit{location update} \\
 & | & \mathbf{deref}\, e & \textit{Lua dereference} \\
 & | & \mathbf{let}\, x := e_1\, \mathbf{in}\, e_2 & \textit{let binding} \\
 & | & \mathbf{cget}\, e\, n\, T_C & \textit{C store access} \\
 & | & \mathbf{cset}\, e_1\, n\, e_2\, T_C & \textit{C store update} \\
 & | & \mathbf{ccall}\, e_1\, e_2\, T_C\, \beta & \textit{C function call} \\
 & | & \mathbf{calloc}\ T_C\, \beta & \textit{C allocation} \\
 & | & \mathbf{cast}\, e\, T_C\, \beta & \textit{C downcast} \\
 & | & e_1;\ e_2 & \textit{sequence} \\
 & | & \mathbf{err}\, \beta & \textit{error expression}
\end{array}
\qquad
\begin{array}{lcll}
v & ::= & \mathbf{nil_L} & \textit{nil value} \\
 & | & r & \textit{register} \\
 & | & c & \textit{constant} \\
 & | & \mathbf{loc}\, n & \textit{Lua store loc.} \\
 & | & \mathbf{ptr_L}\, n\, T_C & \textit{C store pointer} \\
 & | & \lambda x.e & \textit{Lua function} \\
 & | & \mathbf{cfun} & \textit{C function} \\
v_C & ::= & \mathbf{ptr_C}\, n & \textit{C store pointer} \\
 & | & n & \textit{C number literal}
\end{array}
$$

Figure 3 The language of untyped, run-time expressions.

Our function expression is unchanged from FWLua, though we must include a new C function expression to allow FFI calls. Unlike the Lua function, which is a traditional lambda expression, the C function has far less information in it – indeed, it has no function body! Most of the information needed for a C call is stored in the C function call expression itself.

For accesses into C structs, we have the **cget** and **cset** expressions, analogous to **rawget** and **rawset**. **cget** and **cset** are also used for accessing and writing to C pointers, which will be discussed in more detail in Section 4.4. In $\mathbf{cget}\, e\, n\, T_C$, e is a pointer into the C store, n is the offset of the access, and T_C is the type that the **cget** is expecting to read. Similarly in $\mathbf{cset}\, e_1\, n\, e_2\, T_C$, e_1 is a pointer into the C store, n is an offset, e_2 is the value to write, and T_C is the type that the **cset** is expecting the store to contain at the referenced pointer (recall that we store type information for each pointer in the C store).

To call functions, programmers may write a standard function application as $te_1(te_2)$ in the typed language of Figure 2. The type transformation can, depending on the type of te_1, transform the application into either a Lua function application or a C call. The Lua function call expression $e_1(e_2)$ is straightforward, so let us focus on the C call: In **ccall** e_1 e_2 T_C β, e_1 is the C function being called, e_2 is the argument to that function, T_C is the function's type, and β is an identifier associated with the call (its line of code). The type is necessary since C calls exhibit nondeterministic behavior, and we can leverage T_C to reason about the value that is returned from the function. The line of code information β is related to taint, which we will describe fully when giving the semantics of the calls.

There are also a few expressions for functionality unique to C. As one might expect, **calloc** T_C β allocates something of C type T_C, and β is the identifier uniquely associated with the allocation, which allows a trace-back if a run-time error occurs. **cast** e T_C β downcasts the pointer e to type T_C, and again β is a unique identifier associated with the cast.

4.3 Typing Judgment

Making a distinction between typed and untyped languages (or user and run-time languages) makes sense in many optionally or gradually typed languages, where a typed language is compiled into an untyped language which will be the one executing at run-time (recall the two stage compilation process described in the context of Typed Lua in Section 2.3.2). In these settings the typing judgment often needs to be modified to connect the languages together. We define a *type transformation* relation, a modification of the standard *typing judgment* relation, which transforms/compiles a typed expression into its corresponding untyped expression:

$$\Gamma, K \vdash te : T \rightsquigarrow e \tag{1}$$

Here, Γ is the typing environment, which assigns types to variables, and K is the typing context, containing information about the various store typings. Our run-time environment contains three stores: a table store for Lua tables, a C store for C values, and a variable store for variables. K can thus be broken up into three store typings: Σ_T describing the table store, Σ_C for the C store, and Σ_V for the variable store. Roughly speaking, the type transformation takes a typed expression te and "compiles" it into an untyped expression e, assigning to it type T in the context of Γ and K.

In the following typing rules, some auxiliary functions will appear in the preconditions to simplify the notation. They are as follows:

- $goodLayout(n, T_C, \Sigma_C)$ checks to see if location n in the C store typing Σ_C represents type T_C. If T_C is a primitive type or a pointer type, this succeeds if $\Sigma_C(n) = T_C$. As for structs, recall that they are laid out contiguously in the store: If T_C is a struct type (for example, $\{s_1 : T_C^1, ..., s_n : T_C^n\}$), then each of the fields must be present in Σ_C with the correct type, i.e. for all fields s_i we must have $\Sigma_C(n + i) = T_C^i$.
- $offsetForType(s, T_C)$ computes the offset of member s in structure type T_C. Our formalization of the C store lays out structs according to their type, and this function relates their type (T_C) to their layout in the store.

As we mentioned, in Poseidon Lua, Lua can interact with C in the following ways: allocation and access of C data, C function calls, and casting of C pointers. In this section we will focus on the typing rules for the expressions describing this FFI. The full typing rules are given in Appendix A.1.

We will first consider the rule for allocation of C data.

$$\frac{validType(T_C) \qquad \beta\ unused}{\Gamma, K \vdash \mathbf{calloc}\ T_C\ : \mathbf{ptr_L}\, T_C \rightsquigarrow \mathbf{calloc}\ T_C\ \beta} \quad \text{(TT\_CALLOC)}$$

In Poseidon Lua, programmers can allocate Lua pointers to C data types (here, T_C), provided that the type is *valid* for allocation. For this to be the case, T_C must either be a primitive type, pointer type, or struct (itself recursively made up of valid types). This prevents programmers from making nonsensical statements, such as allocating C functions in Lua. The β here is needed when allocating C pointers: In C, allocating a pointer to a pointer can cause issues if the innermost pointer is not properly initialized, due to the default values that C inserts (pointer values are often initialized to `0`, which is an invalid memory address for C to access). This semantics will be dealt with in due course, and the inclusion of β in the **calloc** expression is crucial to achieving the desired behavior – this will be further discussed in Section 4.4.

Having seen C allocation, we turn our attention to typing (Lua pointers to) C values:

$$\frac{\begin{array}{c} n < length(\Sigma_C) \\ goodLayout(n, T_C, \Sigma_C) \end{array}}{\Gamma, (\Sigma_T, \Sigma_C, \Sigma_V) \vdash \mathbf{ptr_L}\, n\, T_C : \mathbf{ptr_L}\, T_C \rightsquigarrow \mathbf{ptr_L}\, n\, T_C} \quad \text{(TT\_LUA\_PTR)}$$

C values are always "hidden behind" a Lua pointer in Poseidon Lua, and so from Lua's point of view all C values have some $\mathbf{ptr_L}$ type. In the expression $\mathbf{ptr_L}\, n\, T_C$, n is the location referenced by the pointer, and T_C specifies the type that the location is intended to have. The type information is required since structures do not directly inhabit the C store, and so accessing a structure would be impossible with a simpler rule, since $\Sigma_C(n)$ will never have a struct type; the type information allows us to check to see if location n does in fact correspond to T_C using the *goodLayout* auxiliary function, and only allow the pointer to type if it does. The typing rule for dereferencing these pointers follows.

$$\frac{\begin{array}{c}\Gamma, K \vdash te : \mathbf{ptr_L}\, T_C \rightsquigarrow e \\ \mathit{validForCDeref}(T_C) \qquad T_L = \mathit{coerceCType}(T_C)\end{array}}{\Gamma, K \vdash \mathbf{deref_C}\ te : T_L \rightsquigarrow \mathbf{cget}\ e\ 0\ T_C} \qquad (\text{TT\_VAR\_C\_DEREF})$$

Here, beyond ensuring that te is in fact a Lua pointer, we need to ensure that it is a pointer to a type that we can dereference. The C store is made up entirely of primitives and pointers, so we disallow dereferencing of things of another type (for example, we cannot dereference a C function pointer). Because our type transformation deals with Lua types only, we need to coerce T_C into a Lua type to type this expression: Indeed, at run-time the dereference will coerce the value it obtains from the C store, and the coercion at this level allows such an expression to type. Note also the untyped expression corresponding to the dereference: **cget** can play the part of either simple dereferencing and also struct field access, depending on the value of its offset parameter (here, `0`). An offset of `0` indicates that we are either getting the first member in a struct, or simply dereferencing a pointer to non-struct data.

We consider C functions next.

$$\frac{}{\Gamma, K \vdash \mathbf{cfun}\,(ct_1 \rightarrow_C ct_2) : (ct_1 \rightarrow_C ct_2) \rightsquigarrow \mathbf{cfun}} \qquad (\text{TT\_C\_FUNCTION})$$

Here, note that the C function expression contains the whole type of the function, and without a body the function trivially types. Type information is necessary because we don't model C's semantics: In typical typing rules for functions, the return type can be determined thanks to the function body, and we have no such body to rely on here. In some sense, this is in line with what one would expect when dealing with FFIs, since part of their API is the full type of the exported functions.

Let us consider how one calls these functions:

$$\frac{\begin{array}{c}\Gamma, K \vdash te_1 : (T \rightarrow_C T') \rightsquigarrow e_1 \\ \Gamma, K \vdash te_2 : T \rightsquigarrow e_2 \qquad \beta\ \mathit{unused}\end{array}}{\Gamma, K \vdash te_1(te_2) : T' \rightsquigarrow \mathbf{ccall}\, e_1\, e_2\, T'\, \beta} \qquad (\text{TT\_C\_FUN\_APPL})$$

In rule TT_C_FUN_APPL, we type the function application according to its return type. Note the T' in the compiled (on the right of the $\rightsquigarrow$) C call: The untyped call requires the return type for reduction to be possible, and we will discuss this in more detail in Section 4.4. Since C calls are sources of taint, we include β as an identifier uniquely associated with the call, which corresponds to the line of code occupied by the call. In the event of a failure, we can determine which call (and, thus, which function handle) is to blame.

We will now consider reading from and writing to C structs. First, reading:

$$\frac{\begin{array}{c}\Gamma, K \vdash te_1 : \mathbf{ptr_L}\, T_1 \rightsquigarrow e_1 \qquad \mathit{structType}(T_1) \\ \Gamma, K \vdash te_2 : s \rightsquigarrow e_2 \qquad s \in T_1 \qquad n = \mathit{offsetForType}\,(s, T_1)\end{array}}{\Gamma, K \vdash te_1.te_2 : T_1(s) \rightsquigarrow \mathbf{cget}\ e_1\ n\ T_1(s)} \qquad (\text{TT\_C\_DOT\_ACCESS})$$

Here, if te_1 types to $\mathbf{ptr_L}\ T_1$, T_1 is a struct type, and te_2 types to a string literal s which is a field name in struct T_1, then the C struct member access types. Note that te_1 must be a Lua pointer to a C struct, as C structs themselves are not allowed in Poseidon Lua unless they are behind a Lua pointer. Also, the resulting **cget** is given the offset of field s in T_1 (determined with the *offsetForType* auxiliary function), since the C store lays out struct members linearly in an array form.

Second, C struct member update:

$$\frac{\begin{array}{c}\Gamma, K \vdash te_1 : \mathbf{ptr_L}\, T_1 \rightsquigarrow e_1 \qquad \mathit{structType}\,(T_1)\\ \Gamma, K \vdash te_2 : s \rightsquigarrow e_2 \qquad \Gamma, K \vdash te_3 : T_1(s) \rightsquigarrow e_3\\ s \in T_1 \qquad n = \mathit{offsetForType}\,(s, T_1)\end{array}}{\Gamma, K \vdash te_1.te_2 := te_3 : \mathbf{value} \rightsquigarrow \mathbf{cset}\ e_1\ n\ e_2\ T_1(s)} \qquad (\text{TT\_C\_DOT\_UPDATE})$$

As before, if te_1 is a Lua pointer to a C struct type T_1, and te_2 is a string s which is a member of that struct, and te_3 is appropriately typed, we can type the C struct update. We again emit an offset (in place of te_2), which the **cset** will use when writing to the C store.

Finally, Poseidon Lua allows C values to be downcast, and they type as follows:

$$\frac{\Gamma, K \vdash te : \mathbf{ptr_L}\, T'_C \rightsquigarrow e \qquad \beta\ \mathit{unused}}{\Gamma, K \vdash \mathbf{ccast}\, te\, T_C\ : T_C \rightsquigarrow \mathbf{ccast}\, e\, T_C\, \beta} \qquad (\text{TT\_C\_CAST})$$

Here, we notice that casting must be done through the Lua pointer, and so long as T_C is a C type we allow the cast to go through. There is no mention of T_C and T'_C being compatible types, as C freely allows casting of pointers, and the cast merely changes the way that the bits referred to by the pointer are read. As with previous mentions of β, it features here to allow errors caused by the cast to be easily traced back to the cast.

At this point, we have explored each of the typing rules associated with Poseidon Lua's C FFI. In many cases, such as in TT_C_FUN_APPL, these rules transferred some type information to their analogous run-time expressions in order to drive the run-time functionality of the system. We discuss reduction of run-time expressions next.

4.4 Operational Semantics

The *reduction relation* on untyped expressions, describing the execution of programs, is:

$$e \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V\ \rightarrow e' \,/\, \sigma'_T \,/\, \sigma'_C \,/\, \sigma'_V \qquad (2)$$

Here, e and e' are expressions in the untyped language, σ_T and σ'_T are *table stores*, σ_C and σ'_C are *C stores*, σ_V and σ'_V are *variable stores.* At a high level, the table store σ_T is a list of Lua tables, the variable store σ_V is a list of values, and finally the C store σ_C is a list of $(v, T_C, \beta?)$ triples, where v is a C value, T_C is its type, and $\beta?$ is optional taint information ($\emptyset$ represents no taint, or a clean location). As we mentioned in Section 3.2, the unusual inclusion of type information in the run-time C store is required to properly model C downcast semantics.

To simplify notation, we sometimes write the reduction relation as:

$$e \,/\, \mathcal{S} \rightarrow e' \,/\, \mathcal{S}' \qquad (3)$$

We refer to $\mathcal{S}$ and $\mathcal{S}'$ above as the *run-time environment*; the set of all the stores making up the state/context of the reduction.

It will be necessary to differentiate between C stores based on whether or not they are tainted; for this purpose, we say that a C store is *clean* if none of the elements of the store are themselves tainted. To simplify discussion of tainted environments, we say that a run-time environment is clean if its C store is also clean.

At the very highest level, we are formalizing a system wherein Lua code can interface with C in the following manner: allocating C data, reading from and writing to some shared memory with C, downcasting C values, and calling C functions.

Our formalization of Lua is based on FWLua [10], and we adapted their big-step semantics to a more standard small-step equivalent. For our discussion of FWLua, see Section 2.3.3. In order to mechanize our formalization, some simplifying modifications to FWLua were required, namely the promotion of variables from syntactic sugar to full-fledged language members. Of course, Lua allows you to declare and use variables, but FWLua desugars variables into access to a special store carried around at run-time. Poseidon Lua requires that FFI calls be made only from well-typed code, and so we adapted the type system of Typed Lua [14], with some modifications made possible by our simplified semantics for Lua.

Notable in Poseidon Lua is the merger of Typed Lua's and C's type systems through the Lua pointer type, and consequently the intermixing of values from both Lua and C. Lua makes reference to C values through the *Lua pointer* expression, and can both access and change the data contained in these pointers, as well as cast them to some C type. Lua may also allocate Lua pointers to C values through the **calloc** expression, without needing to make a **ccall**.

We will now turn our attention to the operational semantics of Poseidon Lua, with a focus on the C FFI, mirroring discussion of the typing judgment in Section 4.3. The full reduction rules are given in Appendix A.2. We start with the semantics of allocating C data. Consider:

$$\frac{\begin{array}{c} n = \mathit{length}\,(\sigma_C) \\ \sigma'_C = \sigma_C + \mathit{layoutTypeAndTaint}\,(T_C, \beta) \end{array}}{\mathbf{calloc}\, T_C\, \beta \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \;\to \mathbf{ptr}\, n\, T_C \,/\, \sigma_T \,/\, \sigma'_C \,/\, \sigma_V} \qquad (\text{R\_CALLOC})$$

The $\mathbf{calloc}\, T_C\, \beta$ expression allocates enough memory in the C store σ_C to accommodate a value of type T_C. The function *layoutTypeAndTaint* lays out type T_C and taints pointer members (as per our earlier discussion in Section 3.2). If T_C either is or contains a C pointer type, then we taint that location (with taint information β) to indicate to our system that its behavior is undefined until it is successfully accessed or written to. If T_C is a primitive or pointer type, then we simply produce a triplet containing a default value (this is `0` for pointers), the type T_C, and taint if T_C is a pointer type, and if T_C is a struct, we lay out each of its members in a similar fashion. Following allocation, a C pointer with the location of the beginning of the newly allocated memory is produced.

Compared with C allocation, C calls have intricate semantics as we do not attempt to model the bodies of arbitrary C functions. Instead, we treat the C functions like black boxes, and consequently C function calls exhibit *nondeterministic* semantics, as any well-typed C call can either succeed or fail if the function body is made up of arbitrary C code (recall that we consider a call successful if it returns to executing the host language with some value of the expected type). In the event of successful execution, we concern ourselves with the return value and the call's potential effects on the rest of the C data. Recall our discussion that even if a call is successful, the function code might have altered the C store in a variety of ways (such as freeing some existing memory), and we must account for this possibility.

We will first consider the reduction rule for a successful C call.

$$\frac{\mathit{value}\,(v_2) \qquad v = \mathit{makeValueOfType}\,(\mathit{ct}_2) \qquad \sigma'_C = \mathit{taintCStore}\,(\sigma_C, \beta)}{\textbf{ccall cfun}\ v_2\ \mathit{ct}_2\ \beta \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V\ \rightarrow v \,/\, \sigma_T \,/\, \sigma'_C \,/\, \sigma_V} \qquad \text{(R\_CCall\_Worked)}$$

Here, **ccall cfun** v_2 ct_2 β calls a C function **cfun** with argument v_2. In this case, the call succeeds, and $\mathit{makeValueOfType}\,(\mathit{ct}_2)$ gives us v, something of type ct_2. Of course, since it's possible that the call tampered with the C store, we taint the store with taint information β, corresponding to the line of code of this function call. This notifies subsequent accesses to these memory locations of potential tampering, which modifies the semantics of those accesses. C function calls can also fail:

$$\frac{\mathit{value}\,(v_2) \qquad \sigma'_C = \mathit{taintCStore}\,(\sigma_C, \beta)}{\textbf{ccall cfun}\ v_2\ \mathit{ct}_2\ \beta \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V\ \rightarrow \textbf{err}\ \beta \,/\, \sigma_T \,/\, \sigma'_C \,/\, \sigma_V} \qquad \text{(R\_CCall\_Failed)}$$

To capture that both success and failure are possible outcomes, we ensure that the premises of both rules are simultaneously satisfied: When all of R_CCall_Worked's preconditions are met, so are R_CCall_Failed's (and vice-versa). The **err** β expression is the result of the failing call, and indicates through taint information β which call is to blame for the failure.

Having seen the intricacies of C calls, we will turn our attention to the semantics of casting C pointers, another source of taint. For brevity, we only present the rule for casting a clean location (the other rule is not notably different). Consider:

$$\frac{\begin{array}{c} n < \mathit{length}(\sigma_C) \\ \sigma_C(n) = (v, T_C, \emptyset) \qquad \sigma'_C = \mathit{update}\,(\sigma_C, n, (v, T'_C, \beta)) \end{array}}{\textbf{ccast}\,(\textbf{ptr}_\mathbf{L}\, n\, T_C)\ T'_C\ \beta \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V\ \rightarrow \textbf{ptr}_\mathbf{L}\, n\, T'_C \,/\, \sigma_T \,/\, \sigma'_C \,/\, \sigma_V} \qquad \text{(R\_CCast)}$$

Here, the location n in σ_C is updated with the new type T'_C and taint information associated with the cast (thanks to the *update* auxiliary function – *update(s,l,v)* reads as "update s at location l with value v"). In C, casting a pointer merely changes how the bits being pointed to are read, and the cast may even cause an error; we achieve similar semantics with taint. When attempting to read location n in σ_C after it was cast, taint indicates that the access should be nondeterministic. To keep our system as general as possible, we don't attempt to model the cast per se, and the next read will replace v with a new value of type T'_C if successful, or fail with an error. We discuss the semantics of accesses next.

Thus far, we focused on the introduction of taint and fairly direct sources of nondeterminism, and we will turn our attention to taint's effect on the semantics of our system, as well as how it can be removed from the run-time environment. As an example, recall our semantics for C casts: When casting a location to some type T_C, the location becomes tainted. Now, imagine that the next use of the location is to store something of type T_C in it; if this write succeeds, from then on we are sure about the value present at the location. Such an operation is said to *clean* the taint from the location; in our formalization, taint represents uncertainty about a C value, and once we become certain of it (e.g., we have accessed the value and no errors have occurred) we can safely remove the taint.

In more formal terms, the presence of taint at a location in σ_C indicates that accessing that location yields nondeterministic results. To capture this, we ensure that a read or write to a tainted location can reduce to more than one expression *under the same premises*; namely, said read or write can succeed or fail.

Consider the following semantics for accessing a clean location in σ_C:

$$\frac{\sigma_C(n+o) = (v_C, T_C, \emptyset) \qquad v_{out} = coerceToLua(v_C)}{\mathbf{cget}\ (\mathbf{ptr_L}\ n\, T'_C)\ o\ T_C\ /\ \sigma_T\ /\ \sigma_C\ /\ \sigma_V\ \rightarrow v_{out}\ /\ \sigma_T\ /\ \sigma_C\ /\ \sigma_V} \qquad (\text{R\_C}\textsc{Get\_No\_Taint})$$

Here, the expression $\mathbf{cget}\ (\mathbf{ptr_L}\ n\, T'_C)\ o\, T_C$ accesses σ_C at location n with offset o, and is expecting something of type T_C. In this reduction rule, location $n + o$ in σ_C is clean, and so the (well-typed) store access cannot fail. The access steps to v_{out}, which is the Lua equivalent of the C value contained in σ_C, determined through the *coerceToLua* auxiliary function. Note that the pointer's type (T'_C) does not necessarily need to match the expected type of the access (T_C); this is because **cget**s can be used for struct member access, where T'_C would be a struct type and T_C would be the type of the member.

coerceToLua (v_C) is a function which takes a C value v and coerces it to a Lua value. If v_C is a C integer, then it is coerced to a Lua constant with the same numeric value. If v_C is a C pointer $\mathbf{ptr_C}\ m\ ct$, then it is coerced into a Lua pointer $\mathbf{ptr_{Lua}}\ m\ ct$ (to the same location). Otherwise, the coercion fails.

Note the presence of a type T_C in the **cget** expression. A condition of reading (and writing) from σ_C is that the type specified for the read must match the type held in σ_C. This allows us to enforce the correct use of downcast locations, as the cast changes the type in σ_C, and future reads (and writes) must specify the new type.

We will now consider accesses to tainted locations, which can either fail or succeed. First, consider a successful access:

$$\frac{\begin{array}{c}\sigma_C(n+o) = (v, T_C, \beta) \qquad v' = makeValueOfType\,(T_C) \\ \sigma'_C = update\,(\sigma_C, n+o, (v', T_C, \emptyset)) \qquad v_{out} = coerceToLua(v')\end{array}}{\mathbf{cget}\ (\mathbf{ptr}\, n\, T'_C)\, o\, T_C\ /\ \sigma_T\ /\ \sigma_C\ /\ \sigma_V\ \rightarrow v_{out}\ /\ \sigma_T\ /\ \sigma_C\ /\ \sigma_V}$$

(R_CGet_Taint_Works)

Here, we access σ_C at location n with offset o, and are expecting something of type T_C as before. However, $\sigma_C(n + o)$ is tainted, resulting in nondeterminism (i.e. we do not know whether an access to this value will fail or succeed). In this reduction rule, we deal with the case of a successful access to tainted locations. Here, a successful access returns some value of the appropriate type (thanks to the *makeValueOfType* auxiliary function). The C store at $n + o$ is cleaned and updated with the new value; from this moment on, use of this location is deterministic. Note that the value was observed to be *something* of type T_C, though not necessarily the same value that was in that location before the C call which initially necessitated the addition of the taint.

The following reduction rule deals with failing access:

$$\frac{\sigma_C(n+o) = (v, T_C, \beta)}{\mathbf{cget}\ (\mathbf{ptr}\, n\, T'_C)\, o\, T_C\ /\ \sigma_T\ /\ \sigma_C\ /\ \sigma_V\ \rightarrow \mathbf{err}\ \beta\ /\ \sigma_T\ /\ \sigma_C\ /\ \sigma_V} \qquad (\text{R\_C}\textsc{Get\_Taint\_Fails})$$

Here, the access fails, reporting the taint information identifying the call which tampered with this data. Note that satisfaction of rule R_CGet_Taint_Works's preconditions implies satisfaction of this rule's preconditions – this ensures that access to tainted locations can fail in any situation that it can succeed.

Similar to **cget**, **cset** has nondeterministic semantics when dealing with tainted locations. First, consider writes to clean locations:

$$\frac{\begin{array}{c}\sigma_C(n+o) = (v, T_C, \emptyset) \qquad value\,(v_2) \\ v_{put} = coerceToC(v_2) \qquad \sigma'_C = update\,(\sigma_C, n+o, (v_{put}, T_C, \emptyset))\end{array}}{\mathbf{cset}\ (\mathbf{ptr_L}\ n\, T'_C)\ o\ v_2\ T_C\ /\ \sigma_T\ /\ \sigma_C\ /\ \sigma_V\ \rightarrow v_2 /\ \sigma_T\ /\ \sigma'_C\ /\ \sigma_V} \qquad (\text{R\_C}\textsc{Set\_No\_Taint})$$

In the expression $\mathbf{cset}\,(\mathbf{ptr_L}\, n\, T'_C)\, o\, v_2\, T_C$, we write v_2 to location n with offset o in σ_C, and we expect the location to have type T_C. Since location $n + o$ in σ_C is clean, the store update cannot fail.

Note that we must first coerce v_2 to a C value v_{put} to store it in σ_C. $\mathit{coerceToC}(v_2)$ is similar to the *coerceToLua* function, though it coerces Lua values to C instead. For example, if v_2 is a numeric constant, the function produces a C integer with the same numeric value, and if v_2 is a Lua pointer $\mathbf{ptr_L}\ m\ ct$, an equivalent C pointer $\mathbf{ptr_C}\ m\ ct$ is produced.

The rule for **cset**s on tainted locations is given below:

$$\frac{\begin{array}{c}\sigma_C(n+o) = (v, T_C, \beta) \qquad \mathit{value}\,(v_2) \\ v_{put} = \mathit{coerceToC}(v_2) \qquad \sigma'_C = \mathit{update}\,(\sigma_C, n+o, (v_{put}, T_C, \emptyset))\end{array}}{\mathbf{cset}\ (\mathbf{ptr_L}\, n\, T'_C)\ o\ v_2\ T_C\ /\ \sigma_T\ /\ \sigma_C\ /\ \sigma_V\ \rightarrow v_2\ /\ \sigma_T\ /\ \sigma'_C\ /\ \sigma_V}$$

(R_CSet_Taint_Works)

Here, we again coerce v_2 to a C value v_{put} to location n with offset o in σ_C, and we expect the location to have type T_C. However, $\sigma_C(n + o)$ is tainted, and so we are in a state of nondeterminism. In rule R_CSet_Taint_Works, the write succeeds: We update $\sigma_C(n + o)$ with the new value v_{put} and clean the taint. Of course, failure is always an option:

$$\frac{\sigma_C(n+o) = (v, T_C, \beta)}{\mathbf{cset}\ (\mathbf{ptr_L}\, n\, T'_C)\ o\ v_2\ T_C\ /\ \sigma_T\ /\ \sigma_C\ /\ \sigma_V\ \rightarrow \mathbf{err}\ \beta\ /\ \sigma_T\ /\ \sigma_C\ /\ \sigma_V}$$

(R_CSet_Taint_Fails)

In this parallel case to T_CSet_Taint_Works, the write fails, and reports the taint information stored at $\sigma_C(n + o)$.

By now, we have explored each of the reduction relations related to Poseidon Lua's C FFI. In Section 3, we claimed that even without a model of C, as is the case in our system, the merger of the type systems of C and Typed Lua allows us to prove meaningful and interesting results about the language as a whole. The next section presents the results which we have proved, and sketches the proofs.

4.5 Proofs

There are two major results that we would like to prove about our semantics of Poseidon Lua. First, we would like to show some form of *soundness*, though clearly we can't have traditional type safety due to interoperation with C. Even so, we designed our semantics in such a way as to *track* C's effect on the overall system, and we can leverage that to show (conditional) soundness of the host language. Note that our proofs are mechanized in Coq, and this code in included in the artifact; a brief sketch of each proof is given here, but for the full details refer to the code.

We start with a sketch of preservation.

▶ **Theorem 1** (Preservation). *For all K, te, T, e, S, and S' such that $\{\}, K \vdash te : T \rightsquigarrow e$, $e\,/\,S \rightarrow e'\,/\,S'$, S is well-typed with respect to K, and both environments S and S' are clean, then there exists store typing K', typed expression te', and type T' such that $\{\}, K' \vdash te' : T' \rightsquigarrow e'$ with $T' <: T$, S' is well-typed with respect to K', and K' extends K.*

Proof sketch. Standard proof by induction on the typing derivation $\{\}, K \vdash te : T \rightsquigarrow e$. Any case where the error expression is reached is in violation of the run-time environments S and S' being clean, as taint is required in order to get an error. ◀

Essentially, the statement of preservation for Poseidon Lua differs from traditional statements of preservation in the stipulation that the run-time environments S and S' be clean. Clean environments ensure that the C error expression cannot be reached, and that the semantics are deterministic, as it's the presence of taint which begets nondeterminism.

We can similarly show progress.

▶ **Theorem 2** (Progress). *For all K, te, T, e, and S such that $\{\}, K \vdash te : T \leadsto e$, S is well-typed with respect to K, and S is clean, then either e is a value, or there exists clean environment S', and expression e' such that $e \,/\, S \to e' \,/\, S'$.*

Proof sketch. Another standard proof by induction on the typing derivation $\{\}, K \vdash te : T \leadsto e$. As with preservation, any case where the error expression is reached is in violation of the run-time environments S and S' being clean. ◀

As was the case in preservation, the statement of progress here is distinguished by the requirement that run-time environment S be clean. With a clean S', progress connects cleanly with preservation, allowing us to show soundness of Poseidon Lua contingent on clean environments. A sketch of soundness follows.

▶ **Theorem 3** (Soundness). *For all K, te, T, e, and S such that $\{\}, K \vdash te : T \leadsto e$, either e diverges, or there exists clean environment S', and value v such that $e \,/\, S \to^* v \,/\, S'$ and all intermediate environments are clean.*

Proof sketch. A standard proof, which basically amounts to applications of progress and preservation, and the intermediate environment of each step in the chain of reductions is guaranteed to be clean by construction (in a sense, progress generates clean environments). ◀

Roughly speaking, Theorem 3 states that Poseidon Lua programs in clean environments do not get stuck. The restriction to clean environments is due to the guest language, C, potentially interfering with the host language: C calls taint the environment, and accessing tainted values can lead to a stuck state even in well-typed programs. This isn't to say that you can't use C at all, as allocating simple pointers and structs does not taint the environment, and it is equally valid if some taint was once present and had been cleaned by successful accesses or writes.

So, what is the purpose of a soundness result such as the one presented here? What we have shown is that programs cannot go wrong if we don't venture into the C world, and proving this is a baseline and sanity check of sorts: It is difficult to discuss the semantics of the whole system if we do not at least know that one component of it is sound. Knowing that a taint-free system allows sound execution allows, for instance, the argument that one can recover sound execution by cleaning all tainted values out of the heap.

Unfortunately, our statement of soundness doesn't say much for the realistic use case of Poseidon Lua (and C FFIs in general), as these systems are designed to call C code. That said, we are not without options: as before, our inclusion of taint allows us to reason about C's effects on the overall language. Crucially, failing C reductions result in the error expression **err** β, and the taint information β can be used to identify the true culprit for the crash, even if that culprit was some earlier, seemingly unrelated expression. In short, we can show that C is to blame for failures in well-typed Poseidon Lua programs.

▶ **Theorem 4** (Always Blame C). *If the error expression **err** β is reached, then there exists some C expression which is to blame.*

```
p = calloc Point
cCall1(p)
cCall2(p)
cCall3(p)
print(p.x)
```

```
p = calloc Point
cCall1(p)
print(p.x)
cCall2(p)
print(p.x)
cCall3(p)
print(p.x)
```

Figure 4 Illustrative example.

Proof sketch. Effectively, this can be shown by construction of our semantics. **err** β can only be reached through reduction from a C expression, and the only way that such a reduction can occur is if there was some taint in the run-time environment. In **err** β, β is taint information which identifies some C call, cast, or allocation (as those are the only expressions which can taint), and it's the identified expression that will be blamed. ◀

At a high level, Theorem 4 indicates that run-time errors in well-typed Poseidon Lua are attributable to C. This signifies that our interoperation scheme does not allow for any additional errors which are the fault of the host language, and any errors introduced by the C FFI can be traced back to C.

Taken together, Theorems 3 and 4 are analogous to soundness of static code and the gradual guarantee in gradually typed languages [28][24], though the context is otherwise quite different. This similarity betrays a certain connection between gradual typing and language interoperation, a connection equally noted by aforementioned work on linking types [22].

As we know, program execution in a tainted environment is nondeterministic. In this state, many executions are possible, and they can be categorized as follows: the program either terminates successfully, terminates unsuccessfully, or it executes until the environment is cleaned of taint. Interestingly, executions which clean the taint actually *reclaim* soundness, and are deterministic at least until the next C call.

We can show one other interesting result about Poseidon Lua programs which call C. First, recall that only clean locations gain taint when a C call occurred; this ensures proper error tracking in the event of multiple C calls possibly tainting the same data. For an illustrative example, consider the code in Figure 4.

Assume the leftmost program fails at the access to `p.x`, blaming `cCall1` and identifying it as the start of our search; here, we cannot say for sure which of `cCall1`, `cCall2`, or `cCall3` mucked with `p.x`. However, we can generate a modified program which can isolate the faulty C call. Consider the snippet on the right. If `cCall1` was the culprit of the failure, then the access immediately following it will fail. If not, and `cCall2` was at fault, then the access immediately after `cCall2` will fail. If neither of these are true, then `cCall3` is at fault, causing the final access to `p.x` to fail. This amounts to fault localization: When we are uncertain about which of a number of unsafe operations are at fault for a run-time failure, we can generate a new program which isolates the faulty operation.

5 Poseidon Lua: Implementation

As a demonstration of the practicality of these semantics, they have been implemented as modifications to Lua 5.3.3 [13] and Typed Lua [14]. Lua is extended to provide low-level interfaces, and Typed Lua is extended to make use of them with C types. The extensions to Lua have no guarantees of safety or correctness on their own, and are treated as an internal implementation language for the modifications of Typed Lua. Typed Lua is extended with C types, through the addition of a C pointer in Lua which refers to C data (as explained in Section 4.1).

Typed Lua's grammar is extended as follows:

T ::= *(all existing Typed Lua types)* | **PtrType**
PtrType ::= `ptr` `ptr`* **PtrTargetType**
PtrTargetType ::= **CVoidType** | **CPrimitiveType** | **Name**
CType ::= **CPrimitiveType** | **PtrType**
CVoidType ::= `void`
CPrimitiveType ::= `char` | `int` | `double`
Statement ::= *(all existing Typed Lua statements)* | **StructDeclaration**
StructDeclaration ::= `struct` **Name StructIdDecList** `end`
StructIdDecList ::= **StructIdDec StructIdDec***
StructIdDec ::= **Id** `:` **CType**
Expression ::= *(all existing Typed Lua expressions)* | **CallocExpr**
CallocExpr ::= `calloc` `(` **PtrTargetType** `)`

T, in particular, is the existing Typed Lua non-terminal for types. As a consequence, any variable, parameter or field in Poseidon Lua may contain a *pointer* to a C value, but may not contain a C value directly. All other types are unmodified, and behave as they do in Typed Lua. As in C, the Poseidon Lua compiler assures that every type named in a C pointer type has a corresponding struct declaration, and that no name corresponds to multiple structure declarations, and as in C, the struct declaration defines the memory layout of objects of that type. Unlike in C, declarations are not required to precede uses of the type they declare. A simple wrapper for `calloc` is provided to assure that allocations are always of the correct size. For this prototype, we implemented only `char`s, `int`s and `double`s, but there is no conceptual limitation on implementing any other primitive type. For convenience, Poseidon Lua also provides syntax and semantics for C arrays, but they are not discussed in this work.

This modified Typed Lua compiles to Lua, extended with intrinsics to manipulate memory directly. Typed Lua code which doesn't use C features is unchanged: That is, if C `ptr`s are not used, `calloc` is not used, and the code passes type checking, then it compiles into identical Lua code without type annotations or declarations (i.e. the types are erased). Lua already provides a datatype, "light user data", intended for storing pointers to C data, and this datatype is used for all `ptr`-typed variables and fields. This is why Lua was used for this prototype. However, Lua's light user data is completely opaque to Lua code: In order to use it, one must implement a C interface, from which the underlying pointers are exposed. Our principle extensions to Lua are low-level operators to directly manipulate memory through these pointers: `CS_loadChar`, `CS_storeChar`, and similar for ints, doubles and pointers. In addition, `CS_calloc` and `CS_free` are provided to give direct access to C's `calloc` and `free`, a literal `CS_NULL` corresponding to C's `NULL` is provided to check for errors, and `CS_loadString` and `CS_storeString` are provided to convert between C strings

```
struct House
    num_rooms : int
end
local house_1 : ptr House = calloc(House)
house_1.num_rooms = 6
```

Figure 5 Simple Poseidon Lua code example.

```
local house_1 = CS_calloc(4)
CS_storeInt(house_1, 0, 6)
```

Figure 6 Simple Lua code example compiled from Poseidon Lua.

(0-terminated `char` arrays) and Lua strings. “CS” in this context is an abbreviation of “C Semantics”.

Each of these low-level operators converts data between Lua’s native data types and C’s, given a C pointer stored in a Lua light user data, and an offset. The conversions themselves are trivial. None of these operators are intended for direct use by end users. Instead, Poseidon Lua’s Typed Lua implementation compiles code which uses C types – that is, code which accesses members of `ptr`-typed variables or fields – to Lua which uses the correct operators. Internally, each low-level operator is compiled to its own opcode in Lua’s bytecode.

As a simple example, the Poseidon Lua in Figure 5 compiles to the Lua in Figure 6.

As the changes in our semantics are concerned principally with C data, rather than C functions, we use a modified luaffifb for the function component of the interface. Poseidon Lua’s modified luaffifb is changed only by replacing their wrapper objects with Lua’s light user data, which can then be handled by Typed Lua types. The jump between C and Lua code incurs much less overhead than wrapping C data for use in Lua, so no further modifications are necessary.

5.1 Performance

Poseidon Lua code which doesn’t use C types is just regular Typed Lua: when compiled into Lua code this will be identical to the equivalent Typed Lua program being compiled into Lua, and so will not display any performance difference. Thus, to compare the performance of Poseidon Lua against luaffifb, we need benchmarks which particularly measure the access to structured data. Unfortunately, we know of no benchmark suite intended specifically for this purpose, so instead we ported four benchmarks from the Computer Language Benchmarks Game [5]. The subset of benchmarks from CLBG were selected because they had Lua versions which used structured data types. In each case, they were rewritten so that every structured datatype used a C struct, the shape of which was taken from the C version of each benchmark. In Poseidon Lua, these structs were represented as `struct` declarations, and in luaffifb, as their dynamic declarations. In both cases, no actual C calls are made: The data is stored in C-compatible structures, and accessed through them, but the benchmark code is entirely Lua. We compare the performance of luaffifb, which uses wrappers, to Poseidon Lua, which does not. We also include the original Lua benchmark, which does not use C structured data, for reference, although we expect no significant performance difference with respect to it. The results and standard deviations are shown in Table 1. As expected, Poseidon Lua shows a substantial speedup over luaffifb, due to the absence of allocated wrappers at run-time.

Table 1 Comparison of performance results over various benchmarks.

	Poseidon Lua		luaffifb		Lua	
Benchmark	Time (s)	Std. Dev.	Time (s)	Std. Dev.	Time (s)	Std. Dev.
binary-trees	18.8	0.447	202.4	2.97	22.0	0.707
n-body	4.0	0	40.6	1.14	4.0	0.707
spectral-norm	108.2	0	270.8	2.59	105.6	0.894
fannkuch-redux	66.8	2.95	528.8	9.68	55.0	0

Our performance is close to original Lua, though in some benchmarks the cost of converting between C's primitive types and Lua's overwhelms other benefits.

The benchmarks were performed on Lua 5.3.3 as well as our modified version thereof, on a quad-core 1.8GHz 64-bit Intel desktop PC running Ubuntu 14.04.3LTS.

6 Conclusions

In this paper, we presented a framework for reasoning about C FFIs without fully modelling the guest language. This framework relies on making the data interface of the FFI static by combining the type systems of the host and guest languages, and doesn't require a model of the guest language beyond its direct interactions with the host. We also saw how making the data interface static eliminates the need for burdensome wrappers in FFI implementations, as the host language can statically check its own use of the FFI instead of needing to rely on the dynamic checks in the wrappers.

To showcase our framework, we presented Poseidon Lua, a Typed Lua C FFI. We gave the formal semantics of the C FFI in Poseidon Lua, and even without modelling C were able to guarantee some level of soundness of the host language, as well as prove that well-typed host language code is not to blame for errors that occur. We also presented an implementation of Poseidon Lua, and confirmed that making the data interface static does indeed improve the performance of the FFI.

While we focus on a C FFI, in principle our approach also works for other choices of guest language, as we deliberately avoid modelling C. That said, our model of C's memory and C's types in the host language make languages with similar memory behavior to C's most suitable, though one could plug in any type system and model memory differently if they are so inclined. We focused on a C FFI because they are very common, and prove particularly challenging to reason about with traditional methods.

References

1 M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic Typing in a Statically-typed Language. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, pages 213–227, New York, NY, USA, 1989. ACM. doi:10.1145/75277.75296.

2 CompCert. CompCert Main Page. http://compcert.inria.fr. Accessed: 2018-07-23.

3 Facebook. luaffifb. https://github.com/facebookarchive/luaffifb. Accessed: 2019-01-10.

4 Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics engineering with PLT Redex*. Mit Press, 2009.

5 Isaac Gouy. The Computer Language Benchmarks Game. https://benchmarksgame-team.pages.debian.net/benchmarksgame/. Accessed: 2019-01-10.

6 Kathryn E Gray. Safe cross-language inheritance. In *European Conference on Object-Oriented Programming*, pages 52–75. Springer, 2008.

7 Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. The essence of JavaScript. In *European Conference on Object-Oriented Programming*, pages 126–150. Springer, 2010.

8 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.

9 Julia. Calling C and Fortran Code. `https://docs.julialang.org/en/stable/manual/calling-c-and-fortran-code/index.html`. Accessed: 2018-07-14.

10 Hanshu Lin. Operational semantics for Featherweight Lua. *Master's Projects*, page 387, 2015.

11 Lisp. CFFI The Common Foreign Function Interface. `https://common-lisp.net/project/cffi/`. Accessed: 2018-07-25.

12 Tidal Lock. Tidal Lock Gradual Static Typing for Lua. `https://github.com/fab13n/metalua/tree/tilo/src/tilo`. Accessed: 2018-06-20.

13 Lua. Lua 5.3 Documentation. `https://www.lua.org/manual/5.3/`. Accessed: 2018-06-20.

14 André Murbach Maidl, Fabio Mascarenhas, and Roberto Ierusalimschy. Typed Lua: An optional type system for Lua. In *Proceedings of the Workshop on Dynamic Languages and Applications*, pages 1–10. ACM, 2014.

15 MathWorks. Matlab Calling C Shared Libraries. `https://www.mathworks.com/help/matlab/using-c-shared-library-functions-in-matlab-.html`. Accessed: 2018-07-04.

16 Jacob Matthews and Robert Bruce Findler. Operational semantics for multi-language programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 31(3):12, 2009.

17 Microsoft. TypeScript – Language Specification Version 1.8. Technical report, Microsoft, January 2016.

18 James Newsome and Dawn Xiaodong Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *NDSS*, volume 5, pages 3–4. Citeseer, 2005.

19 Graham Ollis. Perl Perl Foreign Function Interface based on GNU ffcall. `https://metacpan.org/pod/FFI`. Accessed: 2018-07-06.

20 Oracle. JNI specification. `https://docs.oracle.com/javase/8/docs/technotes/guides/jni/spec/jniTOC.html`. Accessed: 2019-06-10.

21 Mike Pall. The LuaJIT Project. *Web site: http://luajit. org*, 2008.

22 Daniel Patterson and Amal Ahmed. Linking Types for Multi-Language Software: Have Your Cake and Eat It Too. In *2nd Summit on Advances in Programming Languages (SNAPL 2017)*, volume 71 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 12:1–12:15, 2017. `doi:10.4230/LIPIcs.SNAPL.2017.12`.

23 Python. CFFI Documentation. `https://cffi.readthedocs.io/en/latest/`. Accessed: 2018-07-06.

24 Jeremy G Siek, Michael M Vitousek, Matteo Cimini, and John Tang Boyland. Refined criteria for gradual typing. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 32. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

25 Mallku Soldevila, Beta Ziliani, Bruno Silvestre, Daniel Fridlender, and Fabio Mascarenhas. Decoding Lua: Formal Semantics for the Developer and the Semanticist. In *Proceedings of the 13th ACM SIGPLAN International Symposium on Dynamic Languages*, DLS 2017, pages 75–86, New York, NY, USA, 2017. ACM. `doi:10.1145/3133841.3133848`.

26 SWIG Team. SWIG. `swig.org`. Accessed: 2019-06-10.

27 Sam Tobin-Hochstadt, Vincent St-Amour, Eric Dobson, and Asumu Takikawa. Typed Racket Documentation. `https://docs.racket-lang.org/ts-guide/`. Accessed: 2018-08-01.

28 Philip Wadler and Robert Bruce Findler. Well-typed programs can't be blamed. In *European Symposium on Programming*, pages 1–16. Springer, 2009.

29 Alon Zakai. Emscripten: an LLVM-to-JavaScript compiler. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pages 301–312. ACM, 2011.

A Appendix

A.1 Full Typing Rules

$$\frac{\mathit{validType}(T_C) \qquad \beta\ \mathit{unused}}{\Gamma, K \vdash \textbf{calloc}\ T_C\ : \textbf{ptr}_\textbf{L}\, T_C \leadsto \textbf{calloc}\ T_C\ \beta} \qquad (\text{TT\_}\textsc{CAlloc})$$

$$\frac{\begin{array}{c} n < \mathit{length}(\Sigma_C) \\ \mathit{goodLayout}(n, T, \Sigma_C) \end{array}}{\Gamma, (\Sigma_T, \Sigma_C, \Sigma_V) \vdash \textbf{ptr}_\textbf{L}\, n\, T_C : \textbf{ptr}_\textbf{L}\, T_C \leadsto \textbf{ptr}_\textbf{L}\, n\, T_C} \qquad (\text{TT\_}\textsc{Lua\_Ptr})$$

$$\frac{\begin{array}{c} \Gamma, K \vdash \mathit{te} : \textbf{ptr}_\textbf{L}\, T_C \leadsto e \\ \mathit{validForCDeref}(T_C) \qquad T_L = \mathit{coerceCType}(T_C) \end{array}}{\Gamma, K \vdash \textbf{deref}_\textbf{C}\ \mathit{te} : T_L \leadsto \textbf{cget}\ e\ 0\ T_C} \qquad (\text{TT\_}\textsc{Var\_C\_Deref})$$

$$\frac{}{\Gamma, K \vdash \textbf{cfun}\, (\mathit{ct}_1 \to_C \mathit{ct}_2) : (\mathit{ct}_1 \to_C \mathit{ct}_2) \leadsto \textbf{cfun}} \qquad (\text{TT\_}\textsc{C\_Function})$$

$$\frac{\begin{array}{c} \Gamma, K \vdash \mathit{te}_1 : (T \to_C T') \leadsto e_1 \\ \Gamma, K \vdash \mathit{te}_2 : T \leadsto e_2 \qquad \beta\ \mathit{unused} \end{array}}{\Gamma, K \vdash \mathit{te}_1(\mathit{te}_2) : T' \leadsto \textbf{ccall}\, e_1\ e_2\, T'\ \beta} \qquad (\text{TT\_}\textsc{C\_Fun\_Appl})$$

$$\frac{\begin{array}{c} \Gamma, K \vdash \mathit{te}_1 : \textbf{ptr}_\textbf{L}\, T_1 \leadsto e_1 \qquad \mathit{structType}(T_1) \\ \Gamma, K \vdash \mathit{te}_2 : s \leadsto e_2 \qquad s \in T_1 \qquad n = \mathit{offsetForType}\,(s, T_1) \end{array}}{\Gamma, K \vdash \mathit{te}_1.\mathit{te}_2 : T_1(s) \leadsto \textbf{cget}\ e_1\ n\ T_1(s)} \qquad (\text{TT\_}\textsc{C\_Dot\_Access})$$

$$\frac{\begin{array}{c} \Gamma, K \vdash \mathit{te}_1 : \textbf{ptr}_\textbf{L}\, T_1 \leadsto e_1 \qquad \mathit{structType}\,(T_1) \\ \Gamma, K \vdash \mathit{te}_2 : s \leadsto e_2 \qquad \Gamma, K \vdash \mathit{te}_3 : T_1(s) \leadsto e_3 \\ s \in T_1 \qquad n = \mathit{offsetForType}\,(s, T_1) \end{array}}{\Gamma, K \vdash \mathit{te}_1.\mathit{te}_2 := \mathit{te}_3 : \textbf{value} \leadsto \textbf{cset}\ e_1\ n\ e_2\ T_1(s)} \qquad (\text{TT\_}\textsc{C\_Dot\_Update})$$

$$\frac{\Gamma, K \vdash \mathit{te} : \textbf{ptr}_\textbf{L}\, T'_C \leadsto e \qquad \beta\ \mathit{unused}}{\Gamma, K \vdash \textbf{ccast}\, \mathit{te}\, T_C\ : T_C \leadsto \textbf{ccast}\, e\, T_C\ \beta} \qquad (\text{TT\_}\textsc{C\_Cast})$$

$$\frac{\forall i,\ \ f_i = s_i : T_i \vee f_i = \textbf{const}\, s_i : T_i \qquad \forall i,\ \ \Gamma, K \vdash \mathit{tv}_i : T_i \leadsto v_i}{\Gamma, K \vdash \{s_1 = \mathit{tv}_1, ..., s_n = \mathit{tv}_n\} : \{f_1, ..., f_n\} \leadsto \{s_1 = v_1, ..., s_n = v_n\}} \qquad (\text{TT\_}\textsc{Table})$$

$$\frac{\begin{array}{c} \Gamma, K \vdash \mathit{te}_1 : T \leadsto e_1 \\ \Gamma + \{x \mapsto T\}, K \vdash \mathit{te}_2 : T' \leadsto e_2 \end{array}}{\Gamma, K \vdash \textbf{let}\, x : T := \mathit{te}_1\, \textbf{in}\, \mathit{te}_2 : T' \leadsto \textbf{let}\, x := e_1\, \textbf{in}\, e_2} \qquad (\text{TT\_}\textsc{Let})$$

$$\frac{x \in \Gamma}{\Gamma, K \vdash x : \Gamma(x) \leadsto x} \qquad (\text{TT\_}\textsc{Var})$$

$$\frac{n < \mathit{length}(\Sigma_T)}{\Gamma, (\Sigma_T, \Sigma_C, \Sigma_V) \vdash \mathbf{reg}\, n : \Sigma_T(n) \leadsto \mathbf{reg}\, n} \qquad (\text{TT\_}\textsc{Reg})$$

$$\frac{\Gamma, K \vdash \mathit{te} : \mathbf{ref}\, T \leadsto e}{\Gamma, K \vdash \mathbf{deref}\, \mathit{te} : T \leadsto \mathbf{deref}\, e} \qquad (\text{TT\_}\textsc{Var\_Deref})$$

$$\frac{\begin{array}{c} x \in \Gamma \\ \Gamma, K \vdash \mathit{te} : \Gamma(x) \leadsto e \end{array}}{\Gamma, K \vdash x := \mathit{te} : T \leadsto x := e} \qquad (\text{TT\_}\textsc{Var\_Assign})$$

$$\frac{\Gamma, (\Sigma_T, \Sigma_C, \Sigma_V) \vdash \mathit{te} : \Sigma_V(n) \leadsto e}{\Gamma, (\Sigma_T, \Sigma_C, \Sigma_V) \vdash \mathbf{loc}\, n := \mathit{te} : T \leadsto \mathbf{loc}\, n := e} \qquad (\text{TT\_}\textsc{Loc\_Update})$$

$$\frac{\Gamma + \{x \mapsto T\}, K \vdash \mathit{te} : T' \leadsto e}{\Gamma, K \vdash \lambda x : T.\mathit{te} : (T \to_L T') \leadsto \lambda x.e} \qquad (\text{TT\_}\textsc{Function})$$

$$\frac{\begin{array}{c} \Gamma, K \vdash \mathit{te}_1 : (T \to_L T') \leadsto e_1 \\ \Gamma, K \vdash \mathit{te}_2 : T \leadsto e_2 \end{array}}{\Gamma, K \vdash \mathit{te}_1(\mathit{te}_2) : T' \leadsto e_1(e_2)} \qquad (\text{TT\_}\textsc{Lua\_Fun\_Appl})$$

$$\frac{\begin{array}{c} \Gamma, K \vdash \mathit{te}_1 : T_1 \leadsto e_1 \qquad \mathit{tableType}(T_1) \\ \Gamma, K \vdash \mathit{te}_2 : s \leadsto e_2 \qquad s \in T_1 \end{array}}{\Gamma, K \vdash \mathit{te}_1.\mathit{te}_2 : T_1(s) \leadsto \mathbf{rawget}\, e_1\, e_2} \qquad (\text{TT\_}\textsc{Dot\_Access})$$

$$\frac{\begin{array}{c} \Gamma, K \vdash \mathit{te}_1 : T_1 \leadsto e_1 \qquad \mathit{tableType}(T_1) \\ \Gamma, K \vdash \mathit{te}_2 : s \leadsto e_2 \qquad s \in T_1 \\ \Gamma, K \vdash \mathit{te}_3 : T_1(s) \leadsto e_3 \end{array}}{\Gamma, K \vdash \mathit{te}_1.\mathit{te}_2 := \mathit{te}_3 : \mathbf{value} \leadsto \mathbf{rawset}\, e_1\, e_2\, e_3} \qquad (\text{TT\_}\textsc{Dot\_Update})$$

$$\frac{\Gamma, K \vdash \mathit{te} : T \leadsto e \qquad T <: T'}{\Gamma, K \vdash \mathit{te} : T' \leadsto e} \qquad (\text{TT\_}\textsc{Subsumption})$$

$$\frac{c \text{ constant}}{\Gamma, K \vdash c : c \leadsto c} \qquad (\text{TT\_}\textsc{Const})$$

$$\frac{\begin{array}{c} \Gamma, K \vdash \mathit{te}_1 : \mathbf{number} \leadsto e_1 \qquad \Gamma, K \vdash \mathit{te}_2 : \mathbf{number} \leadsto e_2 \\ \mathit{op} \in \{+, -, *, /\} \end{array}}{\Gamma, K \vdash \mathit{te}_1\, \mathit{op}\, \mathit{te}_2 : \mathbf{number} \leadsto e_1\, \mathit{op}\, e_2} \qquad (\text{TT\_}\textsc{Binop\_Arith})$$

$$\frac{\begin{array}{c}\Gamma, K \vdash te_1 : \textbf{number} \leadsto e_1 \qquad \Gamma, K \vdash te_2 : \textbf{number} \leadsto e_2 \\ op \in \{<, \leq, >, \geq\}\end{array}}{\Gamma, K \vdash te_1\, op\, te_2 : \textbf{boolean} \leadsto e_1\, op\, e_2} \qquad \text{(TT\_Binop\_Order)}$$

$$\frac{\begin{array}{c}\Gamma, K \vdash te_1 : \textbf{boolean} \leadsto e_1 \qquad \Gamma, K \vdash te_2 : \textbf{boolean} \leadsto e_2 \\ op \in \{\wedge, \vee\}\end{array}}{\Gamma, K \vdash te_1\, op\, te_2 : \textbf{boolean} \leadsto e_1\, op\, e_2} \qquad \text{(TT\_Binop\_Bools)}$$

$$\frac{\begin{array}{c}\Gamma, K \vdash te_1 : \textbf{string} \leadsto e_1 \\ \Gamma, K \vdash te_2 : T_2 \leadsto e_2 \\ T_2 \in \{\textbf{string}, \textbf{number}\}\end{array}}{\Gamma, K \vdash te_1 \mathbin{..} te_2 : \textbf{string} \leadsto e_1 \mathbin{..} e_2} \qquad \text{(TT\_Binop\_String)}$$

$$\frac{\begin{array}{c}\Gamma, K \vdash te_1 : T_1 \leadsto e_1 \\ \Gamma, K \vdash te_2 : T_2 \leadsto e_2\end{array}}{\Gamma, K \vdash te_1 \;==\; te_2 : \textbf{boolean} \leadsto e_1 \;==\; e_2} \qquad \text{(TT\_Binop\_Eq)}$$

$$\frac{\begin{array}{c}\Gamma, K \vdash te_1 : T_1 \leadsto e_1 \\ \Gamma, K \vdash te_2 : T_2 \leadsto e_2\end{array}}{\Gamma, K \vdash te_1;\, te_2 : T_2 \leadsto e_1;\, e_2} \qquad \text{(TT\_Sequence)}$$

A.2 Full Reduction Rules

$$\frac{\begin{array}{c}n = \mathit{length}\,(\sigma_C) \\ \sigma'_C = \sigma_C + \mathit{layoutTypeAndTaint}\,(T_C, \beta)\end{array}}{\textbf{calloc}\, T_C\, \beta \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \;\rightarrow \textbf{ptr}\, n\, T_C \,/\, \sigma_T \,/\, \sigma'_C \,/\, \sigma_V} \qquad \text{(R\_CAlloc)}$$

$$\frac{\mathit{value}\,(v_2) \qquad v = \mathit{makeValueOfType}\,(ct_2) \qquad \sigma'_C = \mathit{taintCStore}\,(\sigma_C, \beta)}{\textbf{ccall cfun}\; v_2\; ct_2\; \beta \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \;\rightarrow v \,/\, \sigma_T \,/\, \sigma'_C \,/\, \sigma_V} \qquad \text{(R\_CCall\_Worked)}$$

$$\frac{\mathit{value}\,(v_2) \qquad \sigma'_C = \mathit{taintCStore}\,(\sigma_C, \beta)}{\textbf{ccall cfun}\; v_2\; ct_2\, \beta \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \;\rightarrow \textbf{err}\, \beta \,/\, \sigma_T \,/\, \sigma'_C \,/\, \sigma_V} \qquad \text{(R\_CCall\_Failed)}$$

$$\frac{\begin{array}{c}n < \mathit{length}(\sigma_C) \\ \sigma_C\,(n) = (v, T_C, \emptyset) \qquad \sigma'_C = \mathit{update}\,(\sigma_C, n, (v, T'_C, \beta))\end{array}}{\textbf{ccast}\; (\textbf{ptr}_\mathbf{L}\, n\, T_C)\; T'_C\; \beta \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \;\rightarrow \textbf{ptr}_\mathbf{L}\, n\, T'_C \,/\, \sigma_T \,/\, \sigma'_C \,/\, \sigma_V} \qquad \text{(R\_CCast)}$$

$$\frac{\sigma_C(n + o) = (v_C, T_C, \emptyset) \qquad v_{out} = \mathit{coerceToLua}(v_C)}{\textbf{cget}\; (\textbf{ptr}_\mathbf{L}\, n\, T'_C)\; o\; T_C \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \;\rightarrow v_{out} \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V} \qquad \text{(R\_CGet\_No\_Taint)}$$

$$\frac{\begin{array}{c}\sigma_C(n+o) = (v, T_C, \beta) \qquad v' = \mathit{makeValueOfType}\,(T_C) \\ \sigma'_C = \mathit{update}\,(\sigma_C, n+o, (v', T_C, \emptyset) \qquad v_{out} = \mathit{coerceToLua}(v'))\end{array}}{\textbf{cget}\,(\textbf{ptr}\,n\,T'_C)\,o\,T_C \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \;\rightarrow v_{out} \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V}$$

(R_CGet_Taint_Works)

$$\frac{\sigma_C(n+o) = (v, T_C, \beta)}{\textbf{cget}\,(\textbf{ptr}\,n\,T'_C)\,o\,T_C \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \;\rightarrow \textbf{err}\;\beta \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V}$$

(R_CGet_Taint_Fails)

$$\frac{\begin{array}{c}\sigma_C(n+o) = (v, T_C, \emptyset) \qquad \mathit{value}\,(v_2) \\ v_{put} = \mathit{coerceToC}(v_2) \qquad \sigma'_C = \mathit{update}\,(\sigma_C, n+o, (v_{put}, T_C, \emptyset))\end{array}}{\textbf{cset}\;(\textbf{ptr}_\mathbf{L}\,n\,T'_C)\;o\;v_2\;T_C \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \;\rightarrow v_2 / \, \sigma_T \,/\, \sigma'_C \,/\, \sigma_V}$$

(R_CSet_No_Taint)

$$\frac{\begin{array}{c}\sigma_C(n+o) = (v, T_C, \beta) \qquad \mathit{value}\,(v_2) \\ v_{put} = \mathit{coerceToC}(v_2) \qquad \sigma'_C = \mathit{update}\,(\sigma_C, n+o, (v_{put}, T_C, \emptyset))\end{array}}{\textbf{cset}\;(\textbf{ptr}_\mathbf{L}\,n\,T'_C)\;o\;v_2\;T_C \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \;\rightarrow v_2 \,/\, \sigma_T \,/\, \sigma'_C \,/\, \sigma_V}$$

(R_CSet_Taint_Works)

$$\frac{\sigma_C(n+o) = (v, T_C, \beta)}{\textbf{cset}\;(\textbf{ptr}_\mathbf{L}\,n\,T'_C)\;o\;v_2\;T_C \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \;\rightarrow \textbf{err}\;\beta \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V}$$

(R_CSet_Taint_Fails)

$$\frac{n = \mathit{length}\,(\sigma_T) \qquad t_n = \mathit{buildTable}(\{s_1 = v_1, ..., s_n = v_n\})}{\{s_1 = v_1, ..., s_n = v_n\} \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \;\rightarrow\; (\textbf{reg}\;n) \,/\, \sigma_T + t_n \,/\, \sigma_C \,/\, \sigma_V}$$

(R_Table)

$$\frac{\mathit{value}\,(e_1) \qquad l = \mathit{length}(\sigma_V)}{\textbf{let}\,x := e_1\,\textbf{in}\,e_2 \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \;\rightarrow [x \leftarrow l]\,e_2 \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V + e_1}$$

(R_Let)

$$\frac{\mathit{value}\,(e_2) \qquad l = \mathit{length}(\sigma_V)}{(\lambda x.e)(e_2) \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \;\rightarrow [x \leftarrow l]\,e \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V + e_2}$$

(R_Fun_App)

$$\frac{\sigma_V(l) = v \qquad \mathit{value}(v)}{\textbf{deref}\;(\textbf{loc}\,l) \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \;\rightarrow v \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V}$$

(R_Loc_Deref)

$$\frac{\mathit{value}\,(e) \qquad \sigma'_V = \mathit{update}\,(\sigma_V, l, e)}{\textbf{loc}\,l := e \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \;\rightarrow e \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma'_V}$$

(R_Loc_Update)

$$\frac{\sigma_T(n) = T \qquad T(s) = v}{\textbf{rawget}\,(\textbf{reg}\,n)\,s \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \;\rightarrow v \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V} \qquad \text{(R\_\textsc{Rawget})}$$

$$\frac{\begin{array}{c} \mathit{value}\,(e_3) \qquad \sigma_T(n) = T \qquad s \in T \\ T' = \mathit{update}\,(T, s, e_3) \qquad \sigma'_T = \mathit{update}\,(\sigma_T, n, T') \end{array}}{\textbf{rawset}\,(\textbf{reg}\,n)\,s\,e_3 \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \;\rightarrow \textbf{reg}\,n \,/\, \sigma'_T \,/\, \sigma_C \,/\, \sigma_V} \qquad \text{(R\_\textsc{Rawset})}$$

$$\frac{e \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \;\rightarrow e' \,/\, \sigma'_T \,/\, \sigma'_C \,/\, \sigma'_V}{x := e \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \;\rightarrow x := e' \,/\, \sigma'_T \,/\, \sigma'_C \,/\, \sigma'_V} \qquad \text{(R\_\textsc{Var\_Assign\_Step}\_1)}$$

$$\frac{e \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \,/ \rightarrow e' \,/\, \sigma'_T \,/\, \sigma'_C \,/\, \sigma'_V}{\textbf{loc}\,l := e \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \;\rightarrow \textbf{loc}\,l := e' \,/\, \sigma'_T \,/\, \sigma'_C \,/\, \sigma'_V} \qquad \text{(R\_\textsc{Loc\_Update\_Step}\_1)}$$

$$\frac{e_1 \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \;\rightarrow e'_1 \,/\, \sigma'_T \,/\, \sigma'_C \,/\, \sigma'_V}{\textbf{let}\,x := e_1\,\textbf{in}\,e_2 \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \;\rightarrow \textbf{let}\,x := e'_1\,\textbf{in}\,e_2 \,/\, \sigma'_T \,/\, \sigma'_C \,/\, \sigma'_V} \qquad \text{(R\_\textsc{Let\_Step})}$$

$$\frac{e_1 \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \;\rightarrow e'_1 \,/\, \sigma'_T \,/\, \sigma'_C \,/\, \sigma'_V}{\textbf{rawget}\,e_1\,e_2 \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \;\rightarrow \textbf{rawget}\,e'_1\,e_2 \,/\, \sigma'_T \,/\, \sigma'_C \,/\, \sigma'_V} \qquad \text{(R\_\textsc{Rawget\_Step}\_1)}$$

$$\frac{\begin{array}{c} \mathit{value}\,(e_1) \\ e_2 \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \;\rightarrow e'_2 \,/\, \sigma'_T \,/\, \sigma'_C \,/\, \sigma'_V \end{array}}{\textbf{rawget}\,e_1\,e_2 \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \;\rightarrow \textbf{rawget}\,e_1\,e'_2 \,/\, \sigma'_T \,/\, \sigma'_C \,/\, \sigma'_V} \qquad \text{(R\_\textsc{Rawget\_Step}\_2)}$$

$$\frac{e_1 \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \;\rightarrow e'_1 \,/\, \sigma'_T \,/\, \sigma'_C \,/\, \sigma'_V}{\textbf{rawset}\,e_1\,e_2\,e_3 \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \;\rightarrow \textbf{rawset}\,e'_1\,e_2\,e_3 \,/\, \sigma'_T \,/\, \sigma'_C \,/\, \sigma'_V} \qquad \text{(R\_\textsc{Rawset\_Step}\_1)}$$

$$\frac{\begin{array}{c} \mathit{value}\,(e_1) \\ e_2 \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \;\rightarrow e'_2 \,/\, \sigma'_T \,/\, \sigma'_C \,/\, \sigma'_V \end{array}}{\textbf{rawset}\,e_1\,e_2\,e_3 \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \;\rightarrow \textbf{rawset}\,e_1\,e'_2\,e_3 \,/\, \sigma'_T \,/\, \sigma'_C \,/\, \sigma'_V} \qquad \text{(R\_\textsc{Rawset\_Step}\_2)}$$

$$\frac{\begin{array}{c} \mathit{value}\,(e_1) \qquad \mathit{value}\,(e_2) \\ e_3 \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \;\rightarrow e'_3 \,/\, \sigma'_T \,/\, \sigma'_C \,/\, \sigma'_V \end{array}}{\textbf{rawset}\,e_1\,e_2\,e_3 \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \;\rightarrow \textbf{rawset}\,e_1\,e_2\,e'_3 \,/\, \sigma'_T \,/\, \sigma'_C \,/\, \sigma'_V} \qquad \text{(R\_\textsc{Rawset\_Step}\_3)}$$

$$\frac{e_1 \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \;\rightarrow e'_1 \,/\, \sigma'_T \,/\, \sigma'_C \,/\, \sigma'_V}{e_1(e_2) \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \;\rightarrow e'_1(e_2) \,/\, \sigma'_T \,/\, \sigma'_C \,/\, \sigma'_V} \qquad \text{(R\_\textsc{Fun\_App\_Step}\_1)}$$

$$\frac{\begin{array}{c} \mathit{value}\,(e_1) \\ e_2 \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \;\rightarrow e'_2 \,/\, \sigma'_T \,/\, \sigma'_C \,/\, \sigma'_V \end{array}}{e_1(e_2) \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \;\rightarrow e_1(e'_2) \,/\, \sigma'_T \,/\, \sigma'_C \,/\, \sigma'_V} \qquad \text{(R\_\textsc{Fun\_App\_Step}\_2)}$$

$$\frac{e_1 \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \;\to e_1' \,/\, \sigma_T' \,/\, \sigma_C' \,/\, \sigma_V'}{e_1\, \mathit{op}\, e_2 \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \;\to e_1'\, \mathit{op}\, e_2 \,/\, \sigma_T' \,/\, \sigma_C' \,/\, \sigma_V'} \qquad (\text{R\_}\textsc{Binop\_Step\_1})$$

$$\frac{\begin{array}{c} \mathit{value}\,(e_1) \\ e_2 \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \;\to e_2' \,/\, \sigma_T' \,/\, \sigma_C' \,/\, \sigma_V' \end{array}}{e_1\, \mathit{op}\, e_2 \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \;\to e_1\, \mathit{op}\, e_2' \,/\, \sigma_T' \,/\, \sigma_C' \,/\, \sigma_V'} \qquad (\text{R\_}\textsc{Binop\_Step\_2})$$

$$\frac{\begin{array}{cc} \mathit{value}\,(e_1) & \mathit{value}\,(e_2) \\ \mathit{validL}\,(e_1) & \mathit{validR}\,(e_2) \end{array}}{e_1\, \mathit{op}\, e_2 \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \;\to \mathit{evalOp}\,(e_1, e_2, \mathit{op}) \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V} \qquad (\text{R\_}\textsc{Binop})$$

$$\frac{e \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \;\to e' \,/\, \sigma_T' \,/\, \sigma_C' \,/\, \sigma_V'}{\mathbf{cget}\, e\, o\, T \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \;\to \mathbf{cget}\, e'\, o\, T \,/\, \sigma_T' \,/\, \sigma_C' \,/\, \sigma_V'} \qquad (\text{R\_}\textsc{Cget\_Step})$$

$$\frac{e_1 \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \;\to e_1' \,/\, \sigma_T' \,/\, \sigma_C' \,/\, \sigma_V'}{\mathbf{cset}\, e_1\, o\, e_2\, T \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \;\to \mathbf{cset}\, e_1'\, o\, e_2\, T \,/\, \sigma_T' \,/\, \sigma_C' \,/\, \sigma_V'} \qquad (\text{R\_}\textsc{Cset\_Step\_1})$$

$$\frac{\begin{array}{c} \mathit{value}\,(e_1) \\ e_2 \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \;\to e_2' \,/\, \sigma_T' \,/\, \sigma_C' \,/\, \sigma_V' \end{array}}{\mathbf{cset}\, e_1\, o\, e_2\, T \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \;\to \mathbf{cset}\, e_1\, o\, e_2'\, T \,/\, \sigma_T' \,/\, \sigma_C' \,/\, \sigma_V'} \qquad (\text{R\_}\textsc{Cset\_Step\_2})$$

$$\frac{e_1 \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \;\to e_1' \,/\, \sigma_T' \,/\, \sigma_C' \,/\, \sigma_V'}{e_1;\, e_2 \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \;\to e_1';\, e_2 \,/\, \sigma_T' \,/\, \sigma_C' \,/\, \sigma_V'} \qquad (\text{R\_}\textsc{Seq\_Step\_1})$$

$$\frac{\mathit{value}\,(e_1)}{e_1;\, e_2 \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V \;\to e_2 \,/\, \sigma_T \,/\, \sigma_C \,/\, \sigma_V} \qquad (\text{R\_}\textsc{Seq\_Step\_Through})$$

DynaSOAr: A Parallel Memory Allocator for Object-Oriented Programming on GPUs with Efficient Memory Access

Matthias Springer
Tokyo Institute of Technology, Japan
matthias.springer@acm.org

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

Abstract

Object-oriented programming has long been regarded as too inefficient for SIMD high-performance computing, despite the fact that many important HPC applications have an inherent object structure. On SIMD accelerators, including GPUs, this is mainly due to performance problems with memory allocation and memory access: There are a few libraries that support parallel memory allocation directly on accelerator devices, but all of them suffer from uncoalesed memory accesses.

We discovered a broad class of object-oriented programs with many important real-world applications that can be implemented efficiently on massively parallel SIMD accelerators. We call this class *Single-Method Multiple-Objects* (SMMO), because parallelism is expressed by running a method on all objects of a type.

To make fast GPU programming available to domain experts who are less experienced in GPU programming, we developed DynaSOAr, a CUDA framework for SMMO applications. DynaSOAr consists of (1) a fully-parallel, lock-free, dynamic memory allocator, (2) a data layout DSL and (3) an efficient, parallel do-all operation. DynaSOAr achieves performance superior to state-of-the-art GPU memory allocators by controlling both memory allocation and memory access.

DynaSOAr improves the usage of allocated memory with a Structure of Arrays (SOA) data layout and achieves low memory fragmentation through efficient management of free and allocated memory blocks with lock-free, hierarchical bitmaps. Contrary to other allocators, our design is heavily based on atomic operations, trading raw (de)allocation performance for better overall application performance. In our benchmarks, DynaSOAr achieves a speedup of application code of up to 3x over state-of-the-art allocators. Moreover, DynaSOAr manages heap memory more efficiently than other allocators, allowing programmers to run up to 2x larger problem sizes with the same amount of memory.

2012 ACM Subject Classification Software and its engineering → Allocation / deallocation strategies; Software and its engineering → Object oriented languages; Computer systems organization → Single instruction, multiple data

Keywords and phrases CUDA, Data Layout, Dynamic Memory Allocation, GPUs, Object-oriented Programming, SIMD, Single-Instruction Multiple-Objects, Structure of Arrays

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2019.17

Supplement Material ECOOP 2019 Artifact Evaluation approved artifact available at `https://dx.doi.org/10.4230/DARTS.5.2.2`
Source code: `https://github.com/prg-titech/dynasoar`

Acknowledgements This work was supported by a JSPS Research Fellowship for Young Scientists (KAKENHI Grant Number 18J14726). We gratefully acknowledge the support of NVIDIA Corporation with the donation of the TITAN Xp GPU used for this research. We would also like to thank Hoang NT, Jonathon D. Tanks and the anonymous reviewers for their comments and suggestions on earlier versions of this paper.

33rd European Conference on Object-Oriented Programming (ECOOP 2019).
Editor: Alastair F. Donaldson; Article No. 17; pp. 17:1–17:37
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

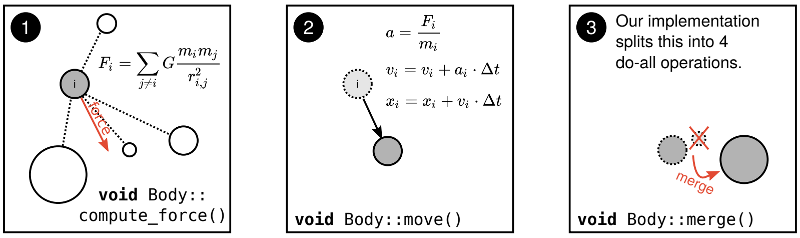


Figure 1 N-body Simulation with Collisions. The simulation consists of multiple do-all operations that are run in a loop for a fixed number of iterations (*time steps*)[1]. Every do-all operation runs in parallel and is a synchronization point: The next one can start only if the previous one has finished.

1 Introduction

General-purpose GPU computing has long been a tedious job, requiring programmers to write hand-optimized, low-level programs. In an attempt to make GPU computing available to a broader range of developers, our efforts are centered around bringing fast object-oriented programming (OOP) to low-level languages such as CUDA.

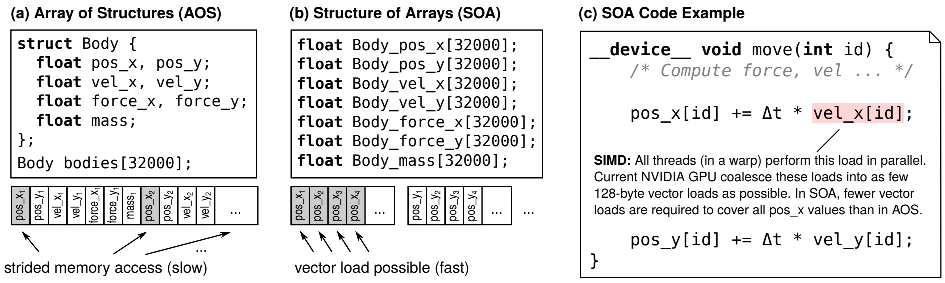
OOP has a wide range of applications in high-performance computing [9, 39, 6, 21, 17] but is often avoided due to bad performance [52]. Dynamic memory management and the ability/flexibility of creating/deleting objects at any time is one of the corner stones of OOP. Due to the massive parallelism and data-parallel execution of GPUs, the number of simultaneous (de)allocations is significantly higher than on other parallel hardware architectures. In recent years, fast, dynamic memory allocators have been developed for GPUs [60, 37, 69, 66, 7, 58, 25, 31] and demanded by application developers [70, 61, 45, 54, 55, 43, 44], showing a growing interest in better programming models and abstractions that have long been available on other platforms. However, while these allocators often provide good (de)allocation performance, they miss key optimizations for structured data, leading to poor data locality and memory bandwidth utilization when accessing allocated memory.

Single-Method Multiple-Objects (SMMO)

We identified a class of high-performance computing applications that can be expressed as object-oriented programs and implemented efficiently on SIMD architectures such as GPUs. We call this class *Single-Method Multiple-Objects* (SMMO). The most fundamental operation of SMMO is parallel *do-all*: Running one method in parallel on all existing objects of a type (*object set*). Such operations fit perfectly with the data-parallel SIMD execution model of GPUs and can be implemented very efficiently. The main challenge lies is the fact that the object set is dynamic: Objects can be created and deleted in GPU code. The main contribution of our work is the design and implementation of a dynamic memory allocator that works well with SMMO applications and runs entirely on the GPU.

SMMO is a broad class of problems with many real-world applications, such as social simulations [27], evacuation simulations [44], predicting wildfire spreading [57], (adaptive [30]) finite element methods [28] or particle systems, to name just a few. As an example, consider the n-body simulation with collisions in Fig. 1. Such simulations are used by astronomers to simulate the collision of galaxies or the formation of planets [4]. Every body is an object in SMMO and the simulation is a sequence of multiple do-all operations.

[1] We implement merging behavior with multiple do-all operations to avoid race conditions.


(a) Array of Structures (AOS)

```
struct Body {
  float pos_x, pos_y;
  float vel_x, vel_y;
  float force_x, force_y;
  float mass;
};
Body bodies[32000];
```

(b) Structure of Arrays (SOA)

```
float Body_pos_x[32000];
float Body_pos_y[32000];
float Body_vel_x[32000];
float Body_vel_y[32000];
float Body_force_x[32000];
float Body_force_y[32000];
float Body_mass[32000];
```

(c) SOA Code Example

```
__device__ void move(int id) {
    /* Compute force, vel ... */

    pos_x[id] += Δt * vel_x[id];
```

SIMD: All threads (in a warp) perform this load in parallel. Current NVIDIA GPU coalesce these loads into as few 128-byte vector loads as possible. In SOA, fewer vector loads are required to cover all pos_x values than in AOS.

```
    pos_y[id] += Δt * vel_y[id];
}
```

Figure 2 Data Layout of N-body Simulation in AOS and SOA. In SOA, multiple values of a field (e.g., $\texttt{pos\_x}_1$ and $\texttt{pos\_x}_2$) can be loaded into a vector register with a single vector load instruction. In AOS, a less efficient, strided memory load or multiple smaller memory loads are necessary, because accessed data is not contiguous.

Structure of Arrays Data Layout

Structure of Arrays (SOA) and Array of Structures (AOS) describe memory layouts for an object set [12] (Fig. 2). In AOS, the standard layout of most platforms, objects are stored as contiguous blocks of memory. In SOA, all values of a field are stored together. This allows for better cache utilization if not all fields are used in a computation. Moreover, it allows for efficient vector loads/stores on SIMD architectures. This is important, because SIMD architectures achieve parallelism by executing the same processor instruction on a *vector* register. Previous work has reported speedups over AOS by multiple factors (e.g., [36]).

Choosing the best data layout for an application is challenging and depends on the data access patterns of the application. Previous work has shown that a mixture of AOS and SOA can sometimes achieve the best performance [29, 40, 68]. How to find good data layouts has been studied before [40, 1] and is out of the scope of this paper. We are focusing on SOA in this work, but DynaSOAr could easily be extended to support other layouts in the future. Unfortunately, custom memory layouts come with a number of disadvantages:

Missing OOP Abstractions. In a hand-written SOA layout, programmers refer to an object with an *integer index* into SOA arrays (Fig. 2c). However, OOP language abstractions (e.g., encapsulation, member access, method calls, type checking, inheritance) only work on object pointers/classes in mainstream languages. To overcome such issues, new languages (e.g., Shapes [29]) and language dialects (e.g., ispc [53]) with built-in support for custom data layouts, as well as data layout libraries/DSLs for existing languages [63, 47, 59] have been developed.

Dynamic Object Set Size. SOA and AOS are not suitable for applications in which the number of objects changes over time, because programmers must specify a maximum object set size per type (e.g., 32,000 in Fig. 2) ahead of time. Dynamic memory allocation solves this problem. As one of our contributions, we show how to allocate memory dynamically while preserving the performance characteristics of SOA.

Subclassing/Inheritance. Inherited methods are shared between superclasses and subclasses. To allow a superclass method implementation to be used for a subclass, the subclass must use the same SOA arrays (and indices) as its superclass. In Columnar Objects, inherited SOA arrays are shared among all objects of all subclasses and newly introduced SOA arrays have a `null` value for objects of a super class [47]. This approach works, but it can waste a considerable amount of memory.

DynaSOAr: A Dynamic Allocator and C++/CUDA DSL for SOA Layout

In this work, we present DynaSOAr, a CUDA framework for SMMO applications. DynaSOAr is a parallel, lock-free, dynamic memory allocator, combined with an efficient do-all operation and an embedded C++/CUDA DSL to enable OOP abstractions with custom object layouts.

We are focusing on DynaSOAr's dynamic memory allocator and do-all operation in this work. DynaSOAr controls the data layout through its memory allocator and data access through its do-all operation. In SMMO applications, DynaSOAr achieves superior performance compared to state-of-the-art allocators due to three main optimizations.

- Objects are stored in a Structure of Arrays (SOA) data layout, a best practice for structured data in SIMD programs, making usage of allocated memory more efficient when used in conjunction with DynaSOAr's do-all operation.
- Memory fragmentation caused by dynamic object (de)allocation is minimized with hierarchical bitmaps. This is important because fragmentation diminishes the benefit of the SOA layout through less efficient vectorized access (more vector transactions are need to access fragmented data) and adversely affects cache performance [32].
- Object allocation and deallocation performance is optimized with a number of low-level techniques. For example, DynaSOAr combines allocation requests within SIMD thread groups (*warps*) to reduce the number of memory accesses during allocations [37] and takes advantage of efficient bit operations/intrinsics.

Contributions and Outline

This paper makes the following contributions.

- The concept of Single-Method Multiple-Objects (SMMO) applications. We show that a variety of important HPC problems are SMMO applications.
- The design and implementation of DynaSOAr, a dynamic object allocator for CUDA; with fast (de)allocation and a do-all operation. To the best of our knowledge, DynaSOAr is the first dynamic allocator that stores objects in an SOA data layout.
- An extension of the SOA data layout to dynamic object sets and subclassing.
- A concurrent, lock-free, hierarchical bitmap, based on atomic operations and retry loops.
- A comparison and evaluation of existing GPU memory allocators on SMMO applications.

The remainder of this paper is organized as follows. Sec. 2 gives an overview of the design goals of DynaSOAr, focusing on memory access considerations of GPUs. Sec. 3 describes the high-level architecture of DynaSOAr and Sec. 4 explains important optimizations such as hierarchical bitmaps. Sec. 5 compares the design of DynaSOAr with other allocators and Sec. 6 evaluates application performance and fragmentation using microbenchmarks and multiple SMMO applications. Finally, Sec. 7 concludes the paper. Additionally, we provide a systematic correctness analysis in the appendix.

2 Design Goals

DynaSOAr is a CUDA framework for SMMO applications and consists of three parts.

Memory Allocator. We developed a dynamic memory allocator that provides `new`/`delete` operations in GPU code and stores objects in an SOA data layout. The main task of the allocator is to decide where to store each field value of each object on the heap.

Data Layout DSL. We developed an embedded C++ DSL to support OOP abstractions while storing objects in a custom layout. We could alternatively implement DynaSOAr in a language that allows programmers to specify custom data layouts (e.g., Shapes [29, 64] or ispc [53]), but such languages have limited GPU support.

Parallel Do-All. We developed an object enumeration strategy for SMMO applications that achieves efficient access of allocated memory on SIMD architectures. By controlling memory allocation and memory access, applications can achive better performance with DynaSOAr than with other state-of-the-art allocators, which are only concerned with memory allocation.

DynaSOAr's DSL builds on top of Ikra-Cpp, an embedded C++ DSL for object-oriented programming with SOA layout [59]. Its purpose is to make DynaSOAr easier to use for programmers. This paper is mainly about the memory allocator and the do-all operation.

2.1 Programming Interface

In contrast to general memory allocators, DynaSOAr is an *object allocator*. The types (classes/structs) that can be allocated must be specified at compile time. DynaSOAr provides five basic operations. All operations except for `parallel_do` and `parallel_new` are *device* functions that can only be called from GPU code.

- `HAllocatorHandle::parallel_do<T, &T::func>(args...)`: Launches a GPU kernel that runs a member function `T::func` for all objects of type T and subtypes[2] existing at launch time (*parallel do-all*). `T::func` may allocate new objects, but those are not enumerated by the same parallel do-all operation. `T::func` may deallocate any object of different type $U \neq T$, but the object it is bound to (`this`) is the only object of type T it may deallocate (delete itself). This is to avoid race conditions.
- `HAllocatorHandle::parallel_new<T>(n, args...)`: Launches a GPU kernel that instantiates n objects of type T. In addition to `args...`, the constructor receives an ID i between 0 and $n-1$ (for the i^{th} object) as the first argument.
- `new(d_allocator) T(args...)`: Allocates a new object of type T and returns a pointer to the object. The *placement new* notation [10] is a common C++ pattern for arena allocation and `d_allocator` is the allocator/arena in which the object is allocated.
- `destroy(d_allocator, ptr)`: Deletes an object that was allocated with `d_allocator`[3].
- `DAllocatorHandle::device_do<T, &T::func>(args...)`: Runs a member function `T::func` for all objects of type T in the current GPU thread. Can only be used inside of a `parallel_do` or a manually launched GPU kernel. This is a sequential *for-each* loop. It is typically used for processing all pairs of objects (e.g., in n-body simulations).

Listing 1 shows parts of the n-body simulation of Fig. 1 to illustrate DynaSOAr's API and DSL.

2.2 Memory Access Performance

The main insight of our work is that optimizing only for fast (de)allocations is not enough. Optimizing the access of allocated memory can result in much higher speedups, because device (*global*) memory access is the biggest bottleneck of memory-bound GPU applications:

[2] To avoid branch divergence, we launch a separate kernel for every type.

[3] There is no *placement delete* syntax, so it is a common pattern to provide a separate `destroy` function [62].

```
#include "dynasoar.h"

class Body;  // Pre-declare all classes. This simple example has only one class.
using AllocatorT = SoaAllocator</*max_num_obj=*/ 16777216, /*T...=*/ Body>;
__device__ DAllocatorHandle<AllocatorT> d_allocator;

class Body : public AllocatorT::Base {  // Can subclass other user-defined class.
 public:
  // Pre-declare all field types. DynaSOAr uses these to compute the size of blocks.
  declare_field_types(Body, float /*pos_x_*/, float /*pos_y_*/,
                            /* ... */, bool /*was_merged_*/)

 private:
  // Declare fields with proxy types but use like normal C++ fields (as in Ikra-Cpp).
  Field<Body, 0> pos_x_;               // Position X
  Field<Body, 1> pos_y_;               // Position Y
  /* other fields omitted... */
  Field<Body, 9> was_merged_;          // Was this body merged into another one?

 public:
  __device__ Body(float pos_x, float pos_y, float vel_x, float vel_y, float mass)
    : pos_x_(pos_x), pos_y_(pos_y), vel_x_(vel_x), vel_y_(vel_y), mass_(mass) {}

  // This constructor is invoked by parallel_new.
  __device__ Body(int idx)
      : Body(/*pos_x=*/ random_float(-kMaxPos, kMaxPos),
             /*pos_x=*/ random_float(-kMaxPos, kMaxPos), /* ... */) {}

  __device__ void apply_force(Body* other) {
    if (other != this) {
      float dx = pos_x_ - other->pos_x_;  float dy = pos_y_ - other->pos_y_;
      float dist = sqrt(dx*dx + dy*dy);
      float F = kGravityConstant * mass_ * other->mass_ / (dist * dist);
      other->force_x_ += F * dx / dist;  other->force_y_ += F * dy / dist;
    }
  }

  __device__ void step_1_compute_force() {
    force_x_ = force_y_ = 0.0f;
    d_allocator->device_do<Body, &Body::apply_force>(this);
  }

  __device__ void step_2_move(float dt) {
    vel_x_ += force_x_ * dt / mass_;  vel_y_ += force_y_ * dt / mass_;
    pos_x_ += dt * vel_x_;            pos_y_ += dt * vel_y_;
  }

  __device__ void step_6_delete_merged() {
    if (was_merged_) { destroy(d_allocator, this); }
  }
};

int main() {
  // Create new allocator. This will allocate a large buffer ("heap") on the GPU.
  auto* h_allocator = new HAllocatorHandle<AllocatorT>();
  // Copy device handle to d_allocator handle.
  cudaMemcpyToSymbol(d_allocator, h_allocator->device_handle(),
                     cudaMemcpyHostToDevice);  // a bit simplified...

  // Create 65536 random body objects. We do not use the new keyword in this example.
  // Alternatively, we could run this in a kernel: new(d_allocator) Body(...)
  h_allocator->parallel_new<Body>(65536);

  for (int i = 0; i < kIterations; ++i) {
    h_allocator->parallel_do<Body, &Body::step_1_compute_force>();
    h_allocator->parallel_do<Body, &Body::step_2_move>(/*dt=*/ 0.5);
    /* some steps omitted... */
    h_allocator->parallel_do<Body, &Body::step_6_delete_merged>();
  }

  delete h_allocator;  // Deallocate buffer and all allocations within.
  return 0;
}
```

Listing 1 DynaSOAr API Example: Excerpt from an n-body simulation with collisions.

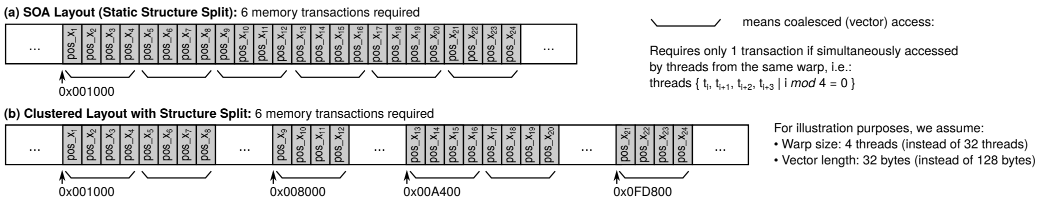

(a) SOA Layout (Static Structure Split): 6 memory transactions required
...
pos_x1
pos_x2
pos_x3
pos_x4
pos_x5
pos_x6
pos_x7
pos_x8
pos_x9
pos_x10
pos_x11
pos_x12
pos_x13
pos_x14
pos_x15
pos_x16
pos_x17
pos_x18
pos_x19
pos_x20
pos_x21
pos_x22
pos_x23
pos_x24
...
0x001000
means coalesced (vector) access:
Requires only 1 transaction if simultaneously accessed by threads from the same warp, i.e.: threads { t_i, t_i+1, t_i+2, t_i+3 | i mod 4 = 0 }
(b) Clustered Layout with Structure Split: 6 memory transactions required
...
pos_x1
pos_x2
pos_x3
pos_x4
pos_x5
pos_x6
pos_x7
pos_x8
...
pos_x9
pos_x10
pos_x11
pos_x12
...
pos_x13
pos_x14
pos_x15
pos_x16
pos_x17
pos_x18
pos_x19
pos_x20
...
pos_x21
pos_x22
pos_x23
pos_x24
...
0x001000
0x008000
0x00A400
0x0FD800
For illustration purposes, we assume:
• Warp size: 4 threads (instead of 32 threads)
• Vector length: 32 bytes (instead of 128 bytes)

Figure 3 Data Layouts: Number of required memory transactions to read 24 floats simultaneously.

Latency. Global memory access instructions have a very high latency at around 400–800 clock cycles, compared to arithmetic instructions at around 6–24 cycles. Programmers can hide latency with *high occupancy* [67] (i.e., running many threads).

Memory Bandwidth. The global memory bandwidth is a limiting factor. Peak memory transfer rates can be achieved only with *memory coalescing*: When the threads in a GPU application simultaneously access different memory addresses, the GPU coalesces accesses from the same SIMD thread group (*warp* in CUDA, every 32 consecutive threads) into one physical transaction if the addresses are on the same 128-byte cache line [38]. However, if threads access data on multiple cache lines (e.g., non-contiguous, spread-out addresses), more transactions are needed[4], which reduces transfer rates significantly. The CUDA Best Practices Guide puts a *high priority* note on coalesced memory accesses [19].

Caches. Hits in the L1/L2 cache are served much faster (less latency, memory bandwidth pressure) than global memory loads. Field reordering and structure splitting are common techniques for increasing the number of hot fields in cache [18].

DynaSOAr achieves good memory access performance with a SOA-style data layout: First, SOA increases memory coalescing because values of the same field, which are accessed simultaneously in SIMD, are stored together. Second, SOA is an extreme form of structure splitting and can improve cache utilization because fields that are not accessed do not occupy cache lines.

2.3 High Density Memory Allocation

A SOA data layout (Fig. 3a) achieves good memory performance but is not suitable for dynamic allocation: The size of SOA arrays is fixed and new allocations cannot be accommodated once all array slots are occupied.

DynaSOAr's design is based on the insight that a *clustered layout* with SOA-style structure splitting (Fig. 3b) has the same cache/vector performance characteristics as a SOA layout, if scalar values are stored in dense clusters of at least 128 bytes (vector and cache line size) and clusters are aligned to 128 bytes, regardless of where the clusters are located. This gives DynaSOAr more freedom in the placement of allocations and is exploited by its allocation policy.

2.4 Parallel Object Enumeration Strategy

Current GPUs follow the Single-Instruction Multiple-Threads (SIMT) execution model. Intuitively, every SIMD lane corresponds to a thread and every group of consecutive 32 threads forms a *warp* which executes the same instruction on a vector register.

[4] This is similar to vectorized loads/stores, but coalescing is performed by the hardware.

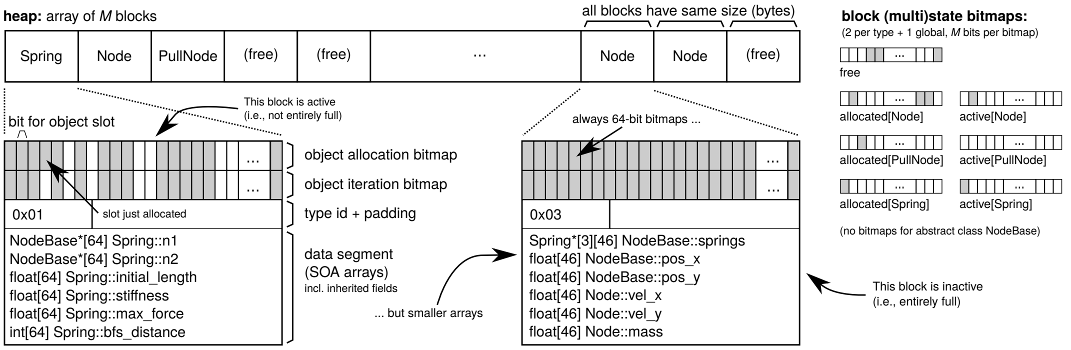


Figure 4 Example: Heap layout for a FEM simulation of a crack in a composite material. The heap is divided into M blocks of equal size. Every block has the same structure: an allocation bitmap, an iteration bitmap, and a type identifier, followed by a data segment storing objects in SOA layout.

To benefit from memory coalescing, the threads of a warp must access addresses on the same 128-byte L1 cache line. In a SOA data layout, this is achieved when the threads of a warp read/write the same fields of objects with contiguous indices at the same time. Intuitively, threads in a warp should process *neighboring* (spatially local) objects.

In DynaSOAr, programmers invoke GPU kernels with parallel do-all operations. These operations must (a) spawn enough GPU threads to hide latency, but not too many to avoid inefficiencies, and (b) assign objects to threads in such a way that memory access is optimized.

2.5 Scalability

Memory allocations require some sort of synchronization between threads to prevent *collisions*, i.e., two threads allocating the same memory location. To avoid collisions, some allocators such as Cilk [14] utilize private heaps, but such designs can lead to high memory consumption (*blowup*) [11] and are in feasible on massively parallel architectures with thousands of threads.

State-of-the-art GPU allocators such as ScatterAlloc [60] and Halloc [3] reduce collisions with hashing, which scatters allocations almost randomly on the heap. This would render a SOA layout useless and defeat one of DynaSOAr's main optimizations.

With such design restrictions, DynaSOAr is bound to have less efficient allocations than other allocators. However, as we show throughout this paper, DynaSOAr can more than make up for slow allocations with more efficient memory access.

Previous CPU memory allocator designs emphasize mechanisms for reducing false sharing, which can degrade performance [11]. This is not an issue on GPUs, because L1 caches are not coherent. Programmers must use the `volatile` keyword or atomic operations to enfore a read/write to the shared L2 cache or global memory.

3 Architecture Overview

DynaSOAr manages a single, large heap in global memory on device. The heap is divided into M blocks of equal number of bytes. M is determined at compile time based on the block size. Multiple objects of the same type (C++ class/struct) are stored in a block in a Structure of Arrays (SOA) data layout (Fig. 4). Once a block is initialized (*allocated*) for a certain type, only objects of that type can be stored in that block until the block (and all its objects) is deallocated again and reinitialized to a different type.

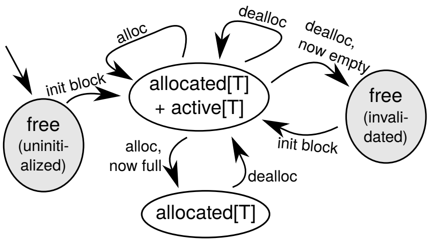


Figure 5 Block State Transitions. At first, blocks are in an uninitialized state. As part of allocation, new active blocks may be initialized (*allocated*). Active blocks become inactive when they are full. Inactive blocks become active again an object is deallocated. Active blocks are invalidated when their last object is deallocated. Invalidated blocks can be reinitialized (to any type) and are handled similar to uninitialized blocks.

The maximum number of objects in a block depends on its type, because different structs/classes may have different sizes. To improve clustering, DynaSOAr allocates new objects in already existing, non-full blocks (*fast path*). We call such blocks *active*, because they participate in allocations (Fig. 5). Only if no active block could be found, a new block is allocated and becomes active (*slow path*).

3.1 Block Structure

Every block has two 64-bit object bitmaps: An *object allocation bitmap* and an *object iteration bitmap*. The allocation bitmap tracks allocated slots in the block. The iteration bitmap is used for object enumeration and overwritten with the allocation bitmap before every parallel do-all operation. Its purpose is to ensure that objects that were created during a do-all operation are not enumerated by the same do-all operation; that would a race condition.

The *type identifier* is a unique ID for the type T of a block. The remainder of the block is occupied by padding and the *data segment*, storing $1 \leq N_T \leq 64$ objects in SOA layout. The data segment begins with SOA arrays for inherited fields and ends with SOA arrays of newly introduced fields.

Slots are marked as (de)allocated with atomic AND/OR operations that change a single bit of the object allocation bitmap. Based on their return value[5], we know ...

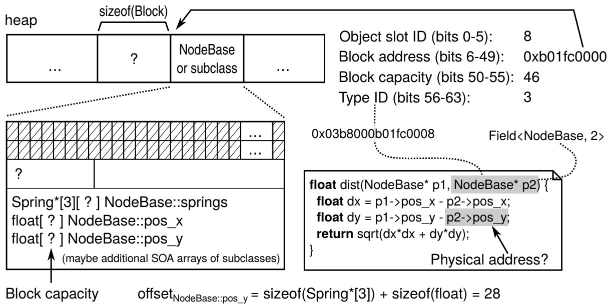
- ... if an allocation was successful or another thread was faster allocating the same slot.
- ... if a particular allocation filled up a block (i.e., allocated the last slot).
- ... if a particular deallocation emptied a block (i.e., deallocated the last slot).

If a thread filled up a block or emptied a block, it is that thread's *responsibility* to update the other internal data structures. This is a common pattern in lock-free designs [49]. Note that every block has the same byte size and structure; e.g., the bitmaps are always at the same offset. This is an important property for the correctness of our lock-free (de)allocation algorithms and simplifies *safe memory reclamation*.

3.2 Block Capacity

The capacity of a block (maximum number of objects) depends on the size (bytes) of the type of objects in the block. If DynaSOAr manages objects of types T_1, T_2, ..., T_n and

[5] An atomic operation returns the value in memory before modification.



heap
sizeof(Block)
...
?
NodeBase or subclass
...
Object slot ID (bits 0-5): 8
Block address (bits 6-49): 0xb01fc0000
Block capacity (bits 50-55): 46
Type ID (bits 56-63): 3
0x03b8000b01fc0008
Field<NodeBase, 2>
...
...
?
Spring*[3][?] NodeBase::springs
float[?] NodeBase::pos_x
float[?] NodeBase::pos_y
(maybe additional SOA arrays of subclasses)
Physical address?
Block capacity
offset$_{\text{NodeBase::pos\_y}}$ = sizeof(Spring*[3]) + sizeof(float) = 28

```
float dist(NodeBase* p1, NodeBase* p2) {
  float dx = p1->pos_x - p2->pos_x;
  float dy = p1->pos_y - p2->pos_y;
  return sqrt(dx*dx + dy*dy);
}
```

Figure 6 Object Pointer Example. The static type of `p2` is `NodeBase*`. The corresponding block has SOA arrays for `NodeBase` fields and for the additional fields of the runtime type of `p2`. The size of those arrays is not statically known and depends on the runtime type of `p2`.

$s = \text{argmin}_{i \in 1 \dots n}\, \mathit{size}(T_i)$ is the index of the smallest type, then the capacity N_T of a block of type T is determined as follows.

$$N_T = \left\lfloor \frac{64 \cdot \mathit{size}(T_s)}{\mathit{size}(T)} \right\rfloor \qquad (\mathit{block\ capacity})$$

A block of the smallest type T_s has capacity 64. Given a fixed heap size, the size of T_s determines the block size in bytes and thus the number of blocks M.

As soon as a type T is more than twice as big as T_s, the benefit of the SOA layout starts fading away for T, because $N_T < 32$. The maximum amount of memory coalescing can only be achieved with vector loads (cluster sizes) of 32 values (assuming 32-bit scalar types). Furthermore, DynaSOAr cannot handle cases in which a type is more than 64 times bigger than the smallest type. In reality, these limitations proved to be insignificant. None of our benchmarks experienced a slowdown due to unfavorable block sizes.

3.3 C++ Data Layout DSL and Object Pointers

Field access is simple in most object-oriented systems: Given an object pointer, which is a memory location, a field value is stored at a fixed offset from the object pointer.

In DynaSOAr, an object pointer is not a memory location, but a combination of various components (*fake pointer* [59]), similar to *global references* in *Shapes* [29]. Upon field access, the DynaSOAr DSL transparently converts object pointers to memory locations, without breaking C++'s OOP abstractions. We follow the implementation strategy of Ikra-Cpp, where fields are declared with proxy types `Field<B, N>`, which are implicitly converted to `T&` values [35], where `T` is the N-th predeclared field type of `B` [59]. This conversion is defined by our DSL and computes the actual, physical memory location within a data segment.

A DynaSOAr object pointer (Fig. 6) is based on the address of the block in which the object is located. All blocks are aligned to 64 bytes, so we can store the object slot ID in the 6 least significant bits. Since recent GPU architectures have at most 24 GB of memory and no virtual memory, only the 35 least significant bits are used in memory addresses and the remaining 29 bits are always zero[6]. We store additional information in these bits: The 8 most significant bits store the type identifier for fast instance-of checks. The next 6 bits store the capacity of the block. Note that, while C++ stores runtime types with a vtable pointer at the beginning of an object, we store runtime type information in unused pointer bits.

[6] We experimentally verified this on NVIDIA Maxwell and NVIDIA Pascal.

While in most object-oriented systems, runtime type information is only required for virtual function calls, DynaSOAr needs the block capacity (a property of the runtime type) also for field accesses, because SOA array offsets within the data segment depend on it.

For example, `p2` in Fig. 6 is statically known to be of type `NodeBase*`, but the block capacity (size of SOA arrays) depends on the runtime type, which can be any subclass of `NodeBase`. Those subclasses can have different block capacities. The size of SOA arrays and the object slot ID are required to compute the physical location of `p2->pos_y`, so we encode both inside object pointers.

This computation, along with bit-shifting and bit-AND operations for extracting all components from an object pointer, is performed on every field read/write (Sec. B). This overhead may seem large, but arithmetic operations are much faster than memory access, even in case of an L2 cache hit. Overall, the performance benefit of SOA is much larger than the address computation overhead.

3.4 Block Bitmaps

To find blocks or free memory quickly during object enumeration or object allocation, DynaSOAr maintains three bitmaps of size M, where M is the maximum number of blocks on the heap.

- The *free block bitmap* indexes block locations that are not yet allocated. This bitmap is used to determine where new blocks are allocated. Bit i is 1 iff block i is free (uninitialized or invalidated). Initially, every bit is 1.
- There is one *block allocation bitmap* for every type T. That bitmap indexes blocks of type T and is used for enumeration of all objects. Blocks of subclasses are not included in bitmaps of the superclass. Initially, every bit is 0.
- There is one *active block bitmap* for every type T, indexing allocated, non-full blocks. If a bit is 1, then the same bit in the block allocation bitmap must also be 1. This bitmap is used to find a block in which a new object can be allocated. Initially, every bit is 0.

Due to concurrent (de)allocations, block bitmaps cannot be kept consistent with the actual block states all the time, as indicated by object allocation bitmaps and type identifiers of blocks. However, we designed our algorithms in such a way that they can handle such inconsistencies and keep block states and block bitmaps *eventually consistent*.

3.5 Object Slot Allocation

When a new object is created, DynaSOAr allocates memory and runs the constructor on the object pointer. Alg. 1 shows how memory is allocated. This algorithm runs entirely on the GPU and is completely lock-free.

DynaSOAr tries to allocate memory in an already existing, active block. If no block could be found, it first initializes a new block at a location that is known to be free (*slow path*). The state of the new block is *allocated* and *active*, so that the new block can also be found by other threads.

Once a block was selected, an object slot is reserved by atomically finding and flipping a bit from 0 to 1 in the object allocation bitmap (details in Alg. 6). Based on the return value of the atomic operation, we know if this operation just allocated the last slot. In that case, the block is marked as *inactive* in the active block bitmap (Line 12).

Algorithm 1: DAllocatorHandle::allocate<T>() : T*. GPU

```
1  repeat                                                   ▷ Infinite loop if OOM
2      bid ← active[T].try_find_set();        ▷ Find and return the position of any set bit.
3      if bid = FAIL then                                       ▷ Slow path
4          bid ← free.clear();          ▷ Find and clear a set bit atomically, return position.
5          initialize_block<T>(bid);          ▷ Set type ID, initialize object bitmaps.
6          allocated[T].set(bid);
7          active[T].set(bid);
8      alloc ← heap[bid].reserve();               ▷ Reserve an object slot. See Alg. 6.
9      if alloc ≠ FAIL then
10         ptr ← make_pointer(bid, alloc.slot);
11         t ← heap[bid].type;                                  ▷ Volatile read
12         if alloc.state = FULL then  active[t].clear(bid) ;
13         if t = T then  return ptr ;
14         deallocate<t>(ptr);                  ▷ Type of block has changed. Rollback.
15 until false;
```

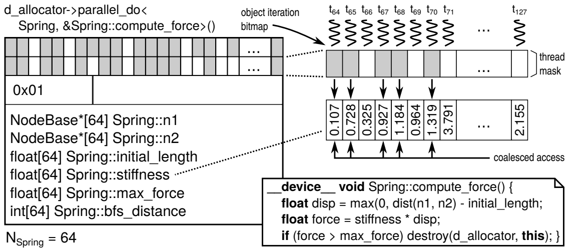
Algorithm 2: DAllocatorHandle::deallocate<T>(T* ptr) : void. GPU

```
1  bid ← extract_block(ptr);
2  slot ← extract_slot(ptr);
3  state ← heap[bid].deallocate(slot);
4  if state = FIRST then
5      active[T].set(bid)
6  else if state = EMPTY then
7      if invalidate(bid) then
8          t ← heap[bid].type;
9          active[t].clear(bid);
10         allocated[t].clear(bid);
11         free.set(bid);
```

Since the allocator is used concurrently by many threads, we may select a block (Line 2) that is full or no longer exists when attempting to reserve an object slot (Line 8). If the block is full, object reservation fails and we retry by selecting a new active block. If the block no longer exists, we have to consider three cases[7].

1. There is currently no block at this location. In this case, object reserveration fails, because all slots are marked as allocated in the object allocation bitmap when a block is deleted. We call this process *block invalidation*.
2. The block was deleted and a new block of the same type was allocated at the same location. Such ABA problems are harmless and allocation will succeed.
3. The block was deleted and there is now a block of different type at the same location. At this point, the constructor has not run yet, so no data in the data segment was corrupted. This is because all blocks have the same structure, i.e., the object allocation bitmap is always at the same location. We can safely rollback the allocation by running the deallocation routine.

[7] We give a more systematic correctness argument in the appendix.


d_allocator->parallel_do<
Spring, &Spring::compute_force>()
object iteration bitmap
t_{64} t_{65} t_{66} t_{67} t_{68} t_{69} t_{70} t_{71} ... t_{127}
thread mask
0.107
0.728
0.325
0.927
1.184
0.964
1.319
3.791
...
2.155
coalesced access
0x01
NodeBase*[64] Spring::n1
NodeBase*[64] Spring::n2
float[64] Spring::initial_length
float[64] Spring::stiffness
float[64] Spring::max_force
int[64] Spring::bfs_distance
N_{Spring} = 64

```
__device__ void Spring::compute_force() {
  float disp = max(0, dist(n1, n2) - initial_length);
  float force = stiffness * disp;
  if (force > max_force) destroy(d_allocator, this); }
```

Figure 7 Thread Assignment Example. 64 threads with consecutive IDs (2 warps) are assigned to every allocated block of type `Spring`. Since not all object slots are in use, as indicated by the block iteration bitmap, some threads have no work to do. All other threads can benefit from memory coalescing when reading/writing fields of the object that they are assigned to.

3.6 Object Deallocation

When an object is deleted, DynaSOAr extracts its runtime type T from the object pointer. Then, DynaSOAr runs the C++ destructor and deallocates the memory (Alg. 2) as follows.

We first extract block and object slot IDs from the object pointer and free the object slot by atomically flipping its bit in the object allocation bitmap from 1 to 0. Based on the return value of the atomic operation we know the fill level of the block right before the deallocation.

If this deallocation freed the first object slot (block previously full), we mark the block as active (Line 5), so that other threads can find it and allocate objects in it. If this deallocation freed the last object slot (block now empty), we attempt to delete the block (Lines 7–11). Safe memory reclamation is known to be difficult in lock-free algorithms [48]. The main problem is that one or more contending threads, in the course of their lock-free operations, may have selected the block that we are about to delete for new allocations.

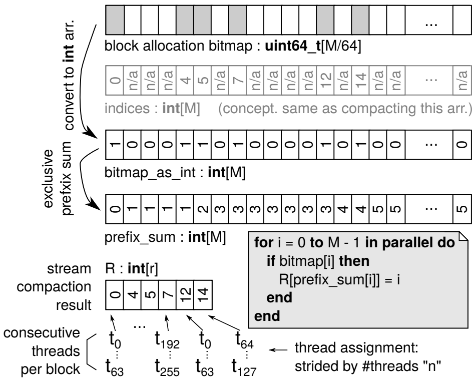
To avoid the block from being modified by other threads, we *invalidate* it. Block invalidation attempts to atomically flipping all bits in the object allocation bitmap from 0 to 1. If this atomic operation failed to flip at least one bit from 0 to 1 (because it was already 1), another thread must have reserved an object slot in the meantime. In this case, we rollback the changes to the object allocation bitmap and abort block invalidation and deletion.

If invalidation was successful, the block is guaranteed to be empty and cannot be modified by other threads anymore because all bits in the object allocation bitmap are 1. The type of the block may have changed in the meantime (Line 8), but it is now safe to mark this block location as *free*, so that a new block can be initialized at this location.

3.7 Parallel Object Enumeration: `parallel_do`

Parallel do-all is the foundation of SMMO applications. It launches a GPU kernel that runs a method `T::func` on all objects of a type T. That method may read and write fields of the object that it is bound to (`this`). The goal of parallel do-all is to assign objects to GPU threads in such a way that memory coalescing is maximized for those field accesses.

Memory coalescing is maximized when all threads of a warp access consecutive memory addresses at the same time. In this case, all those memory accesses can be serviced by efficient vector loads/writes. In CUDA, threads are identified by thread IDs. Each warp consists of a consecutive range of threads. E.g., warp 0 consists of threads $t_0, t_1, \dots t_{31}$. Assuming a block capacity of N_T, DynaSOAr assigns N_T consecutive threads to the objects in a block (Fig. 7). This leads to good memory coalescing on average. Perfect memory coalescing can be achieved if the following two conditions apply.



```
for i = 0 to M - 1 in parallel do
  if bitmap[i] then
    R[prefix_sum[i]] = i
  end
end
```

$$id_O(tid) = tid \ \% \ N_T \qquad (assigned\ obj.\ slot\ idx.)$$

$$id_B(tid) = \left(R\left[\frac{tid + k \cdot n}{N_T} \right] \mid k \in [0; num_B(tid)) \right)$$

$$(assigned\ block\ indices)$$

$$num_B(tid) = \left\lceil \frac{r \cdot N_T - tid}{n} \right\rceil$$

$$(number\ of\ assigned\ blocks)$$

Figure 8 Example: Compacting block allocation bitmap indices and assigning $n = 256$ threads to 6 allocated blocks with $N_T = 64$. This prefix sum-based implementation retains the order of indices (i.e., R is sorted), but this is not necessary for correctness.

- N_T is a multiple of the warp size 32. If this is not the case, then there are warps whose threads process elements in two or more different blocks at the same time.
- Objects have good clustering, i.e., every block except for at most one is entirely full. Due to the way objects are allocated (only in active blocks), we expect a high fill level.

DynaSOAr uses the block allocation bitmap to find blocks to which threads should be assigned. Assigning only one object to a thread is too inefficient if the number of objects is large. Therefore, a thread t_{tid} may have to process an object slot in multiple blocks. Our scheduling strategy always assigns the same object slot position $id_O(tid)$, but in multiple blocks $id_B(tid)$ (strided by the number of threads [34]), to a thread. In those formulas, R is an array of indices of all allocated blocks of type T, i.e., all blocks containing objects of type T. The total number of threads n can be hand-tuned by the programmer. With those formulas, every thread can by itself determine the objects that it should process.

The array R is required because every thread must by itself find the $\frac{tid}{N_T}$-th, $\frac{tid+n}{N_T}$-th, etc. allocated block of type T quickly, without scanning the entire block allocation bitmap. DynaSOAr precomputes R before every parallel-do operation (Fig. 8). Conceptually, this is an application of stream compaction [8] and usually implemented with a prefix sum [56, 13]: Given a bitmap of size M, generate an *indices* array of size M containing i at position i if the i-th bit is set. Otherwise, store an *invalid marker*. Now filter/compact the array to retain only valid values, resulting in an array R of size r. Note that we do not care if the original ordering of indices is retained. Sec. 4.1 describes how this algorithm is further optimized with hierarchical bitmaps to avoid scanning empty bitmap parts.

4 Optimizations

This section describes performance optimizations that DynaSOAr applies in addition to the SOA data layout to achieve good (de)allocation performance.

4.1 Hierarchical Bitmaps

DynaSOAr uses bitmaps for finding blocks or free space for blocks. Since, with growing heap sizes, bitmaps can reach several megabytes in size, we use a hierarchy of bitmaps, such that *set* bits (ones) can be found with a logarithmic order of memory accesses.

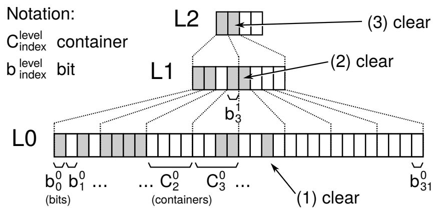


Figure 9 Example: Hierarchical bitmap of size 32 with container size 4 (instead of 64). This example illustrates how (1) a *clear*(18) operation triggers (2) a *clear*(4) operation in the nested bitmap, which triggers (3) a *clear*(1) operation in the next nested bitmap.

Our hierarchical bitmaps are structurally recursive (i.e., bitmaps nested in each other) and hide their hierarchy as an implementation detail from their interface. Such bitmaps are used in database systems [50] and garbage collectors [65], but we do not know of any hierarchical bitmaps that support concurrent modifications.

4.1.1 Data Structure

A hierarchical bitmap of size N bits consists of two parts: an array of size $\lceil N/64 \rceil$ of 64-bit *containers* (`uint64_t`), and a *nested bitmap* of size $\lceil N/64 \rceil$ if $N > 64$. A container C^l_i consists of bits $b^l_{64 \cdot i}$, ..., $b^l_{64 \cdot i + 63}$ and is represented by one bit b^{l+1}_i in the nested (higher-level) bitmap (Fig. 9). That bit is set if at least one bit is set in the container.

$$b^{l+1}_i = \bigvee_{k=0}^{63} b^l_{64 \cdot i + k} \qquad (\textit{container consistency})$$

We chose a container size of 64 bits because C++ has a 64-bit integer type and CUDA (and most other architectures) provide atomic operations for modifying 64-bit values. Bits in a container are changed with atomic operations. Higher-level bits (and thus bitmaps) are *eventually consistent* with their containers. Keeping both consistent all the time is impossible without locking, because two different memory locations cannot be changed together atomically. However, due to the design of the bitmap operations, the bitmap is guaranteed to be in a consistent state when all bitmap operations (of all threads) are completed, at the end of a GPU kernel. Bitmap operations retry or give up (*FAIL*) to handle temporary inconsistencies. This is a key difference compared to other lock-free hierarchical data structures such as SNZI [26], which have stronger runtime consistency guarantees and require more complex algorithms.

4.1.2 Operations

All bitmap operations except for *indices()* are device functions that run entirely on the GPU. All operations that modify memory are thread-safe and their semantics are atomic. Internally, they are all implemented with atomic memory operations.

- `try_clear(pos)` atomically sets the bit at position `pos` to 0. Returns *true* if the bit was 1 before and *false* otherwise.
- `clear(pos)` *switches* the bit at position `pos` from 1 to 0. Retries until the bit was actually changed by the current thread. This is identical to `while (!try_clear(pos)) {}`.
- `set(pos)` switches the bit at position `pos` from 0 to 1. Retries until the bit was changed.

Algorithm 3: Bitmap::try_clear(pos) : void. GPU

```
1 cid ← pos / 64;
2 offset ← pos % 64;
3 mask ← 1 << offset;
4 prev ← atomicAnd(&container[cid], ∼mask);
5 success ← (prev & mask) $\neq$ 0;
6 if success $\wedge$ has_nested $\wedge$ popc(prev) $= 1$ then
7     nested.clear(cid);
8 return success;
```

population cnt.: number of set bits

Algorithm 4: Bitmap::try_find_set() : int. GPU

```
if has_nested then
    cid ← nested.try_find_set();
    if cid = FAIL then return FAIL ;
else
    cid ← 0;
offset ← ffs(container[cid]);
if offset = NONE then
    return FAIL
else
    return 64*cid + offset;
```

find first set: idx. of 1^{st} set bit

- `try_find_set`() returns the position of an arbitrary bit that is set to 1 or *FAIL* if none was found. Must be used with caution, because the returned bit position might already have changed when using the result.
- `clear`() atomically clears and returns the position of an arbitrary set bit. This is identical to `while ((i = try_find_set()) != FAIL && try_clear(i)) {}; return i;`.
- `get`(`pos`) returns the value of the bit at position `pos`.
- `indices`() returns an array of indices of all set bits. This is a host function and cannot be used in a GPU kernel.

4.1.3 Set and Clear with Atomic Operations

As many other lock-free algorithms, our hierarchical bitmaps are based on a combination of atomic operations and retries [20]. The return value of an atomic operation indicates if a bit was actually changed and if it is this thread's responsibility to update the higher-level bitmap (Fig. 9).

As an example, Alg. 3 shows how to clear the bit at position `pos`. In Line 4, the respective container is bit-ANDed with a mask containing ones everywhere except for that position. This will clear the bit at position `pos` but leave all other bits unchanged. The current thread actually changed the bit if it is set in `prev` (Line 5). If this operation cleared the last bit (Line 6), then the bit in the higher-level bitmap must be cleared.

Note that higher-level bits are always changed with *clear(pos)*/*set(pos)* and not with their respective *try_* versions, because other concurrently running bitmap operations that are still in process may not have updated all higher-level bitmaps yet, leaving the data structure in a temporarily inconsistent state. If we were to use *try_* versions, a mandatory update of the higher-level bitmap could be accidentally dropped due to a bitmap inconsistency. *clear(pos)*/*set(pos)* ensure that the update is performed eventually by retrying (and spin-blocking the thread) until the update was successful.

Algorithm 5: Bitmap::indices() : int[N]. CPU

```
1  if has_nested then
2      selected ← nested.indices()
3  else
4      selected ← [0]
5  R ← array(N);
6  r ← 0;
7  for cid ∈ selected in parallel do    ▷ GPU
8      c ← container[cid];
9      s ← atomicAdd(&r, popc(c));
10     for $i$ ← 0 to popc(c) do
11         R[s + i] ← 64*cid + nth_bit(c, i);    idx. of $i^{th}$ set bit in $c$
12 return R.subarray(0, r);
```

4.1.4 Finding an Arbitrary Set Bit

Instead of scanning the entire L0 bitmap, set bits can be found faster with a top-down traversal of the bitmap hierarchy, as shown in Alg. 4. A request is first delegated to the higher-level bitmap (Line 2) to select a container. When that call returns, a set bit is chosen in the selected container (Line 6).

Even if the bitmap has set bits, this operation can fail if it reads an inconsistent combination of containers from different hierarchy levels. For example, consider that a container with exactly one set bit is chosen by the recursive call. However, before reaching Line 6, another thread clears that bit as part of a concurrent bitmap operation. In that case, *try_find_set* fails even though there may be set bits in other containers.

DynaSOAr's performance is affected by such bitmap inconsistencies when searching for active blocks (Alg. 1, Line 2). While bitmap inconsistencies do not affect correctness, they lead to higher fragmentation because DynaSOAr will initialize additional blocks even though objects could be accommodated in already existing blocks. We analyze the effect of such bitmap inconsistencies in our benchmarks (Sec. 6.3).

4.1.5 Enumerating Set Bit Indices

Before launching a parallel do-all kernel, DynaSOAr uses the *indices* operation to generate a compact array of allocated block indices (R in Fig. 8). No GPU code is running at this time, so the bitmap is guaranteed to be in a consistent state. To ensure good scaling with increasing heap sizes, and thus increasing block bitmap sizes, DynaSOAr utilizes the bitmap hierarchy to quickly skip containers without any set bits (Alg. 5).

First, an index array is generated for the higher-level bitmap (Line 2). This array is then processed in parallel; the *for* loop in Line 7 is a GPU kernel and every thread processes one or multiple containers selected by the recursive call. If a container C_i^l does not have any set bits, then its corresponding bit b_i^{l+1} is in a cleared state in the higher-level bitmap and not included in `selected`. Every thread reserves space in the result array R by increasing an atomic counter and fills its portion of the array with bit indices. This algorithm proved to be faster and requires less memory than a prefix sum algorithm, which needs multiple array copies/buffers per bitmap. Note that, in contrast to the prefix sum-based implementation of Sec. 3.7, this algorithm does not retain the order of indices and R and is not sorted.

4.2 Reducing Thread Contention

In Alg. 4 and 6, threads are competing with each other for bits: Only one thread can reserve any given object slot and only a limited number of threads can succeed with allocations in a block. To guarantee correctness, our design is heavily based on atomic operations. These

Algorithm 6: Block::reserve() : (int, state). GPU

```
1  repeat
2      pos ← ffs(∼bitmap);
3      if pos = NONE then return FAIL ;
4      mask ← 1 << pos;
5      before ← atomicOr(&bitmap, mask);
6      success ← (before & mask)) = 0;
7      block_full ← before = 0xFF...F;
8  until success ∨ block_full;
9  if success then
10     if popc(before) = 63 then
11         return (pos, FULL)
12     else
13         return (pos, REGULAR)
14 return FAIL;
```

operations became considerably faster with recent GPU architectures [22, 2], but performance can still suffer when too many threads choose the same bit, because threads have to retry if allocation fails. DynaSOAr employs two techniques to reduce such thread contention.

Allocation Request Coalescing. Originally proposed by XMalloc [37], DynaSOAr combines memory allocation requests of the same type within a warp. One *leader thread* reserves all object slots in a single block on behalf of all participating threads (optimized version of Alg. 6). If the selected active block does not have enough free object slots, DynaSOAr reserves as many slots as possible and then chooses another active block for the remaining allocation requests. This reduces atomic memory operations, because multiple bits in an object allocation bitmap are set in one operation. Furthermore, the constructor for newly allocated objects can run more efficiently, because field accesses are coalesced.

Bitmap Rotation. Instead of a plain *find first set* (ffs) in Alg. 4 and 6, bitmaps are first rotating-shifted by a value depending on the warp ID and a seed that is changed with every retry. This increases the probability of threads choosing different active blocks for allocation and reduces the probability of threads trying to reserve the same object slots in a block. This is a key optimization technique that improved performance by an order of magnitude.

While bitmap traversals are relatively cheap, block initializations are expensive because in addition to initializing object bitmaps, bits in three different bitmaps (plus hierarachy) must be changed (slow path of Alg. 6). To avoid unnecessary block initializations, it proved beneficial to retry the search for active blocks (Line 2) a constant number of times before entering the slow path. This optimization resulted in lower fragmentation and improved performance.

4.3 Efficient Bit Operations

DynaSOAr is taking advantage of efficient bitwise operations such as *ffs* ("find first set") and *popc* ("population count"). Modern CPU and GPU architectures have dedicated instructions for such operations. As an example, Alg. 6 shows how a single object slot is reserved. Instead of checking all bits in a loop, *ffs* in Line 2 is used to find a free slot (index of a cleared bit) in the object allocation bitmap and *popc* in Line 10 counts the number of previously allocated slots (number of set bits) to decide if this request filled up the block.

As another example, due to allocation request coalescing, every thread must now extract its reserved object slot from a set of allocations performed by a leader thread on behalf of the entire warp. This boils down to finding the i-th set bit in a 64-bit bitmap b of newly reserved object slots, where i is the rank of a thread among all allocating threads in the warp. Instead of checking every bit in b one-by-one (loop with 64 iterations in the worst case) and keeping track of the number of set bits seen so far, we apply $b \leftarrow b$ *& (b - 1)* in a loop $i - 1$ times (to clear the first $i - 1$ bits) and then calculate $\mathit{ffs}(b)$. We omit the details of this optimization here, as it is only one example for a variety of similar low-level optimizations.

5 Related Work

CUDA provides an on-device dynamic memory allocator, but it is unoptimized and slow. To solve this issue, multiple custom allocators have been developed over the last years. These allocators achieve good performance by exploiting an allocation pattern that many applications on massively parallel SIMD architectures exhibit: Most allocations are small in size and due to mostly regular control flow, many allocations have the same byte size.

Halloc [3] is one of these allocators. It is a slab allocator and can allocate only a few dozen predetermined byte sizes between 16 bytes and 3 KB. This is fast but can lead to internal fragmentation. DynaSOAr can avoid such internal fragmentation because allocation sizes are determined from compile-time type information of the application. A slab in Halloc contains same-size allocations and tracks allocations with a bitmap. To avoid scanning large bitmaps, a hash function determines which bits to check during allocations. Only one slab can be active per allocation size and if the active slab becomes too full, it is replaced with a new one. In contrast, more than one block per type can be active in DynaSOAr and blocks are filled up entirely.

XMalloc [37] is the first allocator with allocation request coalescing, which was adopted by many other allocators, including DynaSOAr. Coalesced requests are served from *basicblocks*, which are organized in one of multiple lock-free free lists depending on their size.

FDGMalloc maintains a private heap for every warp [69], similar to Hoard [11]. It does not have a general *free* operation and can only deallocate entire heaps, so it is not suitable for SMMO applications.

CircularMalloc (CMalloc) [66] allocates memory in a ring buffer. Every allocation has a pointer to the next allocation or free chunk, wrapping around at the end of the buffer. CMalloc traverses the linked list for free chunks during allocations. To reduce allocation contention, every multiprocessor starts its traversal at a different location. This is similar to DynaSOAr's bitmap rotation optimization.

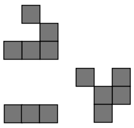
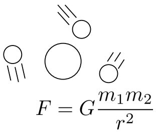
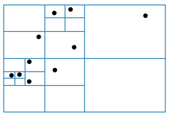
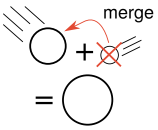
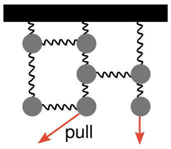
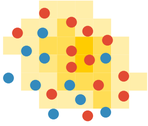
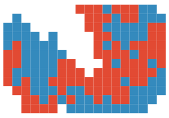
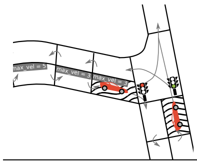
ScatterAlloc [60] hashes allocation requests to memory *pages* depending on their allocation size and the multiprocessor ID. Pages hold allocations of the same size, but slightly smaller requests can be accommodated, leading to internal fragmentation. While DynaSOAr uses hierarchical bitmaps, ScatterAlloc uses hashing with linear probing for finding pages during allocations. For benchmarks, we use mallocMC [24], a reimplementation of ScatterAlloc that is still maintained.

Both Halloc and ScatterAlloc maintain fill levels to quickly skip congested memory areas that are above a certain threshold, because the performance of any hashing technique degrades with an increasing number of collisions. In DynaSOAr, temporary inconsistencies in bitmap hierarchies increase with the number of concurrent allocations, but DynaSOAr can dynamically adapt to such cases by initializing additional blocks.

6 Benchmarks

We evaluated DynaSOAr with multiple real-world SMMO applications that exhibit different memory allocation patterns (Table 1). We ran all benchmarks on a computer with an Intel Core i7-5960X CPU, 32 GB main memory and an NVIDIA TITAN Xp GPU (12 GB device memory), and compiled them with nvcc (-O3) from the CUDA Toolkit 9.1 on Ubuntu 16.04.4.

Table 1 Description of Benchmark Applications.

	Benchmark Description	#par. do-all	#classes	alloc./dealloc.	smallest class	largest class
	**Game of Life:** A cellular automaton due to J. H. Conway. This version has a time complexity of O(#alive cells) instead of the standard O(#cells) algorithm. Cells can be dead, alive or alive-candidates. Alive-candidates are dead cells that may become active in the next iteration. Only alive-candidates and alive cells are processed with parallel do-all operations.	4 / iteration	4 (2 dyn.)	✓ / ✓	5B, 2 fields	8B, 1 field
	**N-Body:** Simulates the movement of particles according to gravitational forces. A `device_do` operation is required to calculate (and then sum up) the gravitational force between every pair of particles. All objects are allocated upfront. This benchmark has no dynamic object (de)allocation.	2 / iteration	1 (0 dyn.)	✗ / ✗	28B, 7 fields	(same)
	**Barnes-Hut:** An extension of N-Body in which bodies are stored in a quad tree [16] (2D), to evaluate DynaSOAr with dynamic tree data structures. The running time is dominated by the construction/maintenance (i.e., frequent node inserts/removals) of the quad tree via parallel top-down/bottom-up tree traversals. Tree nodes are dynamically (de)allocated.	10 / iteration	3 (1 dyn.)	✓ / ✓	68B, 9 fields	102B, 12 fields
	**Particle Collisions:** Similar to N-Body, but particles are merged according to perfectly inelastic collision when they are getting too close. The number of particles decreases gradually. This benchmark has dynamic object deallocation but no dynamic object allocation.	6 / iteration	1 (1 dyn.)	✗ / ✓	38B, 10 fields	(same)
	**Structure:** Simulates a fracture in a composite material, modeled as a FEM. Intuitively, the mesh is a graph and edges between nodes are springs. When pulling the mesh on one side, the material starts to break eventually. Isolated nodes are detected with a BFS [33] and removed. Literature describes extensions that would benefit from dynamic allocation [46].	3 / iteration	5 (4 dyn.)	✗ / ✓	32B, 6 fields	46B, 7 fields
	**Sugarscape:** An agent-based social simulation [27]. Agents inhabit a 2D grid and can move to neighboring cells. Cells contain sugar which is consumed by agents. Sugarscape can simulate a variety social dynamics (e.g., trade, war, environmental pollution). Our simulation is quite simple. We simulate resource consumption, ageing and mating.	12 / iteration	4 (2 dyn.)	✓ / ✓	52B, 7 fields	74B, 11 fields
	**Wa-Tor:** An agent-based predator-prey simulation [23]. Fish/sharks occupy a 2D grid of cells and can move to neighboring cells. Fish and sharks reproduce after some iterations. Fish die when they are eaten and sharks starve to death when they run out of food.	8 / iteration	4 (2 dyn.)	✓ / ✓	60B, 4 fields	64B, 5 fields
	**Nagel-Schreckenberg:** A traffic flow simulation on a street network [51]. This simulation can reproduce traffic jams and other real-world traffic phenomena. Streets are modeled as a network of cells, with at most one vehicle per cell. New vehicles are continuously added to the simulation and existing vehicles are removed at their final destination.	3 / iteration	4 (1 dyn.)	✓ / ✓	97B, 10 fields	124B, 6 fields
	Linux Scalability: Not an SMMO application. This microbenchmark allocates, then deallocates a fixed number of same-size objects in each thread, without ever accessing the memory [42].	n/a	1 (1 dyn.)	✓ / ✓	4B, 1 field	(same)

We compare the running time with different allocators. If possible, we also measured the running time of *baseline* implementations that do not use any dynamic memory management.

Benchmark Applications

We describe all benchmarks and their implementation in detail (incl. their SMMO structure) on GitHub[8]. Our benchmarks are from different domains and fall into four categories.

1. Objects allocated up front, no deallocation: nbody
2. Objects allocated up front, then only deallocation: collision, structure
3. Cellular automaton (CA) with static cells network: sugarscape, traffic, wa-tor
4. Other: barnes-hut, game-of-life

Baselines (SOA/AOS) are application variants without any dynamic memory allocation. Baselines of category (1) are trivial to implement with static allocation. In category (2), every object has a boolean `active` flag to prevent deleted objects from being enumerated in the future. In category (3), classes are merged with the underlying static cell data structure, which wastes memory in case of empty cells (Sec. 6.2). Category (4) applications cannot be implemented with static allocation, unless the application is changed fundamentally.

Parallel do-all in Custom Allocators

Other allocators do not provide do-all operations, which are required for SMMO applications. To compare DynaSOAr with other allocators, we developed standalone `parallel_do` and `device_do` implementations that can be used with any allocator.

These components maintain arrays for allocated and deleted objects of each type. Pointers are inserted into these arrays with atomic operations. At the end of a parallel do-all operation, deleted pointers are removed from the array of allocated pointers. Then, the array of allocated pointers is compacted with a prefix sum operation (same as Fig. 8).

Depending on the number of (de)allocations, this mechanism may take a long time. A better allocator-specific mechanism could likely be developed with some reverse engineering. For that reason, we show the amount of time spent on *parallel enumeration*. This time should not be taken into account when comparing the performance of different allocators.

BitmapAlloc

To analyze the performance of pure bitmap-based object allocation without SOA layout, blocks and fake pointers, we developed a second allocator *BitmapAlloc*. This allocator treats the entire heap as one large object array, whose slots are managed by hierarchical bitmaps, similarly to DynaSOAr: one *allocation bitmap* per type and one *free slot bitmap*. Allocation bitmaps are also used for `parallel_do` and `device_do`.

The main downside of BitmapAlloc is its inefficient memory usage. It supports only a single allocation size, potentially leading to high internal fragmentation.

[8] `https://github.com/prg-titech/dynasoar/wiki/Benchmark-Applications` (also see artifact)

Table 2 Comparison of Allocators. *Coal.* means *Allocation Request Coalescing.*

Allocator	Coal.	SOA	Container	Finding Free Memory
DynaSOAr	✓	✓	Block	Hierarchical Bitmap
DynaSOAr-NoCoal	✗	✓	Block	Hierarchical Bitmap
BitmapAlloc	✗	✗	✗	Hierarchical Bitmap
CircularMalloc	✗	✗	✗	Linked List, Ring Buffer
Default CUDA Allocator	✗	✗	(Unknown)	(Unknown)
FDGMalloc	✓	✗	Priv. Heap, Superblock	Linked List
Halloc	✗	✗	Slab	Bitmap, Hashing
mallocMC (ScatterAlloc)	✓	✗	Superblock, Region, Page	Hashing
XMalloc	✓	✗	(4 block hierarchies)	Lock-free Free Lists

6.1 Performance Overview

Fig. 10 shows the running time of all benchmarked SMMO applications. DynaSOAr achieves superior performance over other allocators due to the SOA layout, a dense object allocation policy and an efficient parallel do-all operation.

All applications except for structure see a speedup by switching from AOS to SOA (compare baselines). In structure, most fields are used together, so SOA does not pay off.

Despite having no dynamic (de)allocation during the benchmark, nbody can see a slight speedup with dynamic memory allocation. This is likely due to fewer cache associativity collisions compared to a denser allocation in array [41].

In collision, DynaSOAr/BitmapAlloc enumerate objects with a bitmap scan of the object allocation bitmap (`device_do`; 1 bit/object), more efficently than other allocators. Other allocators read objects pointers from an array (8 bytes/object). The baseline versions read an `active` flag (1 byte/object) from every object, including deleted ones.

game-of-life and wa-tor are applications that (de)allocate a large number of objects, so enumeration takes a long time. DynaSOAr and BitmapAlloc have much more efficient parallel-do operations than other allocators.

sugarscape and wa-tor exhibit a 2D grid structure of cells. Baseline versions take advantage of this geometric structure, leading to more coalesced memory access, while programmers have no control over where objects are placed in memory by dynamic allocators. For this reason, the baseline versions are faster than the versions with dynamic memory management.

In general, in applications with dynamic memory management, objects are referred to with 64-bit object pointers, while all baseline versions use 32-bit integer indices. This penalizes especially benchmarks with small objects; their object sizes grow considerably just by switching from 32-bit integers indices to 64-bit pointers.

6.2 Space Efficiency

To evaluate how efficiently allocators manage memory, we gave them the same heap size and experimentally determined the max. problem size before running out of memory (Fig. 11).

For category (1) and (2) applications that allocate all memory during startup (collision, nbody, structure), the baseline versions are more space-efficient. The exact number of objects per type is known ahead of time, so placing objects in memory is trivial. However, even though category (2) applications delete objects throughout their runtime, the memory consumption of the baseline versions does not decrease over time. This is a problem even for DynaSOAr because blocks can only be deleted when they are entirely empty, which can take some time.

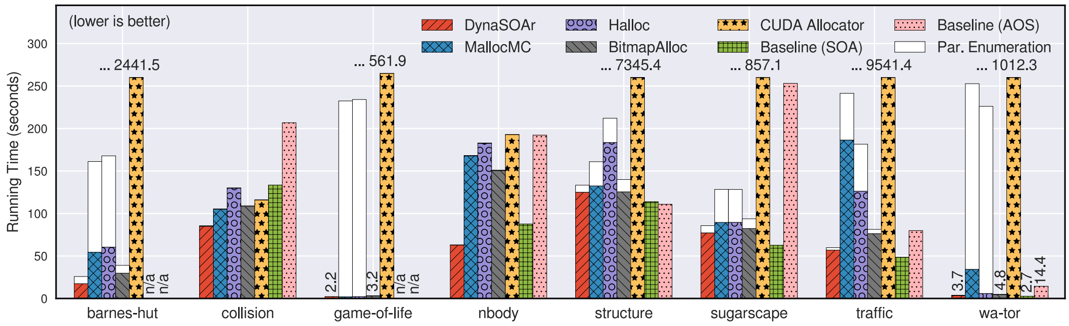


Figure 10 Running Time of SMMO Application Benchmarks. We gave every allocator some extra memory to avoid memory scarcity slowdowns: The heap size is 8 GiB, at least 4 times bigger than the maximum amount of all allocated memory at any point throughout the program execution.

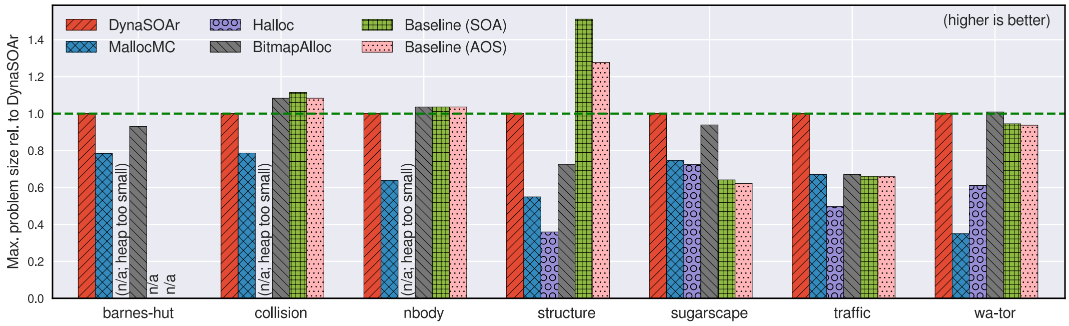
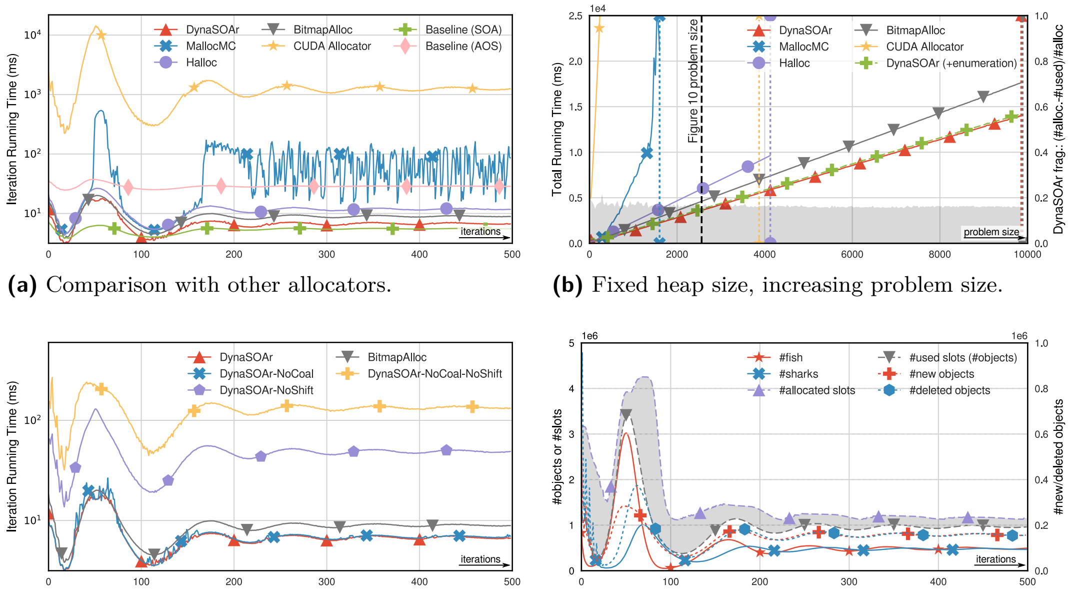


Figure 11 Space Efficiency. We measured the max. problem size of every allocator with the same heap size. Does not take into account enumeration arrays. Results are relative to DynaSOAr.



(a) Comparison with other allocators.

(b) Fixed heap size, increasing problem size.

(c) Isolating single DynaSOAr optimizations.

(d) Number of (de)allocations and fragmentation.

Figure 12 Detailed Analysis of wa-tor. Does not include enumeration time, unless indicated.

Category (3) applications (sugarscape, traffic, wa-tor) exhibit a fixed grid/network structure of cells, upon which a dynamic set of agents is moving. The baseline versions allocate the fields of agents directly inside cells. Classes for agents are combined with the cell class and some fields have `null` values (or garbage) if they are not used. This wastes memory because not all cells are occupied by agents all the time. Here, DynaSOAr is not as fast as optimized SOA baseline implementations, but it can handle significantly larger problem sizes.

Out of all allocators, DynaSOAr is most space-efficient. MallocMC and Halloc are based on a hashing approach. With rising heap fill levels, it becomes increasingly difficult to find free memory for allocations, so they fail to use the entire heap memory. DynaSOAr and BitmapAlloc can avoid this problem with bitmaps, which act as an index for free memory.

Albeit negligible in these benchmarks, DynaSOAr and Baseline (SOA) also benefit from slightly smaller object sizes: Only SOA arrays must be aligned and not every object.

6.3 Detailed Analysis of wa-tor

wa-tor is a particularly interesting benchmark. It exhibits a massive number of (de)allocations in waves, until an equilibrium between fish and sharks is reached. This allows us to measure performance at a massive and at a lower number of concurrent (de)allocations. For a fair comparison of allocators, we do not include time spent on enumeration in this section.

Fig. 12a shows that DynaSOAr always provides superior performance compared to other allocators; during (de)allocation spikes (around iteration 50), as well as if fewer concurrent (de)allocations take place. The performance of mallocMC degrades after a few iterations and does not recover, possibly due to a fragmented heap.

In (b), all allocators were given a heap size of 1 GB and the problem size increases gradually on the x-axis. mallocMC performs well at first, but its performance drops rapidly as soon as the heap starts filling up. DynaSOAr can handle much larger problem sizes, given the same amount of heap memory. The running time grows linearly with the problem size, showing that recent GPU architectures can handle atomic operations quite well.

Fragmentation in DynaSOAr is different from other allocators: DynaSOAr does not have internal or external fragmentation by design, but memory within allocated blocks is only available for a certain type. This sort of fragmentation decreases with better clustering. In DynaSOAr, fragmentation F is the relative number of unused objects slots among all allocated blocks *Blocks* (gray area in (b) and (d)).

$$F = \frac{\sum_{b \in Blocks}(N_{type(b)} - used(b))}{\sum_{b \in Blocks} N_{type(b)}} \approx \frac{1}{\#\text{blocks}} \sum_{b \in Blocks} \frac{\#\text{free slots}(b)}{\#\text{slots}(b)} \qquad (fragmentation)$$

At iterations 60–80 in (d), DynaSOAr has high fragmentation because many fish objects were deallocated. However, a block can only be deallocated when *all* of its objects are deallocated. The fragmentation level decreases gradually because new allocations are performed in existing (active) blocks. Therefore, new blocks are rarely allocated and there is a chance that an active block will eventually run empty. As can be seen in (b), fragmentation is independent of the problem size and constant at around 18% after 500 Wa-Tor iterations.

We implemented multiple DynaSOAr variants to pinpoint the source of DynaSOAr's speedup over other allocators (Fig. 12c. The most important optimization is the rotation-shifting of bitmaps. Without shifting (*-NoShift), performance degrades severely due to thread contention. Allocation request coalescing is another optimization that reduces thread contention significantly (compare DynaSOAr-NoCoal-NoShift and DynaSOAr-NoShift), but it cannot improve performance much further if we are already rotation-shifting bitmaps (compare DynaSOAr and DynaSOAr-NoCoal).

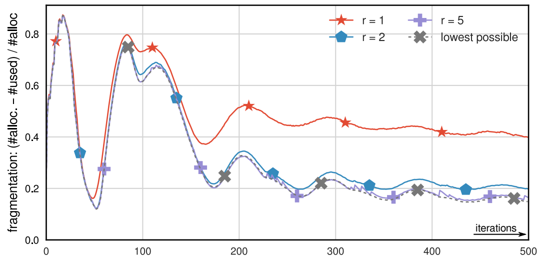
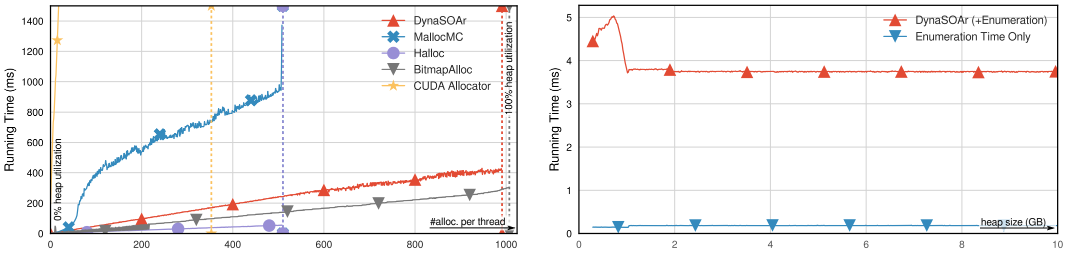


Figure 13 Memory fragmentation (wa-tor) by #active block lookup attemps r (Alg. 6, Line 2). With only 1 retry ($r = 2$), frag. is reduced by 50%. DynaSOAr uses $r = 5$ by default, which is close to the lowest achievable frag. level (i.e., without thread contention). Due to unfortunate alloc.-delloc. patterns, a frag. rate of 0% is not achievable without manually relocating objects or predicting future (de)allocations.



(a) Linux Scalability: Increasing #allocations.

(b) Scaling study: Heap size (wa-tor).

Figure 14 Scaling Study: Number of Allocations and Heap Size.

In Fig. 13, we experiment with the number of active block lookup attempts before entering the slow path, which strongly affects fragmentation.

6.4 Raw Allocation Performance

The *Linux Scalability* microbenchmark [42] measures the raw (de)allocation time of allocators. We set the heap size to 1 GiB and one CUDA kernel allocates n 64-byte objects in each of the 16,384 threads. A second CUDA kernel deallocates all objects. Allocated memory is never accessed. In Fig. 14A, the x-axis denotes the number of allocations per thread n and the y-axis shows the total benchmark running time divided by n.

We chose the size of the heap such that it can hold exactly $16384 \times n$ objects with $n = 1024$ (100% heap utilization). No allocator can reach perfect utilization because some memory is used for internal data structures such as bitmaps.

Halloc is the fastest allocator. Both Halloc and mallocMC fail to allocate more than 510 objects (49.8% utilization). This is better than in some other benchmarks, likely because only objects of one size are allocated. DynaSOAr (96.9% utilization), BitmapAlloc (98.4% utilization) and Halloc scale almost perfectly with the number of allocations.

6.5 Parallel Object Enumeration

The overhead of object enumeration (parallel do-all) is negible in most benchmarks (Fig. 10, Fig. 12B). In Fig. 14B, the problem size is fixed but the heap size increases on the x-axis. DynaSOAr's performance (and that of object enumeration) is independent of the size of the heap, if enough memory is available for the application. This shows that our hierarchical bitmaps work well with various heap sizes.

7 Conclusion

We presented DynaSOAr, a new dynamic object allocator for SIMD architectures. The main insight of our work is that memory allocators should not only aim for good raw (de)allocation performance, but also optimize the usage of allocated memory. DynaSOAr was designed for GPUs, but its basic ideas are applicable to other architectures and systems with good or guaranteed vectorization such as the Intel SPMD compiler [53].

DynaSOAr achieves good memory access performance by controlling (a) memory allocation and (b) memory access with a parallel do-all operation. DynaSOAr's main speedup over other allocators is due to an SOA-style object layout, which can benefit memory bandwidth utilization (through coalesced memory access) and cache utilization. To allow for dynamic (de)allocation of objects, DynaSOAr allocates objects in blocks instead of a plain SOA layout. DynaSOAr utilizes hierarchical bitmaps for fast and compact allocations with low fragmentation.

Our benchchmarks show that DynaSOAr can achieve significant speedups over state-of-the-art allocators of more than 3x in application code with structured data, due to better memory access performance. DynaSOAr also has a significantly lower memory footprint than other allocators, mainly because DynaSOAr has no internal fragmentation by design and is not based on hashing. Our work also shows how an SOA layout can support class inheritance without wasting memory: by allocating objects in blocks and encoding block sizes in object pointers.

In the future, we will investigate how DynaSOAr can be extended to support virtual functions and other custom object layouts.

References

1 James Abel, Kumar Balasubramanian, Mike Bargeron, Tom Craver, and Mike Phlipot. Applications Tuning for Streaming SIMD Extensions. *Intel Technology Journal*, Q2:13, May 1999.

2 Andy Adinets. CUDA pro tip: Optimized filtering with warp-aggregated atomics. `https://devblogs.nvidia.com/cuda-pro-tip-optimized-filtering-warp- aggregated -atomics/`, 2017.

3 Andrew V. Adinetz and Dirk Pleiter. Halloc: A High-Throughput Dynamic Memory Allocator for GPGPU Architectures. In *GPU Technology Conference 2014*, 2014.

4 Stephen G. Alexander and Craig B. Agnor. N-Body Simulations of Late Stage Planetary Formation with a Simple Fragmentation Model. *Icarus*, 132(1):113–124, 1998. `doi:10.1006/icar.1998.5905`.

5 Jade Alglave, Mark Batty, Alastair F. Donaldson, Ganesh Gopalakrishnan, Jeroen Ketema, Daniel Poetzl, Tyler Sorensen, and John Wickerson. GPU concurrency: Weak behaviours and programming assumptions. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 577–591, New York, NY, USA, 2015. ACM. `doi:10.1145/2694344.2694391`.

6 Robert J. Allan. Survey of Agent Based Modelling and Simulation Tools. Technical Report DL-TR-2010-007, Science and Technology Facilities Council, Warrington, United Kingdom, October 2010.

7 Saman Ashkiani, Martin Farach-Colton, and John D. Owens. A Dynamic Hash Table for the GPU. *CoRR*, abs/1710.11246, 2017. `arXiv:1710.11246`.

8 Darius Bakunas-Milanowski, Vernon Rego, Janche Sang, and Chansu Yu. Efficient Algorithms for Stream Compaction on GPUs. *International Journal of Networking and Computing*, 7(2):208–226, 2017. `doi:10.15803/ijnc.7.2_208`.

9 Stefania Bandini, Sara Manzoni, and Giuseppe Vizzari. Agent Based Modeling and Simulation: An Informatics Perspective. *Journal of Artificial Societies and Social Simulation*, 12(4):4, 2009.

10 Eli Bendersky. The many faces of operator new in C++. `https://eli.thegreenplace.net/2011/02/17/the-many-faces-of-operator-new-in-c`, 2011.

11 Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: A Scalable Memory Allocator for Multithreaded Applications. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IX, pages 117–128, New York, NY, USA, 2000. ACM. `doi:10.1145/378993.379232`.

12 Paul Besl. A case study comparing AoS (Arrays of Structures) and SoA (Structures of Arrays) data layouts for a compute-intensive loop run on Intel Xeon processors and Intel Xeon Phi product family coprocessors. Technical report, Intel Corporation, 2013.

13 Markus Billeter, Ola Olsson, and Ulf Assarsson. Efficient Stream Compaction on Wide SIMD Many-core Architectures. In *Proceedings of the Conference on High Performance Graphics 2009*, HPG '09, pages 159–166, New York, NY, USA, 2009. ACM. `doi:10.1145/1572769.1572795`.

14 Robert D. Blumofe and Charles E. Leiserson. Scheduling Multithreaded Computations by Work Stealing. *J. ACM*, 46(5):720–748, September 1999. `doi:10.1145/324133.324234`.

15 Trevor Alexander Brown. Reclaiming Memory for Lock-Free Data Structures: There Has to Be a Better Way. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, PODC '15, pages 261–270, New York, NY, USA, 2015. ACM. `doi:10.1145/2767386.2767436`.

16 Martin Burtscher and Keshav Pingali. Chapter 6 – An Efficient CUDA Implementation of the Tree-Based Barnes Hut n-Body Algorithm. In Wen mei W. Hwu, editor, *GPU Computing Gems Emerald Edition*, Applications of GPU Computing Series, pages 75–92. Morgan Kaufmann, Boston, 2011. `doi:10.1016/B978-0-12-384988-5.00006-1`.

17 John R. Cary, Svetlana G. Shasharina, Julian C. Cummings, John V.W. Reynders, and Paul J. Hinker. Comparison of C++ and Fortran 90 for object-oriented scientific programming. *Computer Physics Communications*, 105(1):20–36, 1997. `doi:10.1016/S0010-4655(97)00043-X`.

18 Trishul M. Chilimbi, Bob Davidson, and James R. Larus. Cache-conscious Structure Definition. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, PLDI '99, pages 13–24, New York, NY, USA, 1999. ACM. `doi:10.1145/301618.301635`.

19 NVIDIA Corporation. CUDA C best practices guide. `https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html#coalesced-access-to-global-memory`, 2018.

20 Cederman Daniel, Gidenstam Anders, Ha Phuong, Sundell Hkan, Papatriantafilou Marina, and Tsigas Philippas. *Lock-Free Concurrent Data Structures*, chapter 3, pages 59–79. Wiley-Blackwell, 2017. `doi:10.1002/9781119332015.ch3`.

21 Kei Davis and Jörg Striegnitz. Parallel Object-Oriented Scientific Computing Today. In Frank Buschmann, Alejandro P. Buchmann, and Mariano A. Cilia, editors, *Object-Oriented Technology. ECOOP 2003 Workshop Reader*, pages 11–16, Berlin, Heidelberg, 2004. Springer-Verlag. `doi:10.1007/978-3-540-25934-3_2`.

22 Simon Garcia De Gonzalo, Sitao Huang, Juan Gómez-Luna, Simon Hammond, Onur Mutlu, and Wen-mei Hwu. Automatic Generation of Warp-level Primitives and Atomic Instructions for Fast and Portable Parallel Reduction on GPUs. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO 2019, pages 73–84, Piscataway, NJ, USA, February 2019. IEEE Press. `doi:10.1109/CGO.2019.8661187`.

23 Alexander K. Dewdney. Computer Creations: Sharks and fish wage an ecological war on the toroidal planet Wa-Tor. *Scientific American*, 251(6):14–26, December 1984.

24 Carlchristian H. J. Eckert. Enhancements of the massively parallel memory allocator ScatterAlloc and its adaption to the general interface mallocMC, October 2014. Junior thesis. Technische Universität Dresden. `doi:10.5281/zenodo.34461`.

25 Harold C. Edwards and Daniel A. Ibanez. Kokkos' Task DAG Capabilities. Technical Report SAND2017-10464, Sandia National Laboratories, Albuquerque, New Mexico, USA, September 2017. doi:10.2172/1398234.

26 Faith Ellen, Yossi Lev, Victor Luchangco, and Mark Moir. SNZI: Scalable nonzero indicators. In *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '07, pages 13–22, New York, NY, USA, 2007. ACM. doi:10.1145/1281100.1281106.

27 Joshua M. Epstein and Robert Axtell. *Growing Artificial Societies: Social Science from the Bottom Up*, volume 1. The MIT Press, 1 edition, 1996.

28 Bruce W.R. Forde, Ricardo O. Foschi, and Siegfried F. Stiemer. Object-oriented finite element analysis. *Computers & Structures*, 34(3):355–374, 1990. doi:10.1016/0045-7949(90)90261-Y.

29 Juliana Franco, Martin Hagelin, Tobias Wrigstad, Sophia Drossopoulou, and Susan Eisenbach. You Can Have It All: Abstraction and Good Cache Performance. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2017, pages 148–167, New York, NY, USA, 2017. ACM. doi:10.1145/3133850.3133861.

30 Dietma Gallistl. The adaptive finite element method. *Snapshots of modern mathematics from Oberwolfach*, 13, 2016. doi:10.14760/SNAP-2016-013-EN.

31 Isaac Gelado and Michael Garland. Throughput-oriented GPU Memory Allocation. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, PPoPP '19, pages 27–37, New York, NY, USA, 2019. ACM. doi:10.1145/3293883.3295727.

32 Dirk Grunwald, Benjamin Zorn, and Robert Henderson. Improving the Cache Locality of Memory Allocation. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, PLDI '93, pages 177–186, New York, NY, USA, 1993. ACM. doi:10.1145/155090.155107.

33 Pawan Harish and P. J. Narayanan. Accelerating Large Graph Algorithms on the GPU Using CUDA. In *Proceedings of the 14th International Conference on High Performance Computing*, HiPC'07, pages 197–208, Berlin, Heidelberg, 2007. Springer-Verlag. doi:10.1007/978-3-540-77220-0_21.

34 Mark Harris. CUDA pro tip: Write flexible kernels with grid-stride loops. https://devblogs.nvidia.com/cuda-pro-tip-write-flexible-kernels-grid-stride-loops/, 2013.

35 Kevlin Henney. Valued Conversions. *C++ Report*, 12:37–40, July 2000.

36 Holger Homann and Francois Laenen. SoAx: A generic C++ structure of arrays for handling particles in HPC codes. *Computer Physics Communications*, 224:325–332, 2018. doi:10.1016/j.cpc.2017.11.015.

37 Xiaohuang Huang, Christopher I. Rodrigues, Stephen Jones, Ian Buck, and Wen-Mei Hwu. XMalloc: A Scalable Lock-free Dynamic Memory Allocator for Many-core Machines. In *2010 10th IEEE International Conference on Computer and Information Technology*, pages 1134–1139, June 2010. doi:10.1109/CIT.2010.206.

38 Byunghyun Jang, Dana Schaa, Perhaad Mistry, and David Kaeli. Exploiting Memory Access Patterns to Improve Memory Performance in Data-Parallel Architectures. *IEEE Transactions on Parallel and Distributed Systems*, 22(1):105–118, January 2011. doi:10.1109/TPDS.2010.107.

39 Laxmikant V. Kale and Sanjeev Krishnan. CHARM++: A portable concurrent object oriented system based on C++. In *Proceedings of the Eighth Annual Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '93, pages 91–108, New York, NY, USA, 1993. ACM. doi:10.1145/165854.165874.

40 Klaus Kofler, Biagio Cosenza, and Thomas Fahringer. Automatic Data Layout Optimizations for GPUs. In Jesper Larsson Träff, Sascha Hunold, and Francesco Versaci, editors, *Euro-Par 2015: Parallel Processing*, pages 263–274, Berlin, Heidelberg, 2015. Springer-Verlag. doi:10.1007/978-3-662-48096-0_21.

41 Florian Lemaitre and Lionel Lacassagne. Batched Cholesky factorization for tiny matrices. In *2016 Conference on Design and Architectures for Signal and Image Processing (DASIP)*, pages 130–137, October 2016. doi:10.1109/DASIP.2016.7853809.

42 Chuck Lever and David Boreham. Malloc() Performance in a Multithreaded Linux Environment. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '00, Berkeley, CA, USA, 2000. USENIX Association.

43 Xiaosong Li, Wentong Cai, and Stephen J. Turner. Efficient Neighbor Searching for Agent-Based Simulation on GPU. In *Proceedings of the 2014 IEEE/ACM 18th International Symposium on Distributed Simulation and Real Time Applications*, DS-RT '14, pages 87–96, Washington, DC, USA, 2014. IEEE Computer Society. doi:10.1109/DS-RT.2014.19.

44 Xiaosong Li, Wentong Cai, and Stephen J. Turner. Cloning Agent-based Simulation on GPU. In *Proceedings of the 3rd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, SIGSIM PADS '15, pages 173–182, New York, NY, USA, 2015. ACM. doi:10.1145/2769458.2769470.

45 Xiaosong Li, Wentong Cai, and Stephen J. Turner. Supporting efficient execution of continuous space agent-based simulation on GPU. *Concurrency and Computation: Practice and Experience*, 28(12):3313–3332, 2016. doi:10.1002/cpe.3808.

46 X. Lu, B.Y. Chen, V.B.C. Tan, and T.E. Tay. Adaptive floating node method for modelling cohesive fracture of composite materials. *Engineering Fracture Mechanics*, 194:240–261, 2018. doi:10.1016/j.engfracmech.2018.03.011.

47 Toni Mattis, Johannes Henning, Patrick Rein, Robert Hirschfeld, and Malte Appeltauer. Columnar Objects: Improving the Performance of Analytical Applications. In *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, Onward! 2015, pages 197–210, New York, NY, USA, 2015. ACM. doi:10.1145/2814228.2814230.

48 Maged M. Michael. Safe Memory Reclamation for Dynamic Lock-free Objects Using Atomic Reads and Writes. In *Proceedings of the Twenty-first Annual Symposium on Principles of Distributed Computing*, PODC '02, pages 21–30, New York, NY, USA, 2002. ACM. doi:10.1145/571825.571829.

49 Maged M. Michael. Scalable Lock-free Dynamic Memory Allocation. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, PLDI '04, pages 35–46, New York, NY, USA, 2004. ACM. doi:10.1145/996841.996848.

50 Mikołaj Morzy, Tadeusz Morzy, Alexandros Nanopoulos, and Yannis Manolopoulos. Hierarchical Bitmap Index: An Efficient and Scalable Indexing Technique for Set-Valued Attributes. In Leonid Kalinichenko, Rainer Manthey, Bernhard Thalheim, and Uwe Wloka, editors, *Advances in Databases and Information Systems*, pages 236–252, Berlin, Heidelberg, 2003. Springer-Verlag. doi:10.1007/978-3-540-39403-7_19.

51 Kai Nagel and Michael Schreckenberg. A cellular automaton model for freeway traffic. *J. Phys. I France*, 2(12):2221–2229, September 1992. doi:10.1051/jp1:1992277.

52 Parag Patel. Object Oriented Programming for Scientific Computing. Master's thesis, The University of Edinburgh, 2006.

53 Matt Pharr and William R. Mark. ispc: A SPMD compiler for High-Performance CPU Programming. In *2012 Innovative Parallel Computing (InPar)*, pages 1–13. IEEE Computer Society, May 2012. doi:10.1109/InPar.2012.6339601.

54 Max Plauth, Frank Feinbube, Frank Schlegel, and Andreas Polze. A Performance Evaluation of Dynamic Parallelism for Fine-Grained, Irregular Workloads. *International Journal of Networking and Computing*, 6(2):212–229, 2016. doi:10.15803/ijnc.6.2_212.

55 Henry Schäfer, Benjamin Keinert, and Marc Stamminger. Real-time Local Displacement Using Dynamic GPU Memory Management. In *Proceedings of the 5th High-Performance Graphics Conference*, HPG '13, pages 63–72, New York, NY, USA, 2013. ACM. doi:10.1145/2492045.2492052.

56 Shubhabrata Sengupta, Aaron E. Lefohn, and John D. Owens. A Work-Efficient Step-Efficient Prefix Sum Algorithm. In *Workshop on Edge Computing Using New Commodity Architectures*, 2006.

57 Hark-Soo Song and Sang-Hee Lee. Effects of wind and tree density on forest fire patterns in a mixed-tree species forest. *Forest Science and Technology*, 13(1):9–16, 2017. doi:10.1080/21580103.2016.1262793.

58 Roy Spliet, Lee Howes, Benedict R. Gaster, and Ana Lucia Varbanescu. KMA: A dynamic memory manager for OpenCL. In *Proceedings of Workshop on General Purpose Processing Using GPUs*, GPGPU-7, pages 9:9–9:18, New York, NY, USA, 2014. ACM. doi:10.1145/2576779.2576781.

59 Matthias Springer and Hidehiko Masuhara. Ikra-Cpp: A C++/CUDA DSL for object-oriented programming with structure-of-arrays layout. In *Proceedings of the 2018 4th Workshop on Programming Models for SIMD/Vector Processing*, WPMVP'18, pages 6:1–6:9, New York, NY, USA, 2018. ACM. doi:10.1145/3178433.3178439.

60 Markus Steinberger, Michael Kenzel, Bernhard Kainz, and Dieter Schmalstieg. ScatterAlloc: Massively parallel dynamic memory allocation for the GPU. In *2012 Innovative Parallel Computing (InPar)*, pages 1–10. IEEE Computer Society, May 2012. doi:10.1109/InPar.2012.6339604.

61 Radek Stibora. Building of SBVH on Graphical Hardware. Master's thesis, Faculty of Informatics, Masaryk University, 2016.

62 Bjarne Stroustrup. Bjarne Stroustrup's C++ style and technique FAQ. is there a "placement delete"? http://www.stroustrup.com/bs_faq2.html#placement-delete, 2017.

63 Robert Strzodka. Chapter 31 - Abstraction for AoS and SoA Layout in C++. In Wen mei W. Hwu, editor, *GPU Computing Gems Jade Edition*, Applications of GPU Computing Series, pages 429–441. Morgan Kaufmann, Boston, 2012. doi:10.1016/B978-0-12-385963-1.00031-9.

64 Alexandros Tasos, Juliana Franco, Tobias Wrigstad, Sophia Drossopoulou, and Susan Eisenbach. Extending SHAPES for SIMD Architectures: An Approach to Native Support for Struct of Arrays in Languages. In *Proceedings of the 13th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*, ICOOOLPS '18, pages 23–29, New York, NY, USA, 2018. ACM. doi:10.1145/3242947.3242951.

65 Katsuhiro Ueno, Atsushi Ohori, and Toshiaki Otomo. An Efficient Non-moving Garbage Collector for Functional Languages. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, pages 196–208, New York, NY, USA, 2011. ACM. doi:10.1145/2034773.2034802.

66 Marek Vinkler and Vlastimil Havran. Register Efficient Dynamic Memory Allocator for GPUs. *Comput. Graph. Forum*, 34(8):143–154, December 2015. doi:10.1111/cgf.12666.

67 Vasily Volkov. *Understanding Latency Hiding on GPUs.* PhD thesis, EECS Department, University of California, Berkeley, August 2016. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-143.html.

68 Nicolas Weber and Michael Goesele. MATOG: Array layout auto-tuning for CUDA. *ACM Trans. Archit. Code Optim.*, 14(3):28:1–28:26, August 2017. doi:10.1145/3106341.

69 Sven Widmer, Dominik Wodniok, Nicolas Weber, and Michael Goesele. Fast Dynamic Memory Allocator for Massively Parallel Architectures. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, GPGPU-6, pages 120–126, New York, NY, USA, 2013. ACM. doi:10.1145/2458523.2458535.

70 Xiangyuan Zhu, Kenli Li, Ahmad Salah, Lin Shi, and Keqin Li. Parallel Implementation of MAFFT on CUDA-enabled Graphics Hardware. *IEEE/ACM Trans. Comput. Biol. Bioinformatics*, 12(1):205–218, January 2015. doi:10.1109/TCBB.2014.2351801.

Algorithm 7: Block::deallocate(pos) : state ▷ Assuming block size 64. GPU

```
mask ← 1 << pos;
before ← atomicAnd(&bitmap, ~mask);
success ← (before & mask)) $\neq$ 0;
assert(success);                        ▷ Precondition.
if popc(before) = 1 then
    return EMPTY;
else if popc(before) = 64 then
    return FIRST;
else
    return REGULAR;
```

A Concurrency and Correctness

CUDA has a weak consistency model for global memory access [5]. Writes to memory performed by one thread are *not* guaranteed to become visible to other threads in the same order. However, atomic writes *have* that property (*sequential consistency*). Furthermore, *thread fences* can be used between two memory writes to enforce sequential consistency, if necessary.

Moreover, global memory reads/writes may be buffered in registers/caches, without a global memory load/store. Thus, memory writes by one thread may not become visible to other threads until the next GPU kernel, unless reads/writes are `volatile` or performed with atomic operations.

All bitmap operations are sequentially consistent and do not suffer from load/store buffering because they are based on atomic memory operations.

A.1 Object Slot Reservation/Freeing

Inside a block, object allocations are tracked with the object allocation bitmap. Every object allocation bitmap has 64 bits, regardless of the block capacity. If a block's capacity is smaller than 64, then the last $64 - N$ bits are set to 1 during block initialization to prevent threads from reserving these slots during object allocation.

Object slots are reserved/freed with atomic operations. These bypass the incoherent L1 caches and are thread-safe: E.g., based on their return value, we know if the current thread reserved a slot or if a contending thread was faster (Alg. 6, Line 5). Based on their return value, we also know if the current thread reserved the last slot (Line 11), in which case the block should be marked as inactive by the allocation algorithm.

A.1.1 Slot Reservation

`Block::reserve()` (Alg. 6) reserves a single object slot in the block. Our actual implementation may reserve multiple slots at once due to allocation request coalescing.

1. **Preconditions:** Block was initialized at least once. (Calling this method on invalidated blocks or full blocks is OK. This function will simply return FAIL.)
2. **Postconditions:** If the result is different from FAIL, the resulting slot at position is reserved for this thread (and no other thread).
3. **Return Value:** Success indicator, atomically reserved slot position, block state.
4. **Linearization Point:** Atomic OR operation (Line 5).

Algorithm 8: DAllocatorHandle::initialize_block<T>(int bid) : void. GPU

```
heap[bid].type ← T;                                       ▷ Volatile write.
__threadfence();
heap[bid].bitmap ← 0;                 ▷ Volatile write, assuming block capacity 64.
```

A.1.2 Slot Freeing

`Block::deallocate(pos)` (Alg. 7) frees a single object slot in the block. To support allocation request coalescing, we have a modified version of this function that can rollback multiple slots at once.

1. **Preconditions:** Bit `pos` is set to 1 in the object allocation bitmap. (Deleting an object multiple times or trying to delete an arbitrary pointer is illegal.)
2. **Postconditions:** Bit `pos` is set to 0 in the object allocation bitmap.
3. **Return Value:** Block state.
4. **Linearization Point:** Atomic AND operation (Line 2).

A.2 Safe Memory Reclamation with Block Invalidation

Safe memory reclamation (SMR) in lock-free algorithms is notoriously difficult. An SMR problem arises in DynaSOAr when deleting blocks. A block should be deleted as soon as its last object has been deleted. This by itself is easy to detect with atomic operations (Alg. 7, Line 6). However, a contending thread may already have selected the now empty block in the course of its own concurrent allocate operation, before the block is actually deleted. Now it is no longer safe to delete the block, but the deleting thread is not aware of that.

Elaborate techniques for SMR such as hazard pointers and epoch-based reclamation have been proposed in previous work [15, 48]. DynaSOAr is able to exploit a key characteristic of its data structure to solve this SMR problem in a simple way: Since all blocks have the same size and structure, object allocation bitmaps are always located at the same position. Therefore, we can optimistically proceed with bitmap modifications and rollback changes if necessary.

Our solution to SMR is *block invalidation.* Before deleting a block, a thread tries to *invalidate* (atomically set to 1) all bits in the object allocation bitmap. Bits that were already 1 are not considered invalidated because those object slots are in use. After successful invaldation, bits remain invalidated until a new block is initialized in the same location. Other threads may still be able to find the block in the active block bitmap for a while, but object slot reservations can no longer succeed.

Allocating threads can detect changes in the block type. Before a previously invalidated block becomes available for allocations again (by initializing its object allocation bitmap), we update the block type. We put a thread fence between both writes to ensure that threads see the new block type before they see free slots in the bitmap (Alg. 8). Threads allocate objects optimistically and rollback changes should they detect a different block type (Alg. 1, Line 14; also see Sec. A.3).

Algorithm 9: DAllocatorHandle::invalidate(int bid) : bool. $\boxed{\text{GPU}}$

```
1  bitmap_ptr ← &heap[bid].bitmap;
2  before ← atomicOr(bitmap_ptr, 0xFF...F);       ▷ Invalidate (set) all obj. allocation bitmap bits.
3  if before ≠ 0xFF...F then                                          ▷ ≥ 1 bit was invalidated.
4      t ← heap[bid].type;
5      if before = 0 then                             ▷ All 64 bits invalidated by this atomicOr.
6          return true;
7      else                                                 ▷ Not all bits invalidated. Rollback.
8          before_rollback ← atomicAnd(bitmap_ptr, before);
9          if before_rollback ≠ 0xFF...F then                       ▷ Other thread cleared a bit.
10             active[t].clear(bid);                     ▷ Other thread expects an inactive block.
11         if (before_rollback & before) = 0 then            ▷ Empty again. Retry invalidation.
12             return invalidate(bid);
13 return false;
```

Details

Block invalidation[9] (Alg. 9) fails if a thread is unable to invalidate at least one bit. In that case, if at least one bit was changed through invalidation, this change must be rolled back (Line 8): In `before` exactly those bits are zero that were invalidated by the thread.

While a thread is running an invalidation operation, other threads may continue to concurrently reserve/free object slots in the same block, unaware of the fact that a thread is trying to invalidate the block. Those threads will update block bitmaps based on the object allocation bitmap state that they are seeing. Therefore, block invalidation must update block bitmaps, as every invalidated bit appears to be an allocated object slot to other threads.

Since block invalidation fills up a block, the block's *active[t]* state should be removed after Line 7, because, if we enter this *else* branch, the thread just *filled up* the block by reserving the remaining object slots (however, not all 64 slots, otherwise, we would be in the *then* branch of Line 5). However, we defer this step, as an invalidation rollback would likely have to mark the same block as *active[t]* again. Unless, another thread concurrently freed an object slot in-between invalidation and invalidation rollback. For such a thread it will seem as if its deallocation just freed the first slot, causing it activate the block (Alg. 2, Line 5). However, since we defered block deactivation, this *set(bid)* operation will spin until we deactivate the block (Alg. 9, Line 10). If invalidation rollback empties the block again, we try to invalidate the block one more time[10].

Note that block invalidation is independent of the type of a block. After invalidating at least one bit, the block type is fixed until invalidation rollback or block initialization, since other threads do not change invalidated bits. As such, the block cannot be deleted or reinitialized to another type by another thread. Other threads can, however, delete and initialize a block with different type after invalidation rollback. It is, nevertheless, safe to assume a block type of t in Line 10, since this is merely an execution of a defered operation that should have happened earlier when the block type was known to be t.

[9] For presentation reasons, we assume a block capacity of 64 in all algorithms in this paper.
[10] Our actual implementation is iterative instead of recursive.

A.3 Object Allocation

The critical parts during allocations (Alg. 1) are *block selection* (Line 2) and *object slot reservation* (Line 8). Both operations by themselves are atomic, but not together. Block selection returns the index of an active block of type T, so we expect that after Line 8, we reserved an object slot in a block of type T. However, due to concurrent operations of other threads, some of these assumptions may be violated.

Block Full. An active block was selected by `try_find_set` but the block filled up before making an allocation (i.e., the block is no longer active). In this case, object slot reservation will fail. Whenever allocation fails, it will restart from the beginning.

Block Deallocated. A block was selected by `try_find_set` but deallocated before reserving a slot. In this case, slot reservation will fail because the block is now in an invalidated state.

Block Replaced (ABA). A block was selected by `try_find_set` but deallocated and reinitialized to a block of same type T. This is harmless: We do not care about block identity.

Block Replaced (Different Type). A block was selected by `try_find_set` but deallocated and reinitialized to another type[11] $t \neq T$. In this case, the allocation must be rolled back (Line 14). All blocks have the same basic structure, so no data can be overwritten accidentally during bitmap updates. Note that the rollback may trigger additional block bitmap updates.

Active Block Not Selected. A block becomes active shortly after `try_find_set` fails. Or, due to bitmap hierarchy inconsistencies, `try_find_set` fails to find an active block even though active blocks exist. This is harmless: No assumption is violated. A new block will be initialized, which merely increases fragmentation.

Note that a block cannot be deallocated after an object slot was already reserved, because block invalidation would fail. Thus, the type of a block can also no longer change.

A.4 Object Deallocation

The critical part during deallocations (Alg. 2) is consistency between *object slot deallocation* (Line 3) and *block state updates*. If the current thread deallocated the first object (i.e., the block was full), then the block bit must be set to active. If the current thread deleted the last object (i.e., the block is empty), then the block must be deleted. The problem is that object slot deallocation and the corresponding block state update together are not atomic.

Allocate After Delete-First. A thread t_1 deleted the first object of a block. However, before marking the block active (Line 6), another thread t_2 allocated this slot again; the block should be inactive. In this case, t_2 reserved the last slot, so it will mark the block as inactive (Alg. 1, Line 12). This operation expects the bit to be in a set state and it will retry until t_1 sets the bit.

Block Deleted after Delete-First. A thread t_1 deleted the first object of a block. However, before marking the block active, other threads deallocated all other objects and a thread t_2 deleted the block. This is not possible because t_2 expects the block to be active (Line 9), i.e., bit set to 1, and blocks until then.

[11] Block initialization (Alg. 8) has a thread fence between setting the block type and resetting the object allocation bitmap, so threads are guaranteed to read the correct type t after an allocation succeeded.

Block Replaced after Delete-First. A thread t_1 deleted the first object of a block. However, before marking the block active, the block was reinitialized to another type. This is not possible because only deleted blocks can be reinitialized (see previous point).

Allocate after Delete-Last A thread t_1 deleted the last object of a block. However, before deleting the block, another thread t_2 allocated an object again, so it is unsafe to delete the block now. This case in handled by block invalidation.

Block Deleted after Delete-Last. A thread t_1 deleted the last object of a block. However, before deleting the block, another thread t_2 allocated an object and yet another thread t_3 deleted that object, rendering the block empty again and deleting it. Now the block is already deleted when t_1 is trying to delete the block. In this case, block invalidation of t_1 will fail because the block is still in an invalidated state and t_1 fails to invalidate all object slot bits.

Block Replaced after Delete-Last. Same as before, but yet another thread t_4 reininitializes the block to a different type. Now t_1 will invalidate and delete a new block whose type is different. This is OK. Block invalidation will succeed only if the block is empty. Both block invalidation and block deletion are independent of the block type.

A.5 Correctness of Hierarchical Bitmap Operations

A container C_i^l consists of bits $b^l_{64\cdot i}$, ..., $b^l_{64\cdot i+63}$ and is represented by one bit b_i^{l+1} in the nested (higher-level) bitmap. That bit is set if and only if at least one bit is set in the container.

▶ **Definition 1** (Consistency). *A bit in level b_i^{l+1} is **consistent** with its corresponding container C_i^l in the lower-level bitmap if and only if:*

$$b_i^{l+1} = \bigvee_{k=0}^{63} b^l_{64\cdot i+k} = \mathbb{1}\left(\sum C^l_{\lfloor i/64\rfloor} > 0\right)$$

We say that the L_{l+1} bitmap is in a consistent state with the L_l bitmap if all bits b_i^{l+1} in the L_{l+1} bitmap satisfy the consistency criterion. The bitmap data structure as a whole is in a consistent state if all bitmap levels L_i satisfy the consistency criterion.

▶ **Definition 2** (Semantics of Bitmap Operations). *Every bitmap L_l provides operations for setting and clearing bits (Sec. 4.1.2). These operations may update bits in the higher-level bitmap L_{l+1} if they **s**et the **f**irst bit ($SF^l_{\lfloor i/64\rfloor}$) or **c**lear the **l**ast bit ($CL^l_{\lfloor i/64\rfloor}$) of a container C_i^l, respectively:*

$$\underbrace{\mathsf{set}(b_i^l) \textit{ and } \mathbb{1}\left(\sum C^l_{\lfloor i/64\rfloor} = 0\right)}_{\textit{set-first: } SF^l_{\lfloor i/64\rfloor}} \textit{ then } \mathsf{set}(b^{l+1}_{\lfloor i/64\rfloor}) \qquad \forall i \in [0;64)$$

$$\underbrace{\mathsf{clear}(b_i^l) \textit{ and } \mathbb{1}\left(\sum C^l_{\lfloor i/64\rfloor} = 1\right)}_{\textit{clear-last: } CL^l_{\lfloor i/64\rfloor}} \textit{ then } \mathsf{clear}(b^{l+1}_{\lfloor i/64\rfloor}) \qquad \forall i \in [0;64)$$

We would like to show that, assuming that a bitmap data structure is initially in a consistent state and given a multiset of bitmap operations O_0 on the L_0 bitmap, the entire bitmap data structure is in a consistent state after executing all operations.

▶ **Definition 3** (Legal Bitmap Operations). *Let $\#\mathsf{set}(b_i^l)$ and $\#\mathsf{clear}(b_i^l)$ be the number of set and clear operations of b_i^l in a multiset of bitmap operations O_l. We call $\mathbb{S}(b_i^l) = \#\mathsf{set}(b_i^l) - \#\mathsf{clear}(b_i^l)$ the **set-surplus** of b_i^l. O_l is **legal** if it satifies the following conditions.*

1. *Overall bit operation is clear, remain or set:* $\$(b_i^l) \in \{-1, 0, 1\}$.
2. *Bit is in a cleared or set state afterwards:* $b_i^l + \$(b_i^l) \in \{0, 1\}$.

E.g., setting a cleared bit twice and clearing it once ($\$ = 2 - 1 = 1$ and $0 + 1 = 1$) is OK, but setting the bit three times and clearing it once ($\$ = 3 - 1 = 2$) would be an illegal usage of the bitmap data structure. Note that illegal bitmap operations deadlock in our implementation because set and clear spin-block and retry until they acutally changed the bit. If a legal bitmap operations multiset is executed fully concurrent (i.e., one thread per operation), then there is always a thread/operation that can make progress.

▶ **Induction Hypothesis 4.** *Let us assume that a multiset of bitmap operations O_l on the L_l bitmap is legal according to Definition 3 for an arbitrary l and that L_l is initially consistent with L_{l+1}.*

▶ **Lemma 5.** *Under the induction hypothesis, the bitmap operations multiset O_{l+1} that is generated by the operations in O_l according to Definition 2 is also legal. Furthermore, after executing O_l, L_l is still consistent with L_{l+1}.*

Proof. Let us first consider the bitmap operations of a single container C_i^l. Let $\#SF_i^l$ be the number of times a first bit is set in the container and $\#CL_i^l$ be the number of times a last bit is cleared in the container. Then, according to Definition 2, $b_{\lfloor i/64 \rfloor}^{l+1}$ is set $\#SF_i^l$ times and cleared $\#CL_i^l$ times. We have to prove that the set-surplus $\$(b_{\lfloor i/64 \rfloor}^{l+1}) = \#SF_i^l - \#CL_i^l$ satisfies the legality criteria of Definition 3.

Without loss of generality, let us assume that all set-first and clear-last operate on the same bit b_k^l. Then, $\$(b_{\lfloor i/64 \rfloor}^{l+1}) = \$(b_k^l) \in \{-1, 0, 1\}$. Hence, the generated bitmap operations O_{l+1} for any bit on the L_{l+1} bitmap satisfy the first legality condition of Definition 3.

Now we have to show that also the second legality condition holds and that $b_{\lfloor i/64 \rfloor}^{l+1}$ is consistent with C_i^l after executing O_l. We consider two cases.

1. $b_{\lfloor i/64 \rfloor}^{l+1} = 0$. Therefore, due to initial consistency, $\sum C_{\lfloor i/64 \rfloor}^l = 0$. Therefore, $\#SF_i^l - \#CL_i^l \in \{0, 1\}$, otherwise, O_l would not be legal. Therefore, $b_{\lfloor i/64 \rfloor}^{l+1} + \$(b_{\lfloor i/64 \rfloor}^{l+1}) \in \{0, 1\}$.
   a. If $\#SF_i^l - \#CL_i^l = 0$, then $\vee_{k=0}^{63} b_{64 \cdot i + k}^l = 0$ after O_l. At the same time, $\$(b_{\lfloor i/64 \rfloor}^{l+1}) = 0$, so $b_{\lfloor i/64 \rfloor}^{l+1} = 0$ after O_l, which is consistent with the state of C_i^l after O_l.
   b. If $\#SF_i^l - \#CL_i^l = 1$, then $\vee_{k=0}^{63} b_{64 \cdot i + k}^l = 1$ after O_l. At the same time, $\$(b_{\lfloor i/64 \rfloor}^{l+1}) = 1$, so $b_{\lfloor i/64 \rfloor}^{l+1} = 1$ after O_l, which is consistent with the state of C_i^l after O_l.
2. $b_{\lfloor i/64 \rfloor}^{l+1} = 1$. Therefore, due to initial consistency, $\sum C_{\lfloor i/64 \rfloor}^l > 0$. Therefore, $\#SF_i^l - \#CL_i^l \in \{-1, 0\}$, otherwise, O_l would not be legal. Therefore, $b_{\lfloor i/64 \rfloor}^{l+1} + \$(b_{\lfloor i/64 \rfloor}^{l+1}) \in \{0, 1\}$.
   a. If $\#SF_i^l - \#CL_i^l = -1$, then $\vee_{k=0}^{63} b_{64 \cdot i + k}^l = 0$ after O_l. At the same time, $\$(b_{\lfloor i/64 \rfloor}^{l+1}) = -1$, so $b_{\lfloor i/64 \rfloor}^{l+1} = 0$ after O_l, which is consistent with the state of C_i^l after O_l.
   b. If $\#SF_i^l - \#CL_i^l = 0$, then $\vee_{k=0}^{63} b_{64 \cdot i + k}^l = 1$ after O_l. At the same time, $\$(b_{\lfloor i/64 \rfloor}^{l+1}) = 0$, so $b_{\lfloor i/64 \rfloor}^{l+1} = 1$ after O_l, which is consistent with the state of C_i^l after O_l.

If all containers in L_l are consistent with their respective bits in L_{l+1}, then the entire L_l bitmap is consistent with the L_{l+1} bitmap. Futhermore, all generated bitmap operations O_{l+1} are legal because they satisfy both legality criteria. ◀

▶ **Base Case 6.** *The bitmap data structure is initially in a consistent state. Furthermore, O_0 is legal. Otherwise, programmers use the bitmap data structure incorrectly.*

B Field Address Computation

This section describes a key implementation technique of the DynaSOAr DSL, that was taken from Ikra-Cpp [59]: Proxy Types. This technique allows us to implement custom data layouts in C++ 11 without breaking OOP abstractions or modifying the compiler.

Even though fields are declared with type `Field<B, N>`, they can be used almost like normal C++ types. There are certain limitations with respect to automatic type deduction (`auto` keyword). Internally, this is implemented with operator overloading, e.g.:

1. **Implicit Conversion Operator:** `Field<B, N>` values can be implicitly converted to the N-th predeclared type in `B`, without an explicit type cast. We call `B` the *base type*.
2. **Member of Object/Pointer Operators:** It is possible to call non-virtual member functions if the base type is (pointer to) a class or struct.
3. **Subscript Operator:** It is possible to use array access syntax (`[]`) for array base types.
4. **Indirection/Address-of Operators:** It is possible to dereference a value of pointer base type and to take the address of a field value.

Listing 2 shows the implementation of the implicit conversion operator. This code first extracts all components that are required for address computation from an object pointer. Then it returns a reference to an object of the base type at the computed memory location.

```
// Implicit conversion operator: E.g., convert Field<NodeBase, 2> to float& in Figure 6.
template<typename B, int N>
Field<B, N>::operator typename B::predeclared_type<N>&() {
  int offset = ...;  // Computed with template metaprogramming. $\text{offset}_{\text{B::fieldname}}$ in Figure 6.
  auto obj_ptr = reinterpret_cast<uint64_t>(this) - 2;  // p2 in Figure 6.
  // Bits 0-49 and clear 6 least significant bits.
  auto* block_address = reinterpret_cast<char*>(obj_ptr & 0x3FFFFFFFFFFC0);
  int obj_slot_id = obj_ptr & 0x3F;  // Bits 0-5
  int block_capacity = (obj_ptr & 0xFC000000000000) >> 50;  // Bits 50-55
  auto* soa_array = reinterpret_cast<typename B::predeclared_type<N>*>(
      block_address + field_offset * block_capacity);
  return soa_array[obj_slot_id];
}
```

Listing 2 Address Computation in Proxy Field Types.

Reliable State Machines: A Framework for Programming Reliable Cloud Services

Suvam Mukherjee
Microsoft Research, Bangalore, India
t-sumukh@microsoft.com

Nitin John Raj
International Institute of Information Technology, Hyderabad, India
nitinjohnraj@gmail.com

Krishnan Govindraj
Microsoft Research, Bangalore, India
t-krgov@microsoft.com

Pantazis Deligiannis
Microsoft Research, Redmond, USA
pdeligia@microsoft.com

Chandramouleswaran Ravichandran
Microsoft Azure, Redmond, USA
chanravi@microsoft.com

Akash Lal
Microsoft Research, Bangalore, India
akashl@microsoft.com

Aseem Rastogi
Microsoft Research, Bangalore, India
aseemr@microsoft.com

Raja Krishnaswamy
Microsoft Azure, Redmond, USA
rajak@microsoft.com

Abstract

Building reliable applications for the cloud is challenging because of unpredictable failures during a program's execution. This paper presents a programming framework, called *Reliable State Machines* (RSMs), that offers fault-tolerance by construction. In our framework, an application comprises several (possibly distributed) RSMs that communicate with each other via messages, much in the style of actor-based programming. Each RSM is fault-tolerant by design, thereby offering the illusion of being "always-alive". An RSM is guaranteed to process each input request exactly once, as one would expect in a failure-free environment. The RSM runtime automatically takes care of persisting state and rehydrating it on a failover. We present the core syntax and semantics of RSMs, along with a formal proof of *failure-transparency*. We provide a .NET implementation of the RSM framework for deploying services to Microsoft Azure. We carry out an extensive performance evaluation on micro-benchmarks to show that one can build high-throughput applications with RSMs. We also present a case study where we rewrite a significant part of a production cloud service using RSMs. The resulting service has simpler code and exhibits production-grade performance.

2012 ACM Subject Classification Software and its engineering → Software reliability; Software and its engineering → Cloud computing; Software and its engineering → Software fault tolerance

Keywords and phrases Fault tolerance, Cloud computing, Actor framework

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2019.18

Related Version A full version of the paper is available at `https://arxiv.org/abs/1902.09502`.

Acknowledgements We thank the anonymous reviewers for suggesting several ways to improve our work. Nitin John Raj's work was done, in part, during an internship at Microsoft Research, India.

33rd European Conference on Object-Oriented Programming (ECOOP 2019).
Editor: Alastair F. Donaldson; Article No. 18; pp. 18:1–18:29
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

The industry trend in Cloud Computing is increasingly moving towards companies building and renting *cloud services* to provide software solutions to their customers [19]. A cloud service in this context refers to a software application that runs on multiple machines in the cloud, making use of the available resources – both compute and storage – to offer a scalable service to its customers. In this paper, we consider the problem of programming *reliable, fault-tolerant* cloud services.

Cloud services are essentially distributed systems consisting of concurrently running, communicating processes or *agents*[1]. Agents typically maintain state, and process user requests as they arrive, which may cause their state to get updated. Consider a word-counting application: the application receives a stream of words (or strings) as input and continuously produces output in the form of the highest frequency word that it has received so far. Programming such an application for the single-machine scenario is easy – the application maintains a map from words to their frequencies seen so far. For each new word, it updates the map and outputs the word if it is the new highest frequency word.

However, to design a more scalable application, this map can be split across multiple distributed agents. More specifically, the distributed word count application can be designed as follows. A *main* agent receives input words from clients, and sends each word to one of the several *counting* agents (based on some criteria, such as the hash of a word) for processing. Every counting agent maintains its own word-frequency map and the local maximum; whenever the local maximum changes, it sends a message to the *max* agent. The max agent collates the local maxima from all the counting agents and outputs the global maximum.

A reliable cloud service must be resilient to hardware and software failures that can cause the agents to crash, and to network failures that can cause message duplications, reorderings, and drops. To handle crashes in the word-counting service, the programmer needs to use some form of persistent storage for the input stream and the word-frequency maps, and write boilerplate code to read and write this state, while carefully orchestrating it with the rest of the computation. The programmer must also handle network message drops (to avoid missing a word) and duplications (to avoid counting the same occurrence of a word twice). While some existing programming frameworks and languages for distributed systems, such as Orleans [12], Kafka [25], Akka [1], Azure Service Fabric [35], among others, provide the necessary building blocks of persistent storage, transactions, etc., the programmer still has to carefully put them all together. Thus, an application that is quite simple to program in the single-machine scenario, quickly becomes a non-trivial task in the distributed setting.

In this paper, we present a novel programming framework, called *Reliable State Machines* (RSMs), to program reliable, fault-tolerant cloud services. The RSM framework enables the programmer to focus only on the application-specific logic, while providing resilience against failures – both machines and network – through language design and runtime.

At a high-level, RSMs are based on the actor style of programming, where the unit of concurrency is a communicating state machine. The programmer defines the types of events that the RSM can receive, and handlers for each event type. Optionally, the programmer can declare some RSM-local state to be persistent. The event handlers can manipulate state, send messages to other RSMs, and create new RSMs. Issues of orchestrating reads and writes of the persistent state with event handlers, handling network failures, etc., are left to

[1] We use the term agents in this paper as a programming construct to distinguish from Systems constructs like processes or physical/virtual machines.

the RSM runtime. The runtime ensures that the effects of an event handler are committed atomically in an all-or-nothing fashion, making it appear that an RSM processes an input message *exactly once*. In addition, the runtime provides a networking module for *exact once* delivery of messages. RSMs are built on top of the P# framework [15], which provides convenient .NET syntax for programming state machines [31] and enables programmers to systematically test their applications against functional specifications [16]. Section 2 provides an overview of the RSM framework and the word-counting application written using RSMs.

We formalize the syntax and semantics of RSMs and prove a *failure transparency* theorem. The theorem states that the semantics of RSMs that includes runtime failures is a refinement of the failure-free semantics in terms of the observable behavior of an RSM. As a result, programmers can program and test their applications assuming failure-free semantics, while the failure transparency theorem guarantees the same behavior even in the presence of runtime failures. Section 3 contains details of our formalization.

We have developed two different implementations of our framework – one using the Azure Service Fabric platform [4] and the other using Apache Kafka [2, 25] – demonstrating that the basic concepts behind RSMs are general and can be implemented on different platforms (Section 4). Our evaluation (Section 6) shows that performance-wise RSMs are competitive with other production cloud programming frameworks, even with the additional guarantees of failure transparency, and it is possible to build high-throughput applications using RSMs. To evaluate the programming and testing experience, we present a case study where we re-implement an existing production-scale backend service of Microsoft Azure. We show that the RSM implementation of the service is simple, easier to reason about, amenable to systematic testing via the P# framework, and meets its scalability requirements (Section 5).

2 Overview

This section presents an overview of the RSM framework. We show how to program the word-count application with RSMs, followed by the details of the RSM runtime and failure transparency. In the rest of the paper, we use *events* and *messages* interchangeably.

2.1 Programming and testing the word-count application

As mentioned in Section 1, we design the distributed word-count application using three types of RSMs: (a) a main RSM that sets up other RSMs, receives words from the client, and forwards them to the word-count RSMs, (b) word-count RSMs that maintain the highest frequency word they have *individually* seen so far, and (c) a max RSM that aggregates local maxima from the word-count RSMs, and outputs the global maximum.

Listing 1 shows the source code for the main RSM using an abbreviated C#-like syntax. RSMs are programmed as state machines. The programmer first declares the three event types to use in the program: `WordEvent`, `WordFreqEvent`, and `InitEvent`, each carrying the mentioned payloads. Values of type **`rsmId`** (e.g., used in the payload of `InitEvent`) are RSM instance ids. We will explain the use of these events as we go along.

The main RSM has two states: `Init` is the start state, and `Receive` is the state in which it receives the input words. The machine declares two persistent fields: a `WordCountMachines` dictionary to maintain the **`rsmId`** of each word count RSM, and a `MaxMachineId` for the **`rsmId`** of the max machine. Fields declared with the "`Persistent`" types denote persistent local state of the RSM. In the `Init` state, the main RSM creates an instance of max RSM and `N` instances of word-count RSMs (using the **`create`** API), and sends the **`rsmId`** of the max RSM instance to every word count RSM as payload in the `InitEvent` event (using the **`send`**

```
event WordEvent: (word: string); // Event types with their payloads
event WordFreqEvent: (word: string, freq: int);
event InitEvent: (target: rsmId);

machine MainMachine {
  PersistentDictionary<rsmId, int> WordCountMachines; // Set of word count machines
  PersistentRegister<rsmId> MaxMachineId; // The rsmId of the aggregator machine

  start state Init { do Initialize }
  state Receive { on WordEvent do ForwardWord }

  void Initialize () {
    var max_id = create (MaxMachine); // First create the max machine
    store (MaxMachineId, max_id); // Store its rsmId
    for (var i = 0; i < N; ++i) { // Create the word count machines
      var id = create (WordCountMachine); store (WordCountMachine[id], 1);
      send (id, new InitEvent (max_id)); // Send max−machine's rsmId to each word count machine
    }
    jump (Receive); // Begin receiving words
  }

  rsmId GetTargetMachine (string s) { return load (WordCountMachines[hash(s) mod N]); }

  void ForwardWord (WordEvent e) { send (GetTargetMachine (e.word), e); } // Forward the event
}
```

Listing 1 Main RSM for the word count example.

API). The persistent fields are also updated (using **`store`**). The machine then transitions to the `Receive` state (using the **`jump`** API). In the `Receive` state, when the machine receives a `WordEvent` from the environment, which contains the next word, it forwards the word to the appropriate max count machine. Since the `Receive` state specifies no transitions, the RSM remains in the `Receive` state, ready to receive the next word.

Listing 2 shows the code for a word-count RSM. It maintains, in its persistent state, a running map of word frequencies (`WordFreq`) and the highest frequency (`HighFreq`) that it has seen so far. Whenever the highest frequency changes, it forwards the corresponding word to the max machine, using the **`rsmId`** stored in the `TargetMachine` field. This RSM also shows the use of volatile state in the form of the field `WordsSeenSinceLastCrash`; this field is reset every time the RSM fails. Such variables can be used for gathering information such as program statistics that are not required to survive failures. Note that the execution of each handler (its call stack, all local variables, etc.) is also carried out on volatile memory.

A word count machine has two states: `Init` and `DoCount`. In the `Init` state, it waits for the `InitEvent` (from the main machine). The rest of the code is straightforward.

The max RSM, shown in Listing 3, simply takes a maximum over the frequencies that it receives, and forwards the maximum one to an external service (which may print to console or write to an output file).

Implementation. The RSM programming framework is embedded in `C#`, and uses the `P#` state machine programming model. Each RSM is defined as a `C#` class, with local-state as class fields, and event handlers as class methods. Using RSMs does not require the user to learn a new programming language. We provide more implementation details in Section 4.

Testing the application. Having written the application in our framework, the programmer can also test it by supplying a specification and asking the `P#` tester to validate it. In the word count application, for example, a functional correctness specification – eventually the word with the highest frequency is output by the `MaxMachine` RSM – can be tested. The `P#` tester uses state-of-the-art algorithms to search over the space of possible executions of an RSM program [18, 16] and can help catch many bugs. For instance, changing any of the

```
machine WordCountMachine {
  PersistentDictionary<string, int> WordFreq; // Local map for words to their frequencies
  PersistentRegister<int> HighFreq; // The highest frequency seen so far
  PersistentRegister<rsmId> TargetMachine; // The max machine rsmId, forwarded by the main machine
  int WordsSeenSinceLastCrash; // A volatile variable to count words seen since last crash

  start state Init { on InitEvent do Initialize }
  state DoCount { on WordEvent do Count }

  void Initialize (InitEvent e) { // Wait for the init event from the main machine
    store (TargetMachine, e.target);
    jump (DoCount);
  }

  void Count (WordEvent e) { // Receive the word from the main machine
    WordsSeenSinceLastCrash++;
    var f = load (WordFreq[e.word]) + 1; // Increment the frequency of the word by 1
    store (WordFreq[e.word], f);  // And store it back
    if (f > load (HighFreq)) { // Update the highest frequency, if required
      store (HighFreq, f);
      send (load (TargetMachine), new WordFreqEvent (e.word, f)); // And send it to the max machine
    }
  }
}
```

Listing 2 Word count RSM for the word count example.

```
machine MaxMachine {
  PersistentRegister<int> HighFreq; // Highest frequency seen so far

  state DoCount { on WordFreqEvent do CheckMax }

  void CheckMax (WordFreqEvent e) { // Update the current highest frequency if needed
    if (e.freq > load (HighFreq)) {
      store (HighFreq, e.freq);
      send (env, e);
    }
  }
}
```

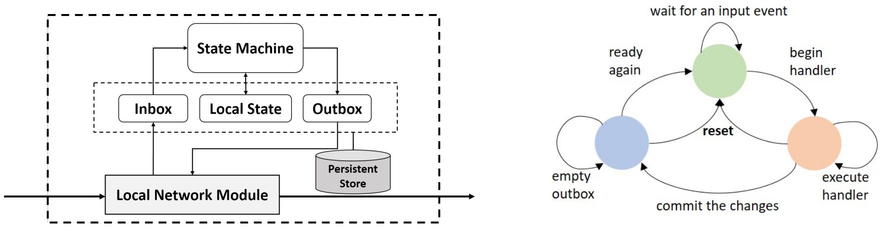
Listing 3 max RSM for the word count example.

persistent variables of the RSMs to volatile will render the program incorrect; indeed if the `MaxMachine`s don't persist their word frequency maps, upon restart, their output may not be correct. If the `MainMachine` does not use the same hash function inside `GetTargetMachine` for all input words, then too the specification fails to hold (because it may forward two different occurrences of the same word to two different RSMs), etc. We confirmed that the `P#` tester is able to find all these errors very quickly.

Summary. Our framework frees the programmer from the burden of designing and programming for failures. In the word count application, as we can see above, the source code *only* contains the application-specific logic, and no boilerplate code for handling failures, restarts, etc. There is still concurrency in the program that may be hard to reason about, which is why we provide `P#` testing. The next section describes the RSM runtime that provides resilience from machine and network failures.

2.2 RSM runtime

Figure 1a shows the runtime architecture of a single RSM. The runtime ensures that each RSM has a unique `rsmId`. An RSM is associated with its own *inbox* of input events, an *outbox* of output events (the events that it sends out), and local state that consists of both persistent and volatile (in-memory) components. The inbox, outbox, persistent fields, and the current state of the RSM state machine (e.g. `Init` or `Receive` in Listing 1) are backed by a persistent store (e.g. a replicated storage system). Each RSM also has a local networking module that



(a) Internals of an RSM.

(b) Flow of operations for an RSM.

Figure 1 Internals and flow of operations for an RSM.

is responsible for communicating with other RSMs or to clients or external services. The inbox and outbox are queues, following the standard FIFO enqueue and dequeue semantics.

The execution of an RSM consists of three operations.

- **Input.** The networking module receives messages over the network and enqueues them to the inbox.
- **Processing.** The processing inside an RSM is single-threaded. It iteratively dequeues an event from the inbox and processes it by executing its corresponding event handler. The handler can create other RSMs or send events to the existing ones. Each of these requests are enqueued to the outbox. The handler can also mutate the persistent and volatile local state of the RSM.
- **Output.** The networking module dequeues messages from the outbox and sends them over the network to their destination.

These operations can execute in any order. In our implementation of RSMs (Section 4), we run them in parallel using background tasks; we ensure that the enqueue and dequeue operations on the queues (inbox and outbox) are linearizable [22], and thus, safe to execute concurrently.

Exact-once processing. The RSM runtime ensures that the effects of an event handler are committed to the persistent storage atomically. In particular, the dequeue of an event e from the inbox, and the result of processing e (including all updates made to persistent fields as well as all enqueues to the outbox) are committed to the persistent storage in a single transaction. Thus, if the RSM fails before committing, then on restarting the RSM, e would still be at the head of the inbox, and none of its effects would have been propagated to the rest of the system. If the RSM fails after committing, then the event e has been processed and will not appear in the inbox on restart. The RSM only sends out those events that have been committed successfully to the outbox.

Networking module. The networking modules work with each other to ensure exact-once delivery of events between RSMs, i.e., an event is dequeued from the outbox of an RSM and enqueued to the inbox of the target RSM atomically. While exact-once delivery is default, the programmer can choose more relaxed delivery semantics. All our examples (Section 4) use the stricter exact-once implementation. To communicate with external non-RSM services, the RSM framework has the notion of an *environment* that acts as an interface to the outside world. The environment can supply input by enqueueing to the inbox of an RSM. The RSMs can in turn send events to a special `rsmId` called `env`, which references the environment. Such events still get enqueued to the outbox of the RSM. When committed, they are forwarded to their intended destination through plug-ins to the networking module supplied by the user.

Non-determinism. We allow RSM handlers to be non-deterministic, i.e., two executions of an event handler on the same event and starting from the same local state may produce different output. For instance, consider an extension to the word-count example where each input word is associated with a timestamp and the main-RSM forwards only those words with a timestamp not older than 24 hours. This requires main-RSM to look up the current time of day and make the decision of forwarding the word or not. This action is non-deterministic because it may not replay exactly on failover. Non-determinism does not change the RSM guarantees in any way: all state changes made by an event handler are first committed locally. This ensures that all non-deterministic choices are resolved and recorded before they are propagated outside the RSM.

Progress. The RSM runtime ensure global progress under the assumptions that: (i) each handler terminates in the absence of failures, (ii) for each handler, the system eventually recovers from failures in order to complete its execution, and (iii) a message that is repeatedly sent over the network is eventually delivered to its destination.

Using `P#` for testing. We chose `P#` for two reasons. First, it provides various programming conveniences for writing state machines and is already in use for writing production code [31, 17]. Second, `P#` offers means of writing end-to-end specifications of a collection of communicating state machines. The specifications (both safety and liveness) can then be validated using powerful systematic search over the space of all interleavings of the program. This method has been shown to be very effective at finding concurrency bugs [18, 15, 16, 28]. In our work, we provide an automatic way of lowering an RSM program to a `P#` program. A programmer can write the specification of an RSM program, then validate the specification using `P#` systematic testing. We provide more details in Section 6.2.

Failure transparency. Using the exact-once processing and exact-once delivery, the RSM framework provides a failure transparency property. The property essentially says that the observable behavior of an RSM is independent of the failures of host machines and the network. This enables the programmers to focus only on the application-specific logic when programming RSMs, and to test only for the failure-free executions. The property relies on the non-interference of the persistent storage from the volatile class fields. Intuitively, the volatile class fields are reset on failures, and so, if they leak into event payloads, for example, the crashes can be observed.

3 Formalization of RSMs

In this section, we formalize a core of the RSM programming model, denoted as R_{SM}, and its operational semantics. We state and prove the *failure transparency* theorem in Section 3.2.

3.1 Syntax and semantics

Figure 2 shows the R_{SM} syntax. For simplicity, we present the syntax in A-normal form [33], where most of the sub-expressions are values. An RSM C in R_{SM} is declared as a class definition D, consisting of `persistent` and `volatile` fields, and an event handler statement $\{\overline{x := n}; s\}$, with local variables $\overline{x}$ and statement s (handlers for specific event types can be encoded in R_{SM} using `if` statements in s). All the variables and fields in R_{SM} are integer-typed.

Statements in the language include local variable assignment ($x := e$), assignment to `volatile` fields ($f := e$), `persistent` field updates (`store` f v), conditional statements

$$\begin{array}{rlll}
\text{Field} & f & & \text{Class name} \quad C \qquad \text{Integer} \quad n, r \\
\text{Value} & v & ::= & n \mid x \mid f \\
\text{Expression} & e & ::= & v \mid \texttt{load}\ f \mid v_1 \oplus v_2 \mid \star \\
\text{Statement} & s & ::= & x := e \mid f := e \mid \texttt{store}\ f\ v \mid \texttt{if}\ v\ s_1\ s_2 \mid s_1; s_2 \mid \texttt{create}\ x\ C \mid \texttt{send}\ v_1\ v_2\ v_3 \\
\text{Class defn.} & D & ::= & \texttt{class}\ C\ \{\overline{\texttt{persistent}\ f := n}; \overline{\texttt{volatile}\ f := n}; \{\overline{x := n}; s\}\}
\end{array}$$

Figure 2 R_{SM} syntax.

$$\begin{array}{rlll}
\text{Field map} & F & ::= & \cdot \mid f \mapsto n, F \\
\text{Local environment} & L & ::= & \cdot \mid x \mapsto n, L \\
\text{Event list} & E & ::= & \cdot \mid (n_r, n_e, n_p), E
\end{array}$$

Figure 3 Runtime configuration syntax for local evaluation.

($\texttt{if}\ v\ s_1\ s_2$) and sequencing ($s_1; s_2$). While the **`volatile`** fields can be operated upon directly (e.g. adding two of them), to work on the **`persistent`** fields, they first need to be **`load`**ed into local variables, and then **`store`**d back. The form **`create`** x C creates a new RSM C and binds its RSM id to the variable x (the ids are also integer-valued). Finally the statement form **`send`** v_1 v_2 v_3 is used to send an event of event type v_2 (an integer) with payload v_3 to the destination machine with RSM id v_1. Expressions e in the language include values v, reading a **`persistent`** field (**`load`** f), and binary operations $v_1 \oplus v_2$. We also model non-determinism in the language – the expression form $\star$ evaluates to a random integer at runtime.

3.1.1 Local evaluation judgment

Operational semantics of R_{SM} consists of two judgments, a local evaluation judgment for reducing the event handler statement to process an event, and a global judgment where the configuration consists of all the RSMs executing concurrently. We first present the local evaluation judgment.

Local evaluation judgments are of the form $E; F; L; s \to E_1; F_1; L_1; s_1$, where the syntax

$$\boxed{F; L \vdash e \Downarrow n} \qquad \boxed{E; F; L; s \to E_1; F_1; L_1; s_1}$$

$$\textsc{E-var}\ \frac{}{F; L \vdash x \Downarrow L[x]} \qquad \textsc{E-volatile}\ \frac{}{F; L \vdash f \Downarrow F[f]} \qquad \textsc{E-persistent}\ \frac{}{F; L \vdash \texttt{load}\ f \Downarrow F[f]}$$

$$\textsc{E-binop}\ \frac{F; L \vdash v_i \Downarrow n_i \qquad n = n_1 \oplus n_2}{F; L \vdash v_1 \oplus v_2 \Downarrow n} \qquad \textsc{E-star}\ \frac{}{F; L \vdash \star \Downarrow n}$$

$$\textsc{L-store}\ \frac{F; L \vdash e \Downarrow n}{E; F; L; \texttt{store}\ f\ e \to E; F[f \mapsto n]; L; \texttt{skip}} \qquad \textsc{L-if}\ \frac{F; L \vdash v \Downarrow n \qquad (n \neq 0 \Rightarrow s = s_1) \wedge (n = 0 \Rightarrow s = s_2)}{E; F; L; \texttt{if}\ v\ s_1\ s_2 \to E; F; L; s}$$

$$\textsc{L-create}\ \frac{\mathsf{fresh}\ n_r \qquad E_1 = (n_r, n_C, 0), E}{E; F; L; \texttt{create}\ x\ C \to E_1; F; L[x \mapsto n_r]; \texttt{skip}} \qquad \textsc{L-send}\ \frac{F; L \vdash v_i \Downarrow n_i \qquad E_1 = (n_1, n_2, n_3), E}{E; F; L; \texttt{send}\ v_1\ v_2\ v_3 \to E_1; F; L; \texttt{skip}}$$

Figure 4 R_{SM} local semantics.

for E, F, and L is shown in Figure 3. F and L are field map and local environment, mapping fields and local variables to values. L contains three special variables x_s, x_e, and x_p that map to the source RSM, event type, and the payload of the *current* event that is being processed; these fields are initialized in the global judgment.

E is a list of output events. An event is a triple of the form (r, n_e, n_p), where r is the destination RSM id, n_e is the event type, and n_p is the event payload. Notably F, L, and E are all non-persistent. Their interaction with the persistent state happens in the global judgment. Statement reduction uses an auxiliary expression evaluation judgment of the form $F; L \vdash e \Downarrow n$. Statements at runtime include an additional `skip` form to denote the terminal statement.

Figure 4 shows the selected rules for statement reduction and expression evaluation. The expression rules are all standard, notably rule E-STAR non-deterministically evaluates the $\star$ expression to some integer n. Most of the statement reduction rules are also standard. For example, rule L-STORE uses the expression evaluation form to evaluate e, and stores the result in the field map F. Rule L-IF branches based on the evaluated value of v. Rule L-CREATE simply records the creation request in the output events list with a special event type n_C. Finally, rule L-SEND evaluates each of the **send** arguments, and updates the output event list E.

3.1.2 Global evaluation judgment

Global evaluation judgment has the form $S \vdash M; \Pi \longrightarrow M_1; \Pi_1$. M and Π are maps with RSM ids as domains. The map M maps the RSM ids to local configurations $E; F; L; s; b$, where E, F, L, and s come from the local judgment, and b is a (volatile) bit that is 1 if the machine is currently processing an event or 0 otherwise. We will also write F_p and F_v to denote the F map components for **persistent** and **volatile** fields respectively. The map Π maps each RSM id to its class C and persistent storage, i.e. $C; I; O; P; T$, where I is the inbox persisting the incoming events, O is the outbox persisting the outgoing events, P is the persistent fields map, and T is the *trace* of the RSM that records its observable behavior; the trace T is ghost and is only used to state and prove the failure transparency theorem. The grammar for I, O, and T is same as that of the event list E, while persistent field map P is a field map like F. Finally, S is the signature that maps class C to its definition.

As shown in Figure 1b, each RSM (a) reads an event from its input queue, (b) processes it using its handler statement, (c) commits the events generated and the persistent field map in its persistent store, (d) empties the outbox in the persistent store, and starts from (a) again. At each of these steps, the machine can crash and recover, where all of its non-persistent data (including the local state E, F, L) is lost. The global semantics essentially implements this state machine for each RSM, while executing the RSMs concurrently with each other.

Figure 5 shows the global semantics judgment. In all the rules, one of the machines r takes the step. Using the Rule G-START, a machine r enters the event handler for processing the head event in the input event queue. The local state of the machine currently is *at rest*, i.e. $M(r).s = \texttt{skip}$ and $M(r).b = 0$, as well as the outbox $\Pi(r).O$ is empty. The rule creates the local environment L (using the initL auxiliary function, shown in the same figure), by initializing the local variables as per the RSM definition $S(C)$, and also adding the mappings for event source, event type and event payload (x_s, x_e and x_p). The local state of the machine is changed to process the handler statement s and the bit b is set to 1. The persistent store Π is left unchanged.

Rule G-LOCAL shows the local evaluation rule, where a machine r takes a local step by executing the event handler. The rule uses the local semantics judgment in the premise, and updates $M(r)$ accordingly.

G-START

$$\dfrac{\begin{array}{c}M(r) = \cdot; F; \_; \texttt{skip}; 0 \quad \Pi(r) = C; \_, (n_s, n_e, n_p); \cdot; P; \cdot \\ F = F_p \cup F_v \quad F_p = P \\ L = \mathsf{initL}(C, n_s, n_e, n_p) \quad s = \mathsf{handler}(C)\end{array}}{S \vdash M; \Pi \longrightarrow M[r \mapsto \cdot; F; L; s; 1]; \Pi}$$

G-LOCAL

$$\dfrac{\begin{array}{c}M(r) = E; F; L; s; 1 \quad \Pi(r).O = \cdot \\ E; F; L; s \to E_1; F_1; L_1; s_1 \\ M_1 = M[r \mapsto E_1; F_1; L_1; s_1; 1]\end{array}}{S \vdash M; \Pi \longrightarrow M_1; \Pi}$$

G-COMMIT

$$\dfrac{\begin{array}{c}M(r) = E; F_p \cup F_v; L; \texttt{skip}; 1 \quad \Pi(r) = C; I, \_; \cdot; \_; T \\ M_1 = M[r \mapsto \cdot; F_p \cup F_v; L; \texttt{skip}; 0]\end{array}}{S \vdash M; \Pi \longrightarrow M_1; \Pi[r \mapsto C; I; E; F_p; T]}$$

G-CREATE

$$\dfrac{\begin{array}{c}M(r) = \_; \_; \_; \texttt{skip}; 0 \\ \Pi(r).O = \_, (r_1, n_C, \_)\end{array}}{S \vdash M; \Pi \longrightarrow M; \mathsf{create}(r, r_1, C, \Pi)}$$

G-SEND

$$\dfrac{M(r) = \_; \_; \_; \texttt{skip}; 0 \quad \Pi(r).O = \_, (r_1, n_e, n_p)}{S \vdash M; \Pi \longrightarrow M; \mathsf{send}(r, r_1, n_e, n_p, \Pi)}$$

G-RESET

$$\dfrac{\begin{array}{c}\Pi(r) = C; \_; \_; P; \_ \\ M_1 = M[r \mapsto \cdot; \mathsf{resetF}(C, P); \cdot; \texttt{skip}; 0]\end{array}}{S \vdash M; \Pi \longrightarrow M_1; \Pi}$$

$$\begin{array}{lcl}
\mathsf{initL}(C, n_s, n_e, n_p) & = & \texttt{let } S(C) = \texttt{class } C\ \{\_; \_; \{\overline{x := n}; \_\}\}\ \texttt{in} \\
 & & (\overline{x \mapsto n}, x_S \mapsto n_s, x_e \mapsto n_e, x_p \mapsto n_p) \\
\mathsf{handler}(C) & = & \texttt{let } S(C) = \texttt{class } C\ \{\_; \_; \{\_; s\}\}\ \texttt{in } s \\
\mathsf{create}(r, r_1, C, \Pi) & = & \texttt{let } S(C) = \texttt{class } C\ \{\overline{\texttt{persistent } f := n}; \_; \_\}\ \texttt{in} \\
 & & \texttt{let } \Pi(r) = C_r; I; O; P; T\ \texttt{in} \\
 & & \texttt{let } \Pi_1 = \Pi[r \mapsto C_r; I; \mathsf{tail}\ O; P; (r_1, n_C, 0), T]\ \texttt{in} \\
 & & \Pi_1[r_1 \mapsto C; \cdot; \cdot; \overline{f \mapsto n}; \cdot] \\
\mathsf{send}(r, r_1, n_e, n_p, \Pi) & = & \texttt{let } \Pi(r) = C; I; O; P; T\ \texttt{in let } \Pi(r_1) = C_1; I_1; O_1; P_1; T_1\ \texttt{in} \\
 & & \texttt{let } \Pi_1 = \Pi[r \mapsto C; I; \mathsf{tail}\ O; P; (r_1, n_e, n_p), T]\ \texttt{in} \\
 & & \Pi_1[r_1 \mapsto C_1; (r, n_e, n_p), I_1; O_1; P_1; T_1] \\
\mathsf{resetF}(C, P) & = & \texttt{let } S(C) = \texttt{class } C\ \{\_; \overline{\texttt{volatile } f := n}; \_\}\ \texttt{in } P, \overline{f \mapsto n}
\end{array}$$

Figure 5 R_{SM} global semantics.

Once a machine r has finished executing the event handler for the head input event, it uses the rule G-COMMIT to commit the persistent state. In the rule, the local state of the machine has reached the end of handler execution ($M(r).s = \texttt{skip}$ and $M(r).b = 1$). $M(r)$ is changed by setting the bit b to 0 and the local event list is reset to empty. The changes to $\Pi(r)$ are: (a) the head event is removed from $\Pi(r).I$, (b) the output event list E from the local state is committed to the outbox $\Pi(r).O$, and (c) the new values of the `persistent` variables from the local state are committed to $\Pi(r).P$. The (ghost) trace of the machine $\Pi(r).T$ remains unchanged; the machine next proceeds to send the events out of the outbox, and append the trace accordingly.

Rule G-CREATE handles the create event (rule L-CREATE, Figure 4). The auxiliary function create updates the persistent store Π. For the creator machine r, it removes the create event from the outbox $\Pi(r).O$, and adds it to the ghost trace $\Pi(r).T$. For the new machine r_1, it initializes the persistent store by reading off the initial persistent variables map from the signature $S(C)$. Rule G-SEND sends an event from machine r to r_1. The auxiliary function send removes the event from the outbox of r, and adds it to the ghost trace, as well as to the inbox of r_1. The rule models the exact-once delivery network module.

Finally, a machine r can fail at any point in the execution. The rule G-RESET models the machine reset. As expected, upon reset, the local volatile state, including the event list E, `volatile` variables, environment L, are all lost. The fields map in the local state is re-initialized (using resetF) by reading off the `persistent` variables from $\Pi(r)$ and `volatile` variables from the signature $S(C)$. The bit b is also set to 0. We next present our main theorem of failure transparency.

3.2 Failure transparency

To state the theorem, we first define a notion of equivalence for local states $M(r)$. Below, r is an RSM id.

▶ **Definition 3.1** (Equivalence of local states). *Two local states, $M_1(r)$ and $M_2(r)$ are equivalent, written as $M_1(r) \cong M_2(r)$, if they are equal in all components, except for the volatile class fields in their field maps, i.e. $M_1(r).E = M_2(r).E$, $M_1(r).F_p = M_2(r).F_p$, $M_1(r).L = M_2(r).L$, $M_1(r).s = M_2(r).s$, and $M_1(r).b = M_2(r).b$.*

Our failure transparency theorm relies on non-interference of persistent state from volatile fields. We formally state the property below (we use $\longrightarrow_r$ to denote the machine r taking a step):

▶ **Property 3.2** (Non-interference). *Let $M; \Pi \longrightarrow_r^* M_1; \Pi$ be a run, s.t. each step in the run is a* G-LOCAL *step taken by machine r, and M_1 is terminal (i.e. $M_1(r).s = \texttt{skip}$). Then, $\forall M'.\ M'(r) \cong M(r)$, there exists M_1' s.t. $M'; \Pi \longrightarrow_r^* M_1'; \Pi$ where $M_1'(r) \cong M_1(r)$ and each step is a* G-LOCAL *step.*

In [29], we present an information-flow type system for R_{SM} that provides this non-interference property for well-typed programs. Note that non-determinism in our language does not raise any complications, since to get this property, we can essentially *replay* the non-deterministic choices from the run in the premise to the run in the conclusion.

Given Property 3.2, we are now ready to state the failure transparency theorem. We consider a run of a machine that processes an event end-to-end. We prove that, given any such run that includes failures (i.e. the rule G-RESET), we can construct a run without failures, but with same observable traces T.

▶ **Theorem 3.3** (Failure transparency). *Let $M; \Pi \longrightarrow_r^* M_p; \Pi_p \longrightarrow_r M_c; \Pi_c \longrightarrow_r^* M_1; \Pi_1$, where $M; \Pi$ is ready for a machine r (i.e. it satisfies the premises of the* G-START *rule), and*

1. *all steps in $M; \Pi \longrightarrow_r^* M_p; \Pi_p$ are either* G-START, G-LOCAL, *or* G-RESET,
2. *$M_p; \Pi_p \longrightarrow_r M_c; \Pi_c$ is a* G-COMMIT *step, and*
3. *all steps in $M_c; \Pi_c \longrightarrow_r^* M_1; \Pi_1$ are either* G-CREATE, G-SEND, *or* G-RESET

Then, $\forall M'.M'(r) \cong M(r)$, there exists M_1' s.t.

(a) $M'; \Pi \longrightarrow_r^* M_1'; \Pi_1$,
(b) *none of the steps in (a) are* G-RESET, *and*
(c) $M_1'(r) \cong M_1(r)$

Crucially, Π_1, and hence the trace of machine r remains same in the conclusion of the theorem. Thus, we prove that the machine run with failures is a refinement of the machine run without failures with respect to its observable behavior.

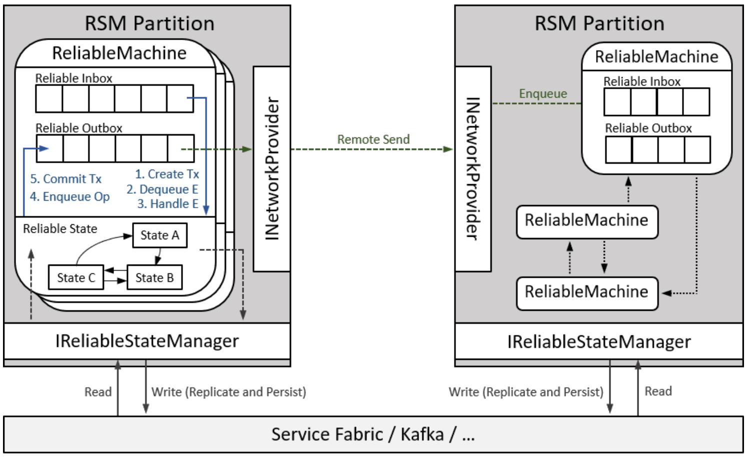


Figure 6 The Reliable State Machines implementation.

4 Implementation

This section describes an instantiation of RSMs as a .NET object-oriented programming framework. The framework is split into two logical parts: the *frontend* and the *backend*. The frontend implements the programmer-facing APIs while the backend is responsible for the distributed-system aspects, including state persistence and inter-machine communication. An illustration of the RSM architecture is shown in Figure 6.

The frontend exposes an `RSM.ReliableMachine` base class. An RSM is programmed as a class that derives from `ReliableMachine`. An RSM instance is an object of such a class and event handlers are implemented as methods of the class. The base class implements the functionality to drive a state machine. The state machine structure is based on `P#`, similar to the word-count code shown in Section 2. We focus the discussion here on the reliability aspects of RSMs. The frontend also provides a runtime, `RSM.ReliableMachineRuntime`, that implements the APIs for creating RSMs and sending messages between them. Each RSM carries a reference to the runtime in order to invoke these APIs. The runtime is also responsible for `rsmId` management, ensuring that each RSM is associated with a unique id throughout its lifetime.

The frontend provides two generic types for declaring local persistent state of an RSM: `RSM.PersistentRegister<T>` and `RSM.PersistentDictionary<TKey,TValue>`. The former implements a `Get-Put` interface for getting access to the underlying `T` object, similar to the `load` and `store` semantics of our formal language. The object is automatically serialized (on `Put`) and deserialized (on `Get`) in the background.[2] The `PersistentDictionary` type is similar, although it additionally allows access to individual keys. This has the advantage that if an RSM handler only accesses a few keys, then only those keys (and their corresponding values) are serialized and stored, without having to serialize the entire dictionary.

The programmer can declare fields inside an RSM class with these `Persistent` types to get access to the persistent local state. Any other fields in the class are treated with volatile semantics. The current state of the state machine is maintained in a `PersistentRegister` so that the RSM resumes operation from the correct state on failover.

[2] We use the `protobuf-net` serializer in RSMs, although other mechanisms are possible.

RSMs, once created, continually listen to incoming messages, until they are explicitly halted. ReliableMachine exposes an option of halting the RSM. The runtime reclaims all resources associated with an RSM when it halts.

The runtime works against RSM.IReliableStateManager and RSM.INetworkProvider interfaces, each of which are implemented by the backend. The IReliableStateManager interface is responsible for creating the inbox and outbox queues, as well as to back the persistent fields of an RSM. The INetworkProvider interface allows communication between RSMs. We provide two backend implementations: one using Azure Service Fabric (Sections 4.1 and 4.2) and the other one using Apache Kafka (Section 4.3). We additionally provide a P#-based backend for the purpose of high-coverage systematic testing (Section 4.4).

4.1 Azure Service Fabric backend

Background. Azure Service Fabric (SF) [4] provides infrastructure for designing and deploying distributed services on Azure. A user begins by setting up an SF cluster on a required number of Azure VMs. SF sets up a replicated on-disk storage system on the cluster. An application deployed to an SF cluster benefits from having access to co-located storage, instead of having to access a remote storage system. The store uses primary-secondary-based replication. The user can choose a replication factor (say R) in which case each update to the store is applied to R replicas, with each replica located on a different machine. Updates are only allowed on the primary, after which they are propagated to the secondaries.

SF provides various means of programming a service for deployment to an SF cluster. The most relevant to our discussion is a stateful application called *reliable services* [8]. Such an application consists of multiple *partitions* [6]; each partition roughly resembles an individual process constituting the failure domain for the application. Each partition is associated with its own primary and $R - 1$ secondaries. The partition's process is co-located with the primary. (Thus, an application with N partitions will have a total of N primaries and $RN - N$ secondaries, distributed evenly across the SF cluster.) From the programmer's perspective, each partition gets its own StateManager [9] object that provides access to its store. When a machine carrying a primary fails, one of its secondaries is promoted to become a primary and the corresponding partition is re-started on the new primary. A new secondary is elected and brought up to date in the background. Thus, a machine failure results in restarting of any partition located on it, but all data written to their StateManager is still available on restart.

The SF StateManager provides APIs for transactional access to storage [7]. A user can create a transaction, use it to perform reads and writes to the store, and then commit it. SF transactions have the database ACID semantics [21], i.e., they are atomic, consistent, isolated, and durable with respect to other transactions. As a form of convenience, the user can access the store via a dictionary interface (IReliableDictionary) and a queue interface (IReliableQueue). These interfaces are shown in Listing 4. (We qualify the SF interfaces with SF and the RSM types with RSM to avoid any confusion.) The SF.IReliableQueue interface, for example, supports enqueue and dequeue operations, each of which require the associated transaction. (These are awaitable C# methods [3], hence the return type Task.) These operations appear to take place (with respect to other transactions) only when their associated transaction is committed. A transaction can span multiple of these reliable collections. The method DictionaryToQueueAtomicTransfer in Listing 4 illustrates an atomic transfer of a value from a dictionary to a queue: it reads from a dictionary and writes to the queue in the same transaction.

```
interface SF.IReliableDictionary<TKey, TValue> {
  Task SetAsync(SF.ITransaction, TKey, TValue);
  Task<ConditionalValue<TValue>> TryGetValueAsync(SF.ITransaction, TKey);
}

interface SF.IReliableQueue<T> {
  Task EnqueueAsync(SF.ITransaction, T);
  Task<ConditionalValue<T>> TryDequeueAsync(SF.ITransaction);
}

async void DictionaryToQueueAtomicTransfer(SF.IReliableDictionary<int, int> D,
SF.IReliableQueue<int> Q)
{
   int key = ...
   using (var tx = StateManager.CreateTransaction())
   {
     var v = await D.TryGetValueAsync(tx, key);
     if(v.HasValue) {
       await Q.EnqueueAsync(tx, v.Value);
     }
     await tx.CommitAsync();
   }
}
```

Listing 4 Reliable collection interfaces of service fabric (shown partially) with sample usage.

RSM backend. We can now describe a vanilla implementation of RSMs using SF. Various optimizations are described in Section 4.2. An RSM program deploys as a stateful service on an SF cluster. A single partition contains exactly one instance of `RSM.ReliableMachineRuntime` that may host any number of RSM instances. `RSM.IReliableStateManager` is implemented as a wrapper of the SF `StateManager` and `RSM.INetworkProvider` is implemented using the SF remoting library for RPC communication [5].

The runtime remembers all hosted RSM instances in a persistent dictionary of the type `SF.IReliableDictionary<rsmId, bool>`. When a partition comes up (or fails over), it creates a new runtime, which then immediately reads this dictionary to identify the set of RSMs that it had hosted before failure (if any). It then re-creates the RSMs with the same ids. All persistent state associated with an RSM is attached to its id so that an RSM can rehydrate its state on failover as long as it retains its id.

The types `RSM.PersistentDictionary` and `RSM.PersistentRegister` are implemented as wrappers of `SF.IReliableDictionary`. The RSM types hide SF transactions from the programmer. The inbox and outbox are just SF reliable queues (`SF.IReliableQueue`). An RSM executes as an event-handling loop. Each iteration of the loop constructs an SF transaction (say, `Tx`) and performs a dequeue on the inbox using the transaction. If it finds that the queue is empty, the loop terminates and is woken up later only when a message arrives to the RSM. (This ensures that the RSM takes no compute resources when it has no work to perform.) If a message is found in the inbox, then the RSM goes on to execute the corresponding handler. Any access made by the handler to a persistent field gets attached with the same transaction `Tx`. Sending a message m to an RSM r is performed as an enqueue of the pair (m, r) to the outbox queue, also on the same transaction `Tx`. When the handler finishes execution, the RSM commits `Tx` and repeats the loop to process other messages in the inbox. Using the same transaction throughout the lifetime of a handler ensures that all effects of processing a message happen atomically with the dequeue of that message.

Networking and exact-once delivery. RSMs have two additional background tasks: the first one is responsible for emptying the outbox, and the other one listens on the network for incoming messages to add them to the inbox. These tasks are spawned on-demand as work arrives in order to avoid unnecessary polling. These tasks co-operate to ensure exact-once delivery between RSMs, even under network failures or delays (as long as the connection is eventually established).

```
do:
  create transaction tx1
  (m, r2) = Outbox.Dequeue(tx1);
  c = SendCounter[r2].Get(tx1);
  SendCounter[r2].Put(c + 1, tx1);
  do:
     send (m, c, r1) to r2
  repeat until an ack is received within
       timeout
  commit tx1
repeat forever
```

Listing 5 Outbox draining task for RSM $r1$.

```
On receiving (m, c, r1):
  create transaction tx2
  d = ReceiveCounter[r1].Get(tx2);
  if d == c then:
     ReceiveCounter[r1].Put(d+1, tx2);
     inbox.Enqueue(m, tx2);
  send ack back to r1;
  commit tx2
```

Listing 6 Input ingestion procedure for RSM $r2$.

The runtime maintains two reliable dictionaries called `SendCounter` and `ReceiveCounter` that map `rsmId` to `int`. Pseudo-code for the outbox-draining task of an RSM with id `r1` is shown in Listing 5. It creates a transaction `tx1` and performs a dequeue on the outbox to obtain the pair $(m, \texttt{r2})$ of message and destination, respectively. It then sends the tuple $(\texttt{r1}, \texttt{SendCounter[r2]}, m)$ over the network to `r2` and waits for an acknowledgement. If it gets the acknowledgement within a certain timeout period, it increments `SendCounter[r2]` and commits `tx1` to complete the message transfer. If it times-out waiting for an acknowledgement from `r2`, it retries by sending the message again.

The automatic retry implies that the receiver might get duplicate messages; however, each such duplicate will be attached with the same counter value, which the receiver can use for de-duplication. This is achieved in the input-ingestion procedure shown in Listing 6. The receiver `r2`, when it gets the tuple $(m, c, \texttt{r1})$, first checks if c equals `ReceiveCounter[r1]`. If so, it increments `ReceiveCounter[r1]` and enqueues m to its inbox. If not, it drops the message because its a duplicate. Regardless, it always sends an acknowledgement back to `r1`.

Note that each of the tasks including input-ingestion, outbox-draining, and the event-handling, use their own transactions that are different from each other. This enables the RSM to run these tasks completely independently and in parallel to each other. SF transactions provide ACID semantics, so concurrent enqueue and dequeue operations on queues are safe.

RSM creation. When an RSM `r1` wishes to instantiate a new RSM of class C, it first creates a globally unique `rsmId` r. This creation can be done in several ways. Our implementation uses inter-partition communication to first decide the partition that will host the newly created RSM. It then grabs a unique counter value from that partition. The pair of partition name and unique counter value on that partition makes the `rsmId` globally unique. Once this value r is obtained, `r1` enqueues the pair (r, C) to its outbox. No RSM is actually created until the pair is committed to the outbox. If `r1` fails before committing, then the value r is lost forever. When `r1` is restarted, it will construct a new (but still globally unique) id.

The outbox-draining task of `r1`, when it picks up a tuple (r, C), will send a message to the partition on which r is located. Like before, this message is sent repeatedly until acknowledged. On the receipt of this message, the RSM runtime instantiates a new RSM of type C *only if* it does not already have an RSM associated with r. If it does have such an RSM, then it drops the message because it must be a duplicate request, one that it has carried out already. The recipient sends back an acknowledgement to the sender regardless.

4.2 Optimizing the SF backend

The following lists some of the most important performance optimizations that we found useful for the SF backend.

Shared inbox and outbox. Creating a separate reliable queue for the inbox and outbox of each RSM does not scale well unfortunately, especially when the application creates a large number of RSMs. Each creation incurs an I/O operation. To optimize the RSM creation time, we instead use a single data structure that is shared across all RSMs in the same partition: one for all inboxes and one for all outboxes. These shared structures are implemented as an `SF.IReliableDictionary` whose key is a tuple of `rsmId` and an index (`long`). Each RSM maintains its own head and tail indices, denoting the contiguous index range that contains its inbox or outbox contents. An RSM `r1`, for instance, can enqueue m to its outbox by writing it to the key ($\texttt{r1}, \text{tail}$) and incrementing tail. For efficiency, the head and tail values are only kept in-memory. On failover, the RSM runtime reads through the shared dictionary to identify the per-RSM head and tail values, before it instantiates the RSMs with these values. Additional care is required to ensure proper synchronized access to head and tail values by the various tasks associated with an RSM. Using these shared structures allowed us to significantly reduce machine creation time (Section 6.1).

Batching. We use batching in various forms to optimize overall throughput (Section 6.1). First, the event-handling loop of an RSM can dequeue multiple messages from its inbox in the same transaction and process all of them (sequentially, one after the other) before committing all of their effects together. The commit is a high-latency operation because SF must replicate all updates to the secondaries and wait for a quorum to acknowledge. This form of inbox-batching helps hide some of this latency. Second, the outbox-draining task can dequeue multiple messages from the outbox in the same transaction, and as long as they are intended for the same destination partition, send them over the network as a batch.

Non-persistent inbox. Sending a message m from RSM `r1` to `r2` requires several I/O operations: `r1` first commits m to its outbox, next it sends m over the network to `r2`, and finally `r2` commits m to its inbox. Interestingly, we can do away with a persistent inbox and only keep it in memory without sacrificing any of the RSM framework guarantees. Our optimization works as follows. The input-ingestion task of `r2` simply enqueues m to an in-memory inbox but it does not immediately send an acknowledgement back to `r1`. Instead, `r2` waits until it is done processing m. After `r2` commits the effects of processing m to its own outbox, it sends the acknowledgement back to `r1`, after which `r1` will remove m from its outbox. This is safe since the message sits in the (persistent) outbox of `r1` until `r2` is done processing it.

4.3 Kafka backend

Apache Kafka [2, 25] is a popular distributed messaging platform that has been used in large production systems by companies such as Netflix and Spotify [23]. Kafka supports fault-tolerant named sequence of messages called *topics*. A *producer* appends messages to the tail of a topic. A message is retained in the topic for a predefined period of time, after which it is deleted automatically. In order to read a message, a *consumer* subscribes to the topic and maintains a per-topic index, referred to as the consumer's *offset*. The read cycle involves the consumer reading the message at its offset, incrementing the offset, and then storing the new offset value in a topic of its own called the *offset-topic*. Kafka supports different consumers to read from different offsets of a topic concurrently. Starting in version v0.11.0, Kafka introduced the notion of cross-topic *transactions*. These allow a producer to write to multiple topics transactionally. Consumers cannot observe the writes made in a transaction until the transaction commits. A Kafka *stream* is a combination of a Kafka producer and consumer: it consumes messages from an input topic and publishes messages to one or more output topics. Kafka supports building stateful applications on top of streams via a key-value state store and convenient Java/Scala APIs. Exact-once processing of messages can be achieved by transactionally writing the offset, state and published messages to their respective topics.

Kafka-based RSMs. A Kafka RSM (K-RSM) has an associated *inbox topic*, and a *state topic* for its persistent local state. The RSM also maintains its read offset into the inbox as part of its persistent local state. An RSM executes as follows: it reads a message m from the inbox at its read offset and starts a Kafka transaction Tx. It then runs the handler code for m. Any changes to the persistent local state are written to the state topic under Tx.

Any message sends are written directly to the inbox topics of the receiver K-RSMs, also under Tx. Finally, the incremented offset is written to the state topic and the transaction Tx is committed. Note that there was no need to have an *outbox*: Kafka transactions ensure that the effects of processing a message by one RSM are not observed by other RSMs until its transaction commits. (SF transactions, on the other hand, cannot span across reliable collections in different partitions, which is why we needed an outbox for the SF backend.) Restart of an RSM simply involves recovering its state from the state topic that additionally provides it the read offset of the last un-processed message.

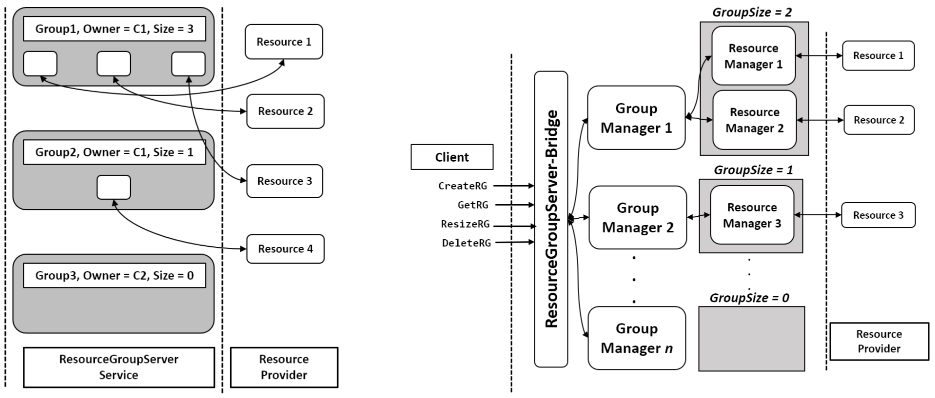
A user begins by starting a Kafka cluster, configured to their own requirements. The K-RSM backend then attaches to the cluster to execute the RSM program. Unlike SF reliable collections, Kafka topics must be preallocated to a fixed number, which would typically be much smaller than the number of RSM instances that a program may create. The K-RSM backend shares a single topic across multiple RSM instances, which works because each RSM maintains its own offset value. The assignment of RSMs to topics is currently done in a simple round-robin fashion but more sophisticated policies are possible as well. Similar to the SF backend, messaging in Kafka benefits greatly from batching: both when writing to a destination topic and when reading from the inbox topic.

4.4 P# backend

We additionally designed a backend for the purpose of testing RSM programs. The backend does not support distribution; it simulates the entire program execution in a single process. The backend essentially translates an RSM program to a P# program for systematic testing against a specification. We first briefly summarize P# capabilities [15].

P# provides an in-memory framework for implementing concurrent programs; it does not provide any support for distribution or persistence. A P# program consists of multiple state machines that communicate via messages. The PSharpTester tool takes a P# program as input and repeatedly executes it multiple times. It takes over the scheduling of the program so that it can search over the space of all possible interleavings. PSharpTester employs a state-of-the-art portfolio of search strategies that has proven to be effective in finding bugs quickly [18, 16]. A user can write a specification in the form of a monitor that is checked by the PSharpTester in each execution of the program. Both safety and liveness specifications [28] are supported.

The P# backend for RSMs allows one to write specification monitors in the same way as P# and test their correctness using PSharpTester. It is worth noting that the backend is designed with the intention of testing the user logic as opposed to the RSM runtime itself. For this, the backend ensures that only the concurrency (and complexity) in the user program is exposed to the PSharpTester; the concurrency inside the runtime (which is useful for gaining performance) is disabled.



(a) Overview of the microservice. **(b)** Architecture of the RSM based implementation.

Figure 7 Achitecture of the ResourceGroupServer service.

An RSM translates almost directly to a `P#` machine, with the following modifications. First, the backend provides mock implementations for all persistent types (simulated in-memory for efficiency). Second, the three tasks associated with an RSM (i.e., input-ingestion, event-handling and outbox-draining) are run sequentially, one after the other. Third, the exact-once network delivery algorithm is assumed correct, so the outbox-to-inbox transfer is done atomically (and in-memory).

An important aspect of the backend is simulating failures in the RSM program. The failure-transparency property of RSMs from Theorem 3.3 crucially helps here: as long as the programmer correctly uses the volatile state as per Property 3.2, failures have no effect at all on the observable behavior of an RSM. Thus, the backend only needs to test for Property 3.2 on the program. This is done as follows. The backend, at the time it is about to commit a transaction in the event-handling loop of an RSM, non-deterministically chooses to carry out the following steps: (1) record the persistent state of the RSM (both local state and outbox), (2) reset the volatile state of the RSM, (3) abort the transaction, thus requiring the RSM to re-process the input message, and (4) when the RSM reaches the commit point again, assert that the persistent state equals the recorded state. If a failure of this assertion is reported by the `PSharpTester`, the programmer is informed of the incorrect usage of volatile state.

5 Case-Study: ResourceGroupServer

We used the RSM framework to redesign the core functionality of an in-production service on Microsoft Azure, which we refer to as *ResourceGroupServer*. This section describes the operations supported by the service (Section 5.1) and its implementation using RSMs (Section 5.2), highlighting the gains in programmability and testing of the service. We demonstrate scalability of the RSM code in Section 6.2.

5.1 Service description

The ResourceGroupServer (RGS) is a generic resource management service. A cloud platform will typically provide various kinds of compute and storage resources, for instance, virtual machines, that can be used in conjunction by a user to implement certain functionality. RGS is designed to offer a convenient abstraction over a low-level resource provider to maintain a collection of resources. A user can request the RGS for a set of n resources (called a *group*). The RGS then calls into the resource provider to allocate these resources.

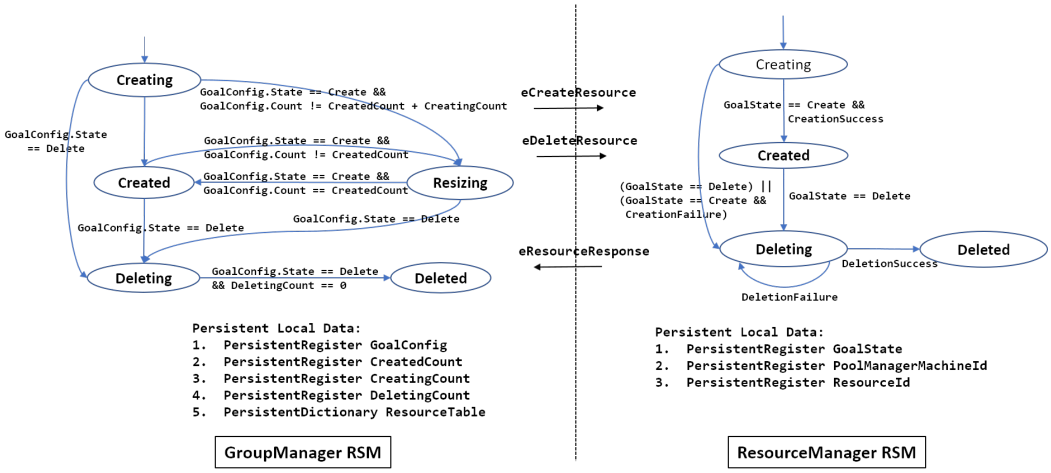

Creating
GoalConfig.State == Create &&
GoalConfig.Count != CreatedCount + CreatingCount
GoalConfig.State
== Delete
GoalConfig.State == Create &&
GoalConfig.Count != CreatedCount
Created
GoalConfig.State == Create &&
GoalConfig.Count == CreatedCount
Resizing
GoalConfig.State == Delete
GoalConfig.State == Delete
Deleting
GoalConfig.State == Delete
&& DeletingCount == 0
Deleted
Persistent Local Data:
1. PersistentRegister GoalConfig
2. PersistentRegister CreatedCount
3. PersistentRegister CreatingCount
4. PersistentRegister DeletingCount
5. PersistentDictionary ResourceTable
GroupManager RSM
eCreateResource
eDeleteResource
eResourceResponse
Creating
GoalState == Create &&
CreationSuccess
Created
(GoalState == Delete) ||
(GoalState == Create &&
CreationFailure)
GoalState == Delete
Deleting
DeletionSuccess
Deleted
DeletionFailure
Persistent Local Data:
1. PersistentRegister GoalState
2. PersistentRegister PoolManagerMachineId
3. PersistentRegister ResourceId
ResourceManager RSM

Figure 8 The group manager and resource manager RSM state machines.

Fig. 7a shows a high-level view of RGS. Each group has a designated *owner* and supervises a number of resources. Individual resources can turn *unhealthy* (e.g., a VM becomes unresponsive), in which case, it is the responsibility of RGS to explicitly delete that resource and allocate a new one to ensure that each group eventually reaches its desired size. Also, there should be no *garbage* resources: one that is allocated by the resource provider but is not associated with any group.

A client C can fire a group creation request to RGS, with the desired number of resources n as a parameter. In response, RGS creates a fresh group, owned by C, with n resources in it. The client can query the health, resize or delete any existing group that it owns.

RGS must be responsive and scalable. It must be able to handle creation requests from multiple clients at the same time. Further, the creation of a group itself should not add much overhead over the actual allocation of the resources. RGS should also tolerate failures: if it crashes, it should not lose information about the groups that it had already created, or was in the middle of creating. For instance, if a requested group of size 10 had reached size 3 when the RGS crashed, it must resume and allocate only the remaining 7.

5.2 RSM-based ResourceGroupServer

We implemented RGS using RSMs. We denote this implementation as `RsmRgs`. It supports the core functionality that was described in the previous section. In comparison, the real production service (denoted `ProdRgs`) offers a richer API to its clients, but the additional features are unrelated to matters of reliability or concurrency. Fig. 7b shows the high-level architecture of `RsmRgs`. There are two RSM types: one called the *resource manager* (RM) that is responsible for the lifetime of a single resource, and another called *group manager* (GM) that is responsible for the lifetime of a single group. This division ensures that the complexities of dealing with the external resource provider are limited to the RM. Future changes to the resource provider APIs will likely not impact the GM.

A client can issue requests such as `CreateRG`, `GetRG`, `ResizeRG` or `DeleteRG` to `RsmRgs`. These requests are translated to messages that are directed to the GM that owns the corresponding group. The state machine structures of RM and GM are shown pictorially in Figure 8. We explain the functioning of these RSMs by tracing through the `CreateRG` operation.

```
void ScaleUp(RsmId gmId, int toCreate)
{
  for (int i = 0; i < toCreate; i++) {
    // Start off an RM to allocate a fresh resource.
    var id = create(ResourceManager);
    send (id, eCreateResource(gmId, ResourceGoalState.Create));
    // Record the creation in the resource table, and we're done.
    store (ResourceTable[id], ResourceState.Creating);
    store (CreatingCount, (load CreatingCount) + 1);
  }
}
```

Listing 7 `ScaleUp` operation to create resources in a group.

In response to a client's creation request, `RsmRgs` creates a new GM instance. Each such instance maintains three counters: `CreatingCount`, `CreatedCount` and `DeletingCount` which are, respectively, the number of resources that are under creation, already created, and under deletion. The GM additionally maintains a `GoalConfig` that specifies the desired number of resources in the group (`Count`), and the intended `State` of the group (either `Create` or `Delete`). Finally, GM also maintains a dictionary `ResourceTable` containing the `rsmId`s of all the RM instances that it owns.

A GM instance starts off in the `Creating` state with an empty `ResourceTable` and each counter set to 0. Its `GoalConfig` will get initialized to the group size that was requested by the client (on receiving the creation request) and the RSM will transition to its `Resizing` state realizing that it does not have enough resources created. In the resizing state, the RSM looks at the difference between `GoalState.Count` and `CreatingCount` + `CreatedCount`, say m, and fires off the operation `ScaleUp(gmId, m)` whose code is shown in Listing 7, where `gmId` is the `rsmId` of the current GM instance. We note that this entire operation is devoid of any failover or retry logic: the GM does not have to worry about failures of the machine hosting it, or about the failures of the RM instances that it creates. The runtime ensures that the exact number of instances requested will be created eventually (and no more).

An RM instance reliably persists the handle (`GroupManagerMachineId`) to the GM instance that created it, the goal state (`GoalState`) that is either `Create` or `Delete`, and the resource identifier (`ResourceId`) returned by the resource provider. An RM starts off in the `Creating` state, fires off a request to the resource provider, which if successful (`CreationSuccess`) causes a transition to the `Created` state. It then informs the GM about successful creation of the resource. The GM waits in its `Resizing` state until it gets enough success responses from its RM instances, i.e., until `GoalConfig.Count == CreatedCount`.

If a resource ever goes unhealthy, the corresponding RM instance transitions to the `Deleting` state and asks the resource provider to de-allocate the resource. On successful deallocation, the RM transitions to the `Deleted` state, and informs the GM, upon which the GM will issue the `ScaleUp` operation to allocate a new resource. Pool deletion is similar and implemented via a corresponding `ScaleDown` operation. Both RMs and GMs halt themselves after transitioning to the `Deleted` state.

Correctness. We use the P#-testing backend to check the conformance of `RsmRgs` to the following specifications. The testing helped weed out several bugs while implementing the RSM program. These properties were tested against a model of the resource provider where the allocation of a resource can non-deterministically fail (but eventually allocation is successful on repeated attempts) and the resource can go unhealthy at any time.

▶ **Property 5.1.** *Immediately following a* `ScaleUp` *or* `ScaleDown` *operation, the number of resources under creation, or already created, equals the desired number of resources.*

▶ **Property 5.2.** *If a client issues the sequence of requests* `CreateRG`(n_1)*,* `ResizeRG`(n_2)*, . . . ,* `ResizeRG`(n_k)*, then* `RsmRgs` *will eventually create a group with exactly* n_k *resources.*

▶ **Property 5.3.** *On issuing a* `DeleteRG`*, eventually all resources of the group are disposed.*

A comparison of `RsmRgs` with `ProdRgs`. The resource and group managers lend themselves naturally to a state machine encoding. The state machines manage the life-cycle of a resource or a group, respectively. `ProdRgs` had a similar design, however, communication was not through message passing but rather via shared tables, maintained as SF reliable collections. One agent would update a table and other agents would continuously poll these tables to get the updates. Polling increased CPU utilization: `RsmRgs` uses roughly 10× less CPU than `ProdRgs`. Implicit communication also made the code harder to reason for correctness.

A direct comparison between the code size of `ProdRgs` and `RsmRgs` is not possible because the former implements more features. However, `RsmRgs` implements all of the core functionality in approximately 2000 lines of code, several times smaller than the corresponding functionality in `ProdRgs`. The designers of `ProdRgs` attest to the benefits listed here.

To contain code complexity, `ProdRgs` was not designed to be responsive during resize operations: it would wait to finish one resize operation before looking at subsequent resize requests. `RsmRgs`, on the other hand, is fully responsive in such scenarios. The GM state machine can handle new resize requests while it is in the `Resize` state: it simply updates its `GoalConfig` and issues either `ScaleUp` and `ScaleDown` until the group reaches its goal state. Importantly, the `P#`-based testing infrastructure of RSMs provides strong confidence in exploring a more responsive (and more complex) state-machine design.

6 Evaluation

This section reports on a performance evaluation of our RSM implementation. Section 6.1 measures common performance metrics on micro-benchmarks. Section 6.2 evaluates the performance of our ResourceGroupServer implementation. We draw comparisons with the Reliable Actors programming model of Service Fabric [35] (denoted `sfActor`). Reliable actors are an implementation of the "virtual actors" paradigm [12]. It serves as a useful baseline for experimentation because it builds on SF much like our SF backend implementation. Further, reliable actors do not provide failure transparency guarantees, although the programmer is given access to a persistent key-value store. This allows us to measure the relative overheads with providing a by-construction fault-tolerant runtime. In the rest of this section, we use the generic term *agents* to denote both `sfActor`s and RSMs.

6.1 Microbenchmarks

Our microbenchmarks evaluate the following three scenarios: (*i*) *creation*: where we measure the creation time for agents, (*ii*) *messaging latency* between two agents and (*iii*) *processing throughput*, where we measure the time taken to process a sequence of messages by an agent. In the subsequent discussion, we use `sfRSM` and `bRSM` to denote the SF-based RSM implementation, with and without optimizations mentioned in Section 4.2, respectively. We use `kRSM` to denote the Kafka-based RSM implementation.

Cluster Setup. The `sfActor`, `bRSM` and `sfRSM` services were deployed on a 5-node Service Fabric cluster on Microsoft Azure, where each node had a `D4_v2` configuration (8 CPU cores, 28GB RAM, and a 400GB local solid-state drive). The Kafka experiments were run on an

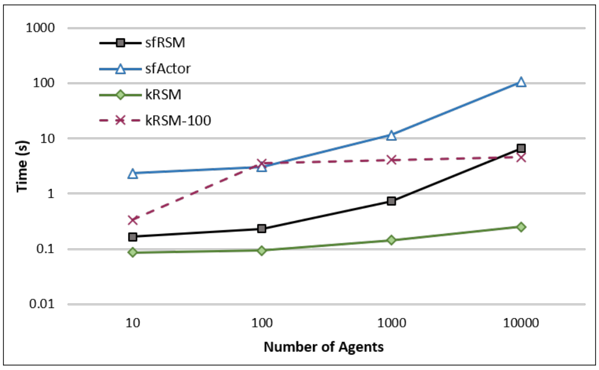


Figure 9 `sfActor` and `sfRSM` creation times.

Table 1 Messaging latencies.

Framework	0.5	0.9	0.99	Mean
sfActor	4.5	8	9.8	4.5
sfActor-Persist	12.5	23	23.5	11.9
sfRSM	23	31.5	70.6	22.8
kRSM	8.8	10	13.5	9.1

Azure HDInsight cluster with the following configuration: (*i*) 2 *head* nodes of type `D4_v2` executing the RSM runtime and application (*ii*) 3 *worker* nodes hosting the Kafka topics, with a total of 24 cores and 84GB RAM, and a total of 6 premium disks of size 1TB each (*iii*) 3 nodes for running Apache Zookeeper, with a total of 12 cores and 21GB RAM. (Zookeeper serves as a coordinator for Kafka nodes and manages cluster metadata.) Because of the different cluster setup, `sfRSM` and `kRSM` are not directly comparable.

Creation. It is important to keep overheads with creation low in order to provide most flexibility in programming RSM applications. In this experiment, we measure the time taken by a client to sequentially create n agents. Both the client and the created agents reside on the same partition, which allows us to eliminate any networking overheads from the creation times. Fig. 9 summarizes the results. The average creation time for `sfRSM` is 5.1ms, nearly 14X faster than `sfActor` (71.4ms), whereas the average creation time of `bRSM` (22.2ms) is 4.4X that of `sfRSM`. The speedup in creation time for `sfRSM` primarily stems from the shared inbox-outbox optimization.

The creation times for both `sfActor` and `sfRSM` scale linearly with the number n of agents created. For `sfRSM`, the bulk of the creation time is expended in committing a single SF transaction, which persists the initial local state of the machine and its `rsmId` to the runtime. Creations in `kRSM` are measured differently. We create the Kafka topics ahead of time because (*i*) creating topics on-the-fly is much slower than pre-creating them in bulk, and more importantly (*ii*) there is a limit to the number of topics that can be supported on each worker node. A `kRSM` creation now simply involves assigning two existing topics from the pool, along with persisting the id and initial state. We run two experiments, `kRSM`-1 and `kRSM`-100, where we multiplex the n RSMs onto a single topic and 100 topics, respectively. Note that all the writes during creation for `kRSM`-1 are batched into a single transaction, while the writes for `kRSM`-100 involve 100 transactions. As Fig. 9 shows, both `kRSM`-1 and `kRSM`-100 creations are fairly lightweight, with average creation times of 2.4ms and 2.6ms respectively (but discounting the topic pre-creation time).

In a separate experiment, we measured the creation throughput, by firing 1000 creation requests in parallel. `sfActor` and `bRSM` could achieve a maximum throughput of 287 and 67 creations per second, respectively, while `sfRSM` could hit a maximum of 1189 creations per second. The faster creations for `sfRSM` stems from its optimizations, which result in frugal CPU and IO requirements. `kRSM` creation throughput was 6661 creations per second.

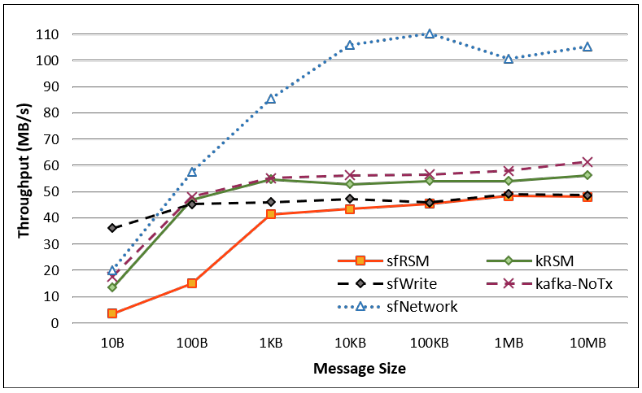


Figure 10 Throughput measurements with RSMs, SF and Kafka.

Messaging. This experiment measures the cost of exact-once messaging. The experiment comprises of two agents that repeatedly send a single message (50 bytes) back-and-forth and we measure messaging latencies. Messaging in `sfActor` is unreliable (best-effort, and lost on failures). We optionally make the agents in `sfActor` persist their incoming message.

Table 1 shows the latency measurements at different quantiles. Unsurprisingly, `sfActor` exhibits the lowest latencies. When we persist the messages in `sfActor`, which introduces one write per message transfer, it increases the latency significantly. `sfRSM` requires two write operations per message, making it nearly twice as expensive as `sfActor` with message persistence. Kafka, being a messaging system, is optimized for low-latency operations, even with exact-once guarantees. `kRSM` has better latency than `sfActor` with message persistence.

Throughput. In this experiment, a producer and a consumer agent are located on different partitions. The producer keeps sending messages (with a varying payload size) to the consumer. The consumer simply keeps a running count of the number of bytes received. We measure the time taken to process all the requests, and report the throughput in MB/s.

Fig. 10 summarizes the results. `sfRSM` automatically batches messages to increase throughput. `sfActor` has no default batching mechanism, although increasing message sizes decreases benefits to be gained from batching. At large message sizes, `sfActor` was able to achieve a maximum throughput of 86.6MB/s. In comparison, the maximum throughput for `sfRSM` (across all message sizes) was 48.4MB/s. To account for this difference, we precisely timed all micro-operations involved in the `sfRSM` runtime.

Sending a message from the producer to the consumer involves writing to the outbox and then sending the message over the network. We separately measured the best throughput of writing to an SF reliable collection (`sfWrite`) and sending data over the network as fast as possible via (unreliable) RPC (`sfNetwork`). Clearly, the throughput of `sfRSM` will be bounded by the smaller of these two values. As Fig. 10 shows, the writes constitute the limiting factor, and `sfRSM` incurs very little overhead over the `sfWrite` throughput, especially for large message sizes. Smaller message sizes implies a larger number of messages per batch, which increases the serialization overhead, and the number of times the consumer executes its handler. This effect, consequently, widens the gap between the `sfRSM` and `sfWrite` throughputs for smaller message sizes. This result shows that any improvements in the write throughput of reliable collections will directly speed up RSMs. The gap between `sfRSM` and `sfNetwork` is the cost of reliable messaging. Nonetheless, even at the small message size of 100bytes, `sfRSM` are able to do roughly 150K message transfers per second; enough for many

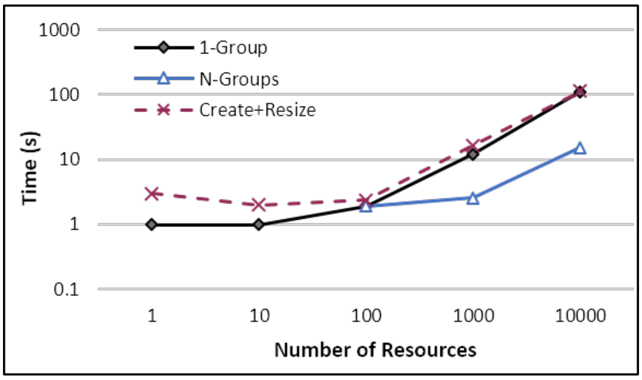


Figure 11 `RsmRgs` resource creation.

realistic applications. `kRSM` throughput peaks at 56.2MB/s. With Kafka, the persistence and message transfer happen together as a topic write. The upper bound for `kRSM` is to use non-transactional writes (`kafka-NoTx`). Fig. 10 shows that `kRSM` have little overhead compared to the throughput of `kafka-NoTx`.

6.2 `RsmRgs` Case Study

Performance. We measure the time taken to create a given number of resources in a single partition, assuming that the resource provider calls are instantaneous. Fig. 11 summarizes the results.

In the first experiment, denoted as 1-Group, we create a *single* group with progressively increasing number of resources. The more realistic scenario, which arises in production, is to have *multiple* groups of small sizes in a single partition. In the N-Groups experiment, we create multiple groups (each of size 100) in parallel such that the total number of resources matches the X-axis. We make two observations: (i) the creation times for both 1-Group and N-Groups increase linearly with the number of resources (ii) for the same number of resources, the increased parallelism in N-Groups results in the creation times being an order of magnitude faster than 1-Group. We would like to emphasize that the workloads here are realistic, and are based on requirements provided by the developers of the in-production `ProdRgs` service. The aforementioned results were reviewed by the developers, who confirmed that `RsmRgs` comfortably scales to production workloads.

To evaluate the responsiveness of `RsmRgs`, we issue `CreateRG`(y), followed immediately by `ResizeRG`(x), where $x = y/100$. The requirement is to ensure that the total time stays close to `CreateRG`(x). The Create+Resize line in Fig. 11 summarizes the result (with the value x on the X-axis). We see that as we increase x, the Create+Resize curve lies very close to 1-Group, which is testament to the service's responsiveness. For small x, the gap is wider because almost all of the y allocations kick-in by the time the resize request is processed.

Testing. For testing, we create mocks of both the client and the Resource Provider services, since they are external to `RsmRgs`. Our mocks are vanilla `P#` machines. The testing exercise was done on a laptop with a dual-core i7 processor, with 8GB RAM. The tester performed 100 iterations, with a scheduling strategy chosen from a predefined portfolio, with each exploration having a depth of $10,000$ steps. Note that the test for Property 5.1 is a safety-check, while the tests for Properties 5.2 and 5.3 are liveness-checks. The client issued a `CreateRG`(100) request. We deliberately injected a bug in the `ScaleUp` operation by removing the updates to `CreatingCount`. The resulting violation of Property 5.1 was detecting in 0.75s, generating an error witness of around 64 steps. We fixed the error, and issued `CreateRG`(100), followed by `ResizeRG`(5), and Property 5.2 was verified in 147.9s. To verify Property 5.3, we issue `CreateRG`(50) followed by a deletion and the tester verified the property in 119.1s. We further injected a bug by converting the `CreatedCount` to be volatile. (This means that if the machine was in the middle of a creation operation when it failed, it would lose track of all the resources it had created, and therefore the group would never reach the `Created` state.) The tester is able to quickly find a violation of property 5.2, in 5.5s.

Other applications. We have evaluated the applicability of the RSM language and runtime by encoding several other real-world applications. One example is a Banking application, comprising *account* and *broker* RSMs, with the latter being tasked with transferring money from one account to the other, without incurring any financial losses on failures. This specification can be encoded as a liveness property. Another example is a Survey application [32, 37], where *subscribers* can create surveys, which users can respond to. Each survey is managed by an RSM, and an overall coordinator RSM creates surveys, reports survey status, deletes surveys, etc. From a user perspective, responsiveness is a key metric. The application also needs to ensure specifications like a user vote is counted exactly once. The RSM framework allowed us to design these responsive applications, with all the specifications thoroughly tested.

7 Related Work

Actor frameworks. In actor-based programming [24], a natural fit for cloud services, an application comprises concurrent entities (called *actors*), each maintaining its local state, which is not shared among other actors. Communication and co-ordination between actors happens via message passing. Some popular instances of actor-based frameworks and languages include Akka [1], Erlang [20], and Orleans [12]. Fault-tolerance in these frameworks is achieved by checkpointing state to a persistent store, which is automatically restored upon actor rehydration. The responsibilities of checkpointing the state, ensuring consistency with the rest of the system, managing messaging retries and de-duplication, all rests with the programmer. More specifically, unlike RSMs, these frameworks do not provide failure transparency by-construction. Orleans introduced the concept of "virtual actors": these actors need not be explicitly created. They are instantiated on demand when they receive a message. Further, they are location independent, allowing the Orleans runtime to dynamically load-balance the placement of actors across a cluster, even putting frequently-communicating actors together [30]. RSM instances must be explicitly created, but they are location independent. Our implementation, however, currently does not attempt to move an RSM after it has been created. The initial placement of a fresh RSM can be controlled by the programmer, after which the instance is permanently tied to that location. Service Fabric Reliable Actors [35] are also an implementation of the virtual actors paradigm. We provide an empirical comparison of RSMs with Reliable Actors in Section 6.

Reactive programming. Reactive frameworks [10] are used in the development of event-driven and interactive applications. These frameworks provide a programmatic way of setting up a *dataflow graph* that marks functional dependencies between variables. As the value of certain variables change over time, the rest of the dependent variables are updated automatically. Recent work [27] describes an extension to REScala [34] in order to provide fault-tolerance support in distributed reactive programming. The framework relies on taking snapshots of critical data and then uses replay to construct the entire program state on failure. This requires deterministic execution. Further, the input signals are not captured as part of the snapshots, causing them to differ on re-execution or even get duplicated. These issues require programmer support. On the other hand, RSMs can support non-deterministic handlers and guarantees exact-once processing because input (i.e. inbox) is part of the reliable state that RSMs maintain. The REScala extension provides eventual consistency for updates to shared data, making use of state-based conflict-free replicated data types (CRDTs) [36]. RSMs do not have shared state; maintaining common state between two RSMs can be done by creating (and communicating with) another RSM that owns the state. RSM messaging is reliable: this provides strong consistency between RSMs, however, it is less resilient to network outages than CRDTs because the latter allows for progress even in a disconnected state.

Big-data analytics. Big-data processing systems such as SPARK [38] and SCOPE [13] are popular frameworks for data analytics. They provide a SQL-like programming interface that gets compiled to map-reduce stages for distributed execution on a fault-tolerant runtime. These systems, however, are meant for data-parallel batch processing. They execute on immutable input that is known ahead of time.

Other frameworks. Ramalingam et al. [32] provide a monadic framework that makes functional computation idempotent. Their transformation records the sequence of steps that have already been executed. On re-execution, such steps are skipped. Idempotent computation enables fault-tolerance: simply keep re-executing until completion. Their work focuses on state updates made by a single sequential agent. They assume determinism of the computation and do not handle communication. RSM programs, on the other hand, support multiple concurrent agents with possible non-deterministic execution. RSMs ensure idempotence by atomically committing the effects of processing of a message along with the dequeue of the message from the inbox. Another class of languages for distributed systems, including Orca [11] and X10 [14], rely on distributed shared memory. They enable applications that span multiple machines while allowing the freedom to access memory across machine boundaries. They mostly focus on in-memory computation, without support for state persistence or fault tolerance.

The setting of Replicated State Machines [26] concerns a single *deterministic* state machine, replicated for fault tolerance. All replicas agree on a global ordering of submitted operations. This is the foundational concept in the domain of distributed consensus. In contrast, RSMs are at a higher level of abstraction, allowing a programmer to string together concurrent *interacting* state machines to encode fail-free business logic. RSMs delegate consensus to the storage layer.

References

1 Akka. `https://akka.io/`. [Online; accessed 10-January-2019].

2 Apache Kafka. `https://kafka.apache.org/`. [Online; accessed 1-January-2019].

3 Asynchronous programming with async and await in C#. `https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/async/`.

4 Azure Service Fabric. `https://azure.microsoft.com/services/service-fabric/`.

5 Azure Service Fabric Communication. `https://docs.microsoft.com/en-us/azure/service-fabric/service-fabric-reliable-services-communication-remoting`.

6 Azure Service Fabric Partitioning. https://docs.microsoft.com/en-us/azure/service-fabric/service-fabric-concepts-partitioning.

7 Azure Service Fabric Reliable Collections. https://docs.microsoft.com/en-us/azure/service-fabric/service-fabric-reliable-services-reliable-collections.

8 Azure Service Fabric Reliable Services. https://docs.microsoft.com/en-us/azure/service-fabric/service-fabric-reliable-services-introduction.

9 Azure Service Fabric Reliable State Manager. https://docs.microsoft.com/en-us/dotnet/api/microsoft.servicefabric.data.ireliablestatemanager?view=azure-dotnet.

10 Engineer Bainomugisha, Andoni Lombide Carreton, Tom Van Cutsem, Stijn Mostinckx, and Wolfgang De Meuter. A survey on reactive programming. *ACM Comput. Surv.*, 45(4):52:1–52:34, 2013. doi:10.1145/2501654.2501666.

11 Henri E. Bal, M. Frans Kaashoek, and Andrew S. Tanenbaum. Orca: A Language For Parallel Programming of Distributed Systems. *IEEE Trans. Software Eng.*, 18(3):190–205, 1992. doi:10.1109/32.126768.

12 Philip A Bernstein, Sergey Bykov, Alan Geller, Gabriel Kliot, and Jorgen Thelin. Orleans: Distributed virtual actors for programmability and scalability. *MSR-TR-2014–41*, 2014.

13 Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. Apollo: Scalable and Coordinated Scheduling for Cloud-Scale Computing. In Jason Flinn and Hank Levy, editors, *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014.*, pages 285–300. USENIX Association, 2014. URL: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/boutin.

14 Philippe Charles, Christian Grothoff, Vijay A. Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In Ralph E. Johnson and Richard P. Gabriel, editors, *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16-20, 2005, San Diego, CA, USA*, pages 519–538. ACM, 2005. doi:10.1145/1094811.1094852.

15 Pantazis Deligiannis, Alastair F. Donaldson, Jeroen Ketema, Akash Lal, and Paul Thomson. Asynchronous programming, analysis and testing with state machines. In David Grove and Steve Blackburn, editors, *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 154–164. ACM, 2015. doi:10.1145/2737924.2737996.

16 Pantazis Deligiannis, Matt McCutchen, Paul Thomson, Shuo Chen, Alastair F. Donaldson, John Erickson, Cheng Huang, Akash Lal, Rashmi Mudduluru, Shaz Qadeer, and Wolfram Schulte. Uncovering Bugs in Distributed Storage Systems during Testing (Not in Production!). In Angela Demke Brown and Florentina I. Popovici, editors, *14th USENIX Conference on File and Storage Technologies, FAST 2016, Santa Clara, CA, USA, February 22-25, 2016.*, pages 249–262. USENIX Association, 2016. URL: https://www.usenix.org/conference/fast16/technical-sessions/presentation/deligiannis.

17 Ankush Desai, Vivek Gupta, Ethan K. Jackson, Shaz Qadeer, Sriram K. Rajamani, and Damien Zufferey. P: safe asynchronous event-driven programming. In Hans-Juergen Boehm and Cormac Flanagan, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 321–332. ACM, 2013. doi:10.1145/2491956.2462184.

18 Ankush Desai, Shaz Qadeer, and Sanjit A. Seshia. Systematic testing of asynchronous reactive systems. In Elisabetta Di Nitto, Mark Harman, and Patrick Heymans, editors, *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, pages 73–83. ACM, 2015. doi:10.1145/2786805.2786861.

19 Enterprise workloads in the cloud. https://www.forbes.com/sites/louiscolumbus/2018/01/07/83-of-enterprise-workloads-will-be-in-the-cloud-by-2020/#636ee7856261.

20 Erlang. https://www.erlang.org/. [Online; accessed 10-January-2019].

21 Jim Gray. The Transaction Concept: Virtues and Limitations (Invited Paper). In *Very Large Data Bases, 7th International Conference, September 9-11, 1981, Cannes, France, Proceedings*, pages 144–154. IEEE Computer Society, 1981.

22 Maurice Herlihy and Jeannette M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990. doi:10.1145/78969.78972.

23 Kafka Powered By. https://kafka.apache.org/powered-by. [Online; accessed 1-January-2019].

24 Rajesh K. Karmani and Gul Agha. Actors. In David A. Padua, editor, *Encyclopedia of Parallel Computing*, pages 1–11. Springer, 2011. doi:10.1007/978-0-387-09766-4_125.

25 Jay Kreps, Neha Narkhede, and Jun Rao. Kafka: a Distributed Messaging System for Log Processing. In *6th International Workshop on Networking Meets Databases (NetDB)*, 2011.

26 Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, 21(7):558–565, 1978. doi:10.1145/359545.359563.

27 Ragnar Mogk, Lars Baumgärtner, Guido Salvaneschi, Bernd Freisleben, and Mira Mezini. Fault-tolerant Distributed Reactive Programming. In Todd D. Millstein, editor, *32nd European Conference on Object-Oriented Programming, ECOOP 2018, July 16-21, 2018, Amsterdam, The Netherlands*, volume 109 of *LIPIcs*, pages 1:1–1:26. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018. doi:10.4230/LIPIcs.ECOOP.2018.1.

28 Rashmi Mudduluru, Pantazis Deligiannis, Ankush Desai, Akash Lal, and Shaz Qadeer. Lasso detection using partial-state caching. In Daryl Stewart and Georg Weissenbacher, editors, *2017 Formal Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, October 2-6, 2017*, pages 84–91. IEEE, 2017. doi:10.23919/FMCAD.2017.8102245.

29 Suvam Mukherjee, Nitin John Raj, Krishnan Govindraj, Pantazis Deligiannis, Chandramouleswaran Ravichandran, Akash Lal, Aseem Rastogi, and Raja Krishnaswamy. Reliable State Machines: A Framework for Programming Reliable Cloud Services. *CoRR*, abs/1902.09502, 2019. arXiv:1902.09502.

30 Andrew Newell, Gabriel Kliot, Ishai Menache, Aditya Gopalan, Soramichi Akiyama, and Mark Silberstein. Optimizing distributed actor systems for dynamic interactive services. In *Proceedings of the Eleventh European Conference on Computer Systems, EuroSys 2016, London, United Kingdom, April 18-21, 2016*, pages 38:1–38:15, 2016. doi:10.1145/2901318.2901343.

31 P#. P#: Safe Asynchronous Event-Driven Programming. https://github.com/p-org/PSharp. [Online; accessed 1-January-2019].

32 Ganesan Ramalingam and Kapil Vaswani. Fault tolerance via idempotence. In Roberto Giacobazzi and Radhia Cousot, editors, *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, pages 249–262. ACM, 2013. doi:10.1145/2429069.2429100.

33 Amr Sabry and Matthias Felleisen. Reasoning about Programs in Continuation-Passing Style. *Lisp and Symbolic Computation*, 6(3-4):289–360, 1993.

34 Guido Salvaneschi, Gerold Hintz, and Mira Mezini. REScala: bridging between object-oriented and functional style in reactive applications. In Walter Binder, Erik Ernst, Achille Peternier, and Robert Hirschfeld, editors, *13th International Conference on Modularity, MODULARITY '14, Lugano, Switzerland, April 22-26, 2014*, pages 25–36. ACM, 2014. doi:10.1145/2577080.2577083.

35 Service Fabric Reliable Actors. https://docs.microsoft.com/en-us/azure/service-fabric/service-fabric-reliable-actors-introduction.

36 Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. *JHAL-Inria*, page 50, 2011. URL: https://hal.inria.fr/inria-00555588.

37 The TailSpin Scenario. https://docs.microsoft.com/en-us/azure/architecture/multitenant-identity/tailspin. Accessed: 2019-1-10.

38 Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In Steven D. Gribble and Dina Katabi, editors, *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012*, pages 15–28. USENIX Association, 2012. URL: https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia.

Transferring Obligations Through Synchronizations

Jafar Hamin
imec-DistriNet, Department of Computer Science, KU Leuven, Belgium
jafar.hamin@cs.kuleuven.be

Bart Jacobs
imec-DistriNet, Department of Computer Science, KU Leuven, Belgium
bart.jacobs@cs.kuleuven.be

Abstract

One common approach for verifying safety properties of multithreaded programs is assigning appropriate permissions, such as ownership of a heap location, and obligations, such as an obligation to send a message on a channel, to each thread and making sure that each thread only performs the actions for which it has permissions and it also fulfills all of its obligations before it terminates. Although permissions can be transferred through synchronizations from a sender thread, where for example a message is sent or a condition variable is notified, to a receiver thread, where that message or that notification is received, in existing approaches obligations can only be transferred when a thread is forked. In this paper we introduce two mechanisms, one for channels and the other for condition variables, that allow obligations, along with permissions, to be transferred from the sender to the receiver, while ensuring that there is no state where the transferred obligations are lost, i.e. where they are discharged from the sender thread but not loaded onto the receiver thread yet. We show how these mechanisms can be used to modularly verify deadlock-freedom of a number of interesting programs, such as some variations of *client-server* programs, *fair readers-writers locks*, and *dining philosophers*, which cannot be modularly verified without such transfer. We also encoded the proposed separation logic-based proof rules in the VeriFast program verifier and succeeded in verifying the mentioned programs.

2012 ACM Subject Classification Theory of computation → Separation logic; Software and its engineering → Deadlocks; Software and its engineering → Process synchronization; Software and its engineering → Formal software verification; Theory of computation → Hoare logic

Keywords and phrases Hoare logic, separation logic, modular program verification, synchronization, transferring obligations, deadlock-freedom

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2019.19

Funding This research received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731453 (project VESSEDIA), as well as Flemish Research Fund project grant G.0962.17N.

Acknowledgements We thank three anonymous reviewers for their careful reading of our manuscript and their many insighful comments and suggestions, and also Amin Timany for his guidance on Coq.

1 Introduction

One common approach for verifying safety properties of multithreaded programs, such as absence of data races and deadlock, is assigning appropriate permissions [3, 37] and obligations [29, 33, 1] to each thread and making sure that each thread only performs the actions for which it has permissions and it also fulfills all of its obligations before it terminates. In a separation logic-based approach [42], for example, the ownership of a heap location is a permission for accessing that location, which is assigned to the thread allocating that location. Since there is only one instance of such permission, only one thread, the one having

33rd European Conference on Object-Oriented Programming (ECOOP 2019).
Editor: Alastair F. Donaldson; Article No. 19; pp. 19:1–19:58
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

such permission, can access that location, which ensures absence of data races. Absence of deadlock, as another example, can be verified by making sure that for any waitable object *o*, such as a lock, a channel, or a condition variable, for which a thread is waiting there is a thread obliged to fulfill an obligation for that object (which only waits for objects whose levels, some arbitrary numbers[1] associated with waitable objects, are lower than the level of *o*) [33, 2, 24, 15]. In this setting a thread can discharge an obligation for a lock, a channel, or a condition variable by releasing that lock, sending a message on that channel, or notifying that condition variable (under some conditions), respectively.

Permissions can be transferred through synchronizations [33, 34] by consuming them in the sender thread, where for example a message is sent or a condition variable is notified, and producing them in the receiver thread, where that message or that condition variable is received. However, this technique cannot be used for transferring obligations because the transferred obligations are lost if no thread receives them. Additionally, even in the presence of a receiver there might be a state where these obligations are discharged from the sender but not received by the receiver yet. If the obligations are transferred through channels, for example, if the receive is not scheduled to be executed *immediately* after the send then there would be a state where no thread holds the transferred obligations, which makes the approach unsound. Note that in the case that the send operation is synchronous [29], i.e. the sender thread is suspended until a thread receives the sent message, sending obligations through channels is perfectly fine because these obligation are never lost.

In this paper we introduce two mechanisms allowing threads to transfer their obligations through synchronizations while ensuring that there is no state where these obligations are lost, i.e. where they are discharged from the sender thread but not loaded onto the receiver thread yet. The main idea behind the first mechanism is that when the transferred obligations are discharged from the sender thread the levels of these obligations have been already loaded onto a (receiver) thread. These levels are discharged from the receiver thread when it receives the transferred obligations. In the second mechanism, which is specifically for condition variables, the obligations are discharged from a notifying thread only if there is a waiting thread which is notified and receives these obligations. We show that using these mechanisms in some modular approaches, verifying finite blocking [2] and deadlock-freedom of channels and locks [33, 24], semaphores [21], and monitors [15], enables them to verify a wider range of interesting programs, such as some variations of *client-server* programs, *fair mutexes*, *fair readers-writers locks*, and *dining philosophers.* We encoded the proposed proof rules in the VeriFast program verifier [25, 26, 22] and succeeded in verifying the programs above[2]. Additionally, we proved the soundness of both mechanisms (see Appendixes C and D): the soundness proof of the second mechanism is machine-checked with Coq[3].

In the rest of this paper Sections 2 and 3 introduce two mechanisms for transferring obligations through channels and notifications, Section 4 discusses related work, and Section 5 draws a conclusion.

[1] In this paper, for simplicity we use numbers as levels, but any partially ordered set can be used as the set of levels.

[2] The proof of these programs, verified by the VeriFast program verifier, can be found in [17].

[3] The soundness proof of the second mechanism, machine-checked in Coq, can be found in [17].

2 Transferring Obligations Through Channels

In this section we first review an approach, introduced by Leino et al. [33], which modularly verifies deadlock-freedom of channels. Then we extend this approach such that it also allows obligations to be transferred from the thread sending on a channel to the thread receiving from that channel. Lastly, we provide some variations of a client-server program which can be verified thanks to such extension.

2.1 Verifying Channels

Leino et al. [33] introduced a modular approach to verify deadlock-freedom of programs which communicate through channels. The main idea in this approach is to associate with each thread a bag[4] of channels, namely the bag of *obligations* of that thread, which must be empty when that thread terminates. A thread can discharge an obligation for a channel by either sending a message on that channel or delegating that obligation to a forked thread. This approach guarantees absence of deadlock by making sure that for any thread trying to receive a message from a channel *ch* there is either a message in *ch* or an obligation for *ch* in the bag of obligations of a thread which only waits for objects whose levels are lower than the level of *ch* (preventing circular dependencies). This constraint is applied by making sure that if a thread tries to receive from a channel *ch* then 1) it spends a *credit* for *ch*, where a thread can obtain a credit for *ch* by adding *ch* to the bag of its obligations, and 2) the level of *ch* is lower than the levels of the obligations of that thread. Note that a credit for a channel in this approach indicates that there is either a message on that channel or there is a thread holding an obligation for that channel.

As an example, consider the deadlock-free program shown in the left hand side of Figure 1, where after creating a channel *ch*, the main thread first forks a receiver thread, trying to receive a message from *ch*, and then sends a message on *ch*. The verification of this program is shown in the right hand side of this figure, where we use separation logic [42] to reason about the ownership of permissions. The verification of this program is started with an empty bag of obligations, denoted by $\mathsf{obs}(\{\!\!\}\!\!\})$. As indicated below each command, by creating a channel a (*duplicable* and *leakable*) permission $\mathsf{channel}$ for accessing that channel is produced and an arbitrary level, denoted by a function R, is assigned to that channel. Before forking the receiver thread, using a *ghost command*[5] $\mathsf{g\_credit}$, a credit and an obligation for this channel are loaded onto the main thread. The former is given to the forked thread, where this credit is spent by the command $\mathsf{receive}(ch)$, and the latter is discharged by the main thread when it executes the command $\mathsf{send}(ch, 12)$.

The separation logic-based proof rules, introduced by Jacobs et al. [23, 24], used to verify this program are shown in Figure 2. Note that the postcondition of each command in this figure is a lambda expression that given the return value of that command returns the assertion held after execution of that command. When a channel is created, as shown in Rule NewChannel, any arbitrary level can be assigned to that channel by the proof author. Note that, generally, this level must be chosen such that the constraint number 2, mentioned above, is met at each receive operation. Sending a message on a channel, as shown in Rule Send, discharges an obligation for that channel. As shown in Rule Receive, a thread can

[4] We model bags of objects as functions from objects to natural numbers. We also use ⊎ indicating the union (i.e. the pointwise sum) of two bags.

[5] The ghost commands are inserted into the program for verification purposes and have no effect on the program's behavior.

```
ch := new_channel();
fork(receive(ch));
send(ch, 12)
```

```
{obs({})}
ch := new_channel();
{obs({}) ∗ channel(ch) ∧ R(ch)=r}
g_credit(ch);
{obs({ch}) ∗ channel(ch) ∗ credit(ch)}
fork(
  {obs({}) ∗ channel(ch) ∗ credit(ch) ∧ ch ≺ {}}
  receive(ch)
  {obs({})}
);
{obs({ch}) ∗ channel(ch)}
send(ch, 12)
{obs({})}
```

Figure 1 Verification of deadlock-freedom of channels, where O in $\mathsf{obs}(O)$ is the bag of obligations of the running thread, $\mathsf{R}(ch)$ denotes the level of ch, and $ch \prec O \Leftrightarrow \forall o{\in}O.\ \mathsf{R}(ch) < \mathsf{R}(o)$.

try to receive a message from a channel ch only if that thread spends a credit for ch and the level of ch is lower than the levels of all obligations of that thread. As shown in Rule Fork, a thread can transfer a part of its permissions and obligations to a forked thread, provided that the forked thread discharges all the delegated obligations. Lastly, as shown in Rule Credit, using a ghost command g_credit a thread can obtain a credit for a channel if that channel is loaded onto the bag of the obligations of that thread.

It can be proved that any program verified by the mentioned proof rules, where the verification starts from an empty bag of obligations and also ends with such bag, never deadlocks, i.e. it always has a running thread, not waiting for any channel, until it terminates. We know that for any channel ch all of these proof rules preserve the following invariant, where $Ct(ch)$ and $Ot(ch)$ denote the total number of credits and obligations for ch in the system, respectively, and $\mathsf{size}(ch)$ denotes the number of messages in ch:

$$Ct(ch) \leqslant Ot(ch) + \mathsf{size}(ch) \tag{1}$$

Now consider a deadlocked state, where each thread of a verified program is waiting for a channel. Among all of these channels take the one having a minimal wait level, namely ch_{min}. Since $\mathsf{size}(ch_{min})=0$ and there exists a credit for ch_{min} in the system held by the waiting thread, according to the invariant above and the constraint number 2, there exists a thread having an obligation for ch_{min} that is waiting for a channel whose level is lower than the level of ch_{min}, which contradicts minimality of the level of ch_{min}.

2.2 Transferring Permissions and Obligations Through Channels

The approach presented in the previous section fails to verify some applications of channels where some obligations must be transferred from a thread sending on a channel to the thread receiving from that channel. Consider the *client-server* program shown in Figure 3, for example, where the server waits to receive a message that consists of a client request, which must be processed by the server, and a client channel, from which the client expects to receive the response of the server. Although the client routine in this example is deadlock-free, the Leino et al. approach fails to verify this program, since the routine main cannot give any credit for the channel ch', which is created inside the client, to the client. This program can be verified if after creating ch' a credit and an obligation for ch' are loaded onto the client such that the latter is transferred to the server through the client's request and the former is spent for the command $\mathsf{receive}(ch')$ executed in the client.

NewChannel

$$\{\mathsf{true}\}\ \mathsf{new\_channel}\ \{\lambda ch.\ \mathsf{channel}(ch) \wedge \mathsf{R}(ch){=}r\}$$

Send

$$\{\mathsf{obs}(O) * \mathsf{channel}(ch)\}\ \mathsf{send}(ch, m)\ \{\lambda\_.\ \mathsf{obs}(O - \{\!\![ch]\!\!\}) * \mathsf{channel}(ch)\}$$

Receive

$$\{\mathsf{obs}(O) * \mathsf{channel}(ch) * \mathsf{credit}(ch) \wedge ch \prec O\}\ \mathsf{receive}(ch)\ \{\lambda\_.\ \mathsf{obs}(O) * \mathsf{channel}(ch)\}$$

Fork

$$\frac{\{a * \mathsf{obs}(O)\}\ c\ \{\lambda\_.\ \mathsf{obs}(\{\!\![]\!\!\})\}}{\{a * \mathsf{obs}(O \uplus O')\}\ \mathsf{fork}(c)\ \{\lambda\_.\ \mathsf{obs}(O')\}}$$

Credit

$$\{\mathsf{obs}(O) * \mathsf{channel}(ch)\}\ \mathsf{g\_credit}(ch)\ \{\lambda\_.\ \mathsf{obs}(O \uplus \{\!\![ch]\!\!\}) * \mathsf{channel}(ch) * \mathsf{credit}(ch)\}$$

Figure 2 Proof rules ensuring deadlock-freedom of channels.

```
routine server(channel ch){          routine client(channel ch){       routine main(){
 (req, ch') := receive(ch);           ch' := new_channel();             ch := new_channel();
 result := process(req);              send(ch, (request(), ch'));       fork(server(ch));
 send(ch', result)}                   receive(ch')}                     client(ch)}
```

Figure 3 A client-server program.

Similar to permissions, two necessary conditions for transferring obligations are: 1) when a thread sends a message on a channel ch the transferred obligations of ch, which are transferred through ch, are discharged from the bag of the obligations of that thread, and 2) when a thread receives a message from a channel ch the transferred obligations of ch are loaded onto the bag of the obligations of that thread. However, these conditions are not sufficient because these obligations are lost if no thread receives from ch. Additionally, even in the presence of a receiver if the $\mathsf{receive}(ch)$ is not scheduled to be executed *immediately* after the $\mathsf{send}(ch, m)$ then there would be a state where no thread holds the transferred obligations, which makes the approach unsound.

To address this problem, in addition to the bag of obligations, we associate with each thread a new bag, namely the bag of *importers* of that thread, which consists of the channels that transfer (import) some obligations to that thread. Similar to the bag of obligations, the bag of importers of a thread must be empty when that thread terminates and a thread can wait for a channel ch only if the level of ch is lower than the levels of all obligations which are *possibly* imported by all importers of that thread except for ch itself. Having this bag, we enforce an additional condition, numbered 3, when a thread sends some obligations on an importer channel, which imports some obligations, this channel must be already in the bag of importers of a (receiver) thread. This importer channel is discharged from the receiver thread as it receives a message from this channel. This additional condition is met by making sure that any thread sending on an importer channel ch spends a *transferring credit* for ch, where a thread can obtain a transferring credit of ch by adding ch to the bag of its importers.

Formally, the third condition, which holds for any importer channel ch, ensures that for any transferring credit of ch or any message in ch (which means a transferring credit of ch has been spent by sending a message on ch and no thread has received it yet), there exists an instance of ch in the bag of importers of a (receiver) thread. This invariant is shown in the following as Invariant 2, where $\mathsf{M}^{\mathsf{r}}(ch)$ denotes the bag of the levels of the obligations which are *possibly* imported by ch, $Tt(ch)$ denotes the total number of transferring credits of ch in the system, $It(ch)$ denote the number of occurrences of ch in the bags of importers of all threads in the system, and $\mathsf{size}(ch)$ denotes the number of messages in the queue of ch.

$$\forall ch.\ \mathsf{M}^{\mathsf{r}}(ch){\neq}\{\!\}\ \Rightarrow Tt(ch) + \mathsf{size}(ch) \leqslant It(ch) \tag{2}$$

Additionally, the two first conditions mentioned above, as well as the ones mentioned in the previous section, ensure that if a thread waits on a channel ch' then there is either an obligation of ch' in the system, or a message on ch', or a message in the queue of a channel through which an obligation of ch' is transferred (which means an obligation of ch' has been transferred through this message and no thread has received it yet). This invariant is formally shown in the following as Invariant 3, where $Ot(ch)$ and $Ct(ch)$ denote the total number of obligations and credits of ch in the system, $\mathsf{queue}(ch)$ denotes the list of messages in the channel ch, and $\mathsf{M}'(ch)$ is a function, that given a message, specifies the bag of the obligations which are transferred through that message in ch. Note that the levels of these obligations must be in the bag of the levels of the obligations which are possibly imported by ch, as shown formally in Invariant 4, where $\mathsf{levels}(O)$ maps each element of O to its level.

$$\forall ch'.\ Ct(ch') \leqslant Ot(ch') + \mathsf{size}(ch') + \sum_{ch}\ \sum_{m\in\mathsf{queue}(ch)} \mathsf{M}'(ch)(m)(ch') \tag{3}$$

$$\forall ch, m{\in}\mathsf{queue}(ch).\ \mathsf{levels}(\mathsf{M}'(ch)(m)) \subseteq \mathsf{M}^{\mathsf{r}}(ch) \tag{4}$$

It can be proved that any program preserving such invariants never deadlocks, i.e. it always has a running thread, not waiting for any channel, until it terminates. Consider a deadlocked state, where each thread of a program is waiting for a channel. Among all of these channels take the one having a minimal wait level, namely ch_{min}. Since $\mathsf{size}(ch_{min}){=}0$ and there exists a credit for ch_{min} in the system held by the waiting thread, according to Invariant 3, there exists either 1) a thread having an obligation of ch_{min} that is waiting for a channel whose level is lower than the level of ch_{min}, or 2) there exists a message m on a channel ch through which an obligation of ch_{min} is transferred, i.e. $0{<}\mathsf{M}'(ch)(m)(ch_{min})$ which by Invariant 4 implies $\mathsf{R}(ch_{min}){\in}\mathsf{M}^{\mathsf{r}}(ch)$, where $\mathsf{R}(ch_{min})$ denotes the level of ch_{min}. In the first case minimality of the level of ch_{min} is contradicted. In the second case, since $0{<}\mathsf{size}(ch)$ and $\mathsf{M}^{\mathsf{r}}(ch){\neq}\{\!\}$, by Invariant 2, there exists a thread having an importer channel ch that is waiting for a channel whose level is lower than the level of ch_{min} (because ch imports an obligation of level of ch_{min}), which again contradicts minimality of ch_{min}.

The proof rules enforcing such invariants are shown in Figure 4, where $\mathsf{M}^{\mathsf{r}}(ch)$ denotes the bag of the levels of the obligations which are possibly imported by ch, and the parameters M and M' in the permission $\mathsf{channel}$ of a channel are two functions that given a message return the permissions and the obligations which are transferred through that message. When a channel ch is created, as shown in Rule NewChannel, the value of these functions for this channel can be specified arbitrarily. As shown in Rule Send, when a message m is sent on ch, the permissions which are transferred through m as well as one transferring credit of ch, denoted by $\mathsf{trandit}(ch)$, (if ch is an importer channel) are consumed and the obligations

NewChannel

$$\{\mathsf{true}\}\ \mathsf{new\_channel}\ \{\lambda ch.\ \mathsf{channel}(ch, M, M') \land \mathsf{R}(ch){=}r \land \mathsf{M}^{\mathsf{r}}(ch){=}R\}$$

Send

$$\begin{array}{l}\{\mathsf{obs}(O, I) * \mathsf{channel}(ch, M, M') * M(m) * (\mathsf{M}^{\mathsf{r}}(ch){=}\{\![\,]\!\} \lor \mathsf{trandit}(ch)) \land \\ \qquad \mathsf{levels}(M'(m)) \subseteq \mathsf{M}^{\mathsf{r}}(ch)\}\ \mathsf{send}(ch, m) \\ \quad \{\lambda\_.\ \mathsf{obs}(O{-}\{\![ch]\!\}{-}M'(m), I) * \mathsf{channel}(ch, M, M')\}\end{array}$$

Receive

$$\begin{array}{l}\{\mathsf{obs}(O, I) * \mathsf{channel}(ch, M, M') * \mathsf{credit}(ch) \land ch \prec O \land ch \prec^{r} I\}\ \mathsf{receive}(ch) \\ \quad \{\lambda m.\ \mathsf{obs}(O{\uplus}M'(m), I{-}\{\![ch]\!\}) * \mathsf{channel}(ch, M, M') * M(m)\}\end{array}$$

Fork

$$\frac{\{a * \mathsf{obs}(O, I)\}\ c\ \{\lambda\_.\ \mathsf{obs}(\{\![\,]\!\}, \{\![\,]\!\})\}}{\{a * \mathsf{obs}(O{\uplus}O', I{\uplus}I')\}\ \mathsf{fork}(c)\ \{\lambda\_.\ \mathsf{obs}(O', I')\}}$$

Credit

$$\begin{array}{c}\{\mathsf{obs}(O) * \mathsf{channel}(ch, M, M')\}\ \mathsf{g\_credit}(ch) \\ \{\lambda\_.\ \mathsf{obs}(O{\uplus}\{\![ch]\!\}) * \mathsf{channel}(ch, M, M') * \mathsf{credit}(ch)\}\end{array}$$

Trandit

$$\begin{array}{c}\{\mathsf{obs}(O, I) * \mathsf{channel}(ch, M, M')\}\ \mathsf{g\_trandit}(ch) \\ \{\lambda\_.\ \mathsf{obs}(O, I{\uplus}\{\![ch]\!\}) * \mathsf{channel}(ch, M, M') * \mathsf{trandit}(ch)\}\end{array}$$

Figure 4 The updated proof rules ensuring deadlock-freedom of importer channels, where $\mathsf{M}^{\mathsf{r}}(ch)$ denotes the bag of the levels of the obligations which are possibly imported by ch; M and M' in the permission channel of a channel are functions, that given a message, return the permissions and the obligations which are transferred through that message, respectively; bag I in the permission $\mathsf{obs}(O, I)$ of a thread denotes the channels importing some obligations onto that thread; $\mathsf{levels}(O)$ maps each element of O to its level; and $ch \prec^{r} I \Leftrightarrow \forall ch'{\in}I.\ ch{=}ch' \lor ch \prec \mathsf{M}^{\mathsf{r}}(ch')$.

which are transferred through m as well as an obligation of ch are discharged from the bag of the obligations. Additionally, this rule makes sure by adding m to the queue of ch Invariant 4 is still preserved. As shown in Rule Receive, when a thread tries to receive a message m from ch the level of ch must be lower than the levels of the obligations of the thread and also the levels of the obligations which are possibly imported by all importers of that thread except for ch itself, i.e. $\forall ch'{\in}I.\ ch{=}ch' \lor ch \prec \mathsf{M}^{\mathsf{r}}(ch')$ where I is the bag of the importers of the receiving thread. Additionally, a credit of ch is consumed, the permissions which are transferred through m are produced, ch is discharged from the bag of the importers, and the obligations which are transferred through m are loaded onto the bag of the obligations. As shown in Rule Fork, a thread can transfer a part of its permissions, obligations, and importer channels to a forked thread, provided that the forked thread discharges all of the delegated obligations and importer channels. As shown in Rule Trandit a thread can obtain a transferring credit of ch by loading ch onto the bag of its importers.

The verification of the program in Figure 3 using the proposed proof rules is illustrated in Figure 5. Note that for the sake of readability we elide repeated occurrences of the permissions channel in the proof of the given programs. As shown in the precondition and the postcondition of the routine server, denoted by **req** and **ens**, this routine discharges an importer channel ch, where a message from ch is received. Since this routine tries to wait on ch, it requires a credit and a permission channel for ch. The permission channel in the precondition of this routine indicates that along with the client's request the server receives an obligation and a permission channel for the client's channel through which no permission or obligation is transferred. The specification of the routine client indicates that this routine discharges an obligation for the server channel ch, since it sends a request to this channel. As it is shown in the verification of this routine, after creating the client channel ch' and before sending it to the server, one obligation and one credit for ch' are loaded onto the system. The former is transferred to the server, where it is discharged by sending a response on ch', and the latter is spent in the rest of this routine for receiving from ch'. The routine main in this program is successfully verified, since starting with an empty bag of obligations and importers the verification of this routine is finished with such bags too.

2.3 Conditional Channels

A different variation of a client-server program is shown in Figure 6, where a server keeps accepting the clients' requests until it receives a specific message done. The verification of this program using the proposed proof rules is illustrated in Figure 7. Note that if the client sends a message done to the server the session is finished and the client should not transfer an obligation of ch' through this message, i.e an obligation of ch' is transferred through a message sent on ch only if that message is not a message done. Since in this program a client can send multiple requests to the server and for each request it requires a permission trandit(ch), the server sends this permission to the client each time it replies to the client, i.e. a permission trandit(ch) is transferred through the client's channel ch'. Additionally, since the server might wait for ch more than once and for each wait it requires a credit(ch), before replying to the client a credit and an obligation for ch are loaded onto the server and the loaded obligation is transferred to the client, where this obligation is discharged by sending on ch, i.e. an obligation for ch is transferred through the client's channel ch'. Since the channel ch' transfers some obligations from the server to the client, the server requires a permission trandit(ch') before sending on ch', which can be obtained from the client through the channel ch, i.e. if the client sends a request (and not a message done) to the server it also sends a permission trandit(ch') to the server.

2.4 Server Channels

Another variation of a client-server program is shown in Figure 8, where a server *infinitely* accepts the clients' requests through a *server channel* s. A desired property of these kinds of programs is that if they have a thread waiting for a *non-server* channel they also have a running thread, not waiting for any channel. In other words, a program can be considered as a terminated program even if there are still some specific threads, namely *daemon threads*, such as a server thread or a garbage collector thread, which are not terminated. In Java these threads are terminated by the JVM when there is no longer any user thread running.

To achieve the mentioned property we only need to make sure that if a thread tries to receive from a server channel s it has no obligation and the bag of the importers of this thread only contains s.

It can be proved that any program verified by enforcing the constraint above meets the mentioned desired property, i.e. if this program has a thread waiting for a non-server channel it also has a running thread, not waiting for any channel. Consider an undesired state, where each thread is waiting for a channel while some of these channels are non-server channels. Among all of these non-server channels take the one having a minimal wait level, namely

```
Mch ::= λm. channel(snd(m), λ_. true, λ_. ⦃⦄) ∧ M^r(snd(m))=⦃⦄
M'ch ::= λm. ⦃snd(m)⦄

routine server(channel ch){
req : {obs(⦃⦄, ⦃ch⦄) ∗ channel(ch, Mch, M'ch) ∗ credit(ch)}
 (req, ch') := receive(ch);
 {obs(⦃ch'⦄, ⦃⦄) ∗ channel(ch', λ_. true, λ_. ⦃⦄) ∧ M^r(ch')=⦃⦄}
 result := process(req);
 send(ch', result)
ens : {obs(⦃⦄, ⦃⦄)}}

routine client(channel ch){
req : {obs(⦃ch⦄, ⦃⦄) ∗ channel(ch, Mch, M'ch) ∗ trandit(ch) ∧ M^r(ch)=⦃1⦄}
 ch' := new_channel();
 {obs(⦃ch⦄, ⦃⦄) ∗ channel(ch', λ_. true, λ_. ⦃⦄) ∗ trandit(ch) ∧ R(ch')=1 ∧ M^r(ch')=⦃⦄}
 g_credit(ch');
 {obs(⦃ch, ch'⦄, ⦃⦄) ∗ trandit(ch) ∗ credit(ch')}
 send(ch, (request(), ch'));
 {obs(⦃⦄, ⦃⦄) ∗ credit(ch')}
 receive(ch')
ens : {obs(⦃⦄, ⦃⦄)}}

routine main(){
req : {obs(⦃⦄, ⦃⦄)}
 ch := new_channel();
 {obs(⦃⦄, ⦃⦄) ∗ channel(ch, Mch, M'ch) ∧ R(ch)=1 ∧ M^r(ch)=⦃1⦄}
 g_credit(ch);
 {obs(⦃ch⦄, ⦃⦄) ∗ credit(ch)}
 g_trandit(ch);
 {obs(⦃ch⦄, ⦃ch⦄) ∗ credit(ch) ∗ trandit(ch)}
 fork(
   {obs(⦃⦄, ⦃ch⦄) ∗ credit(ch)}
   server(ch)
   {obs(⦃⦄, ⦃⦄)});
 {obs(⦃ch⦄, ⦃⦄) ∗ trandit(ch)}
 client(ch)
ens : {obs(⦃⦄, ⦃⦄)}}
```

Figure 5 Verification of the program in Figure 3.

```
routine cserver(channel ch){      routine client(channel ch){     routine main()
 (req, ch') := receive(ch);        ch' := new_channel();           {
 while(req ≠ done){                send(ch, (request(), ch'));      ch := new_channel();
   send(ch', process(req));        receive(ch');                    fork(cserver(ch));
   (req, ch') := receive(ch) }}    send(ch, (done, ch'))}           client(ch)}
```

Figure 6 A server keeps serving a client until receiving a specific message done.

```
Mch(ch) ::= λm. channel(snd(m), λ_. trandit(ch), λ_. {ch}) ∧ M^r(snd(m))={1} ∗
  fst(m)=done ? true : trandit(snd(m))
M′ch ::= λm. fst(m)=done ? {} : {snd(m)}

routine cserver(channel ch){
req : {obs({}, {ch}) ∗ channel(ch, Mch(ch), M′ch) ∗ credit(ch) ∧ R(ch)=1 ∧ M^r(ch)={1}}
 (req, ch′) := receive(ch);
 {obs(req=done ? {} : {ch′}, {}) ∗ channel(ch′, λ_. trandit(ch), λ_. {ch}) ∗
  (req=done ? true : trandit(ch′)) ∧ M^r(ch′)={1}}
 while(req ≠ done){
 inv : {M^r(ch′)={1} ∧ req=done ? obs({}, {}) : (obs({ch′}, {}) ∗ trandit(ch′))}
   g_trandit(ch); g_credit(ch);
   {obs({ch′, ch}, {ch}) ∗ trandit(ch′) ∗ trandit(ch) ∗ credit(ch)}
   send(ch′, process(req));
   {obs({}, {ch}) ∗ credit(ch)}
   (req, ch′) := receive(ch)
 }ens : {obs({}, {})}}

routine client(channel ch){
req : {obs({ch}, {}) ∗ channel(ch, Mch(ch), M′ch) ∗ trandit(ch) ∧ R(ch)=1 ∧ M^r(ch)={1}}
 ch′ := new_channel();
 {obs({ch}, {}) ∗ trandit(ch) ∗ channel(ch′, λ_. trandit(ch), λ_. {ch}) ∧ R(ch′)=1 ∧
  M^r(ch′)={1}}
 g_credit(ch′); g_trandit(ch′);
 {obs({ch, ch′}, {ch′}) ∗ trandit(ch) ∗ credit(ch′) ∗ trandit(ch′)}
 send(ch, (request(), ch′));
 {obs({}, {ch′}) ∗ credit(ch′)}
 receive(ch′);
 {obs({ch}, {}) ∗ trandit(ch)}
 send(ch, (done, ch′))
ens : {obs({}, {})}}

routine main(){
req : {obs({}, {})}
 ch := new_channel();
 {obs({}, {}) ∗ channel(ch, Mch(ch), M′ch) ∧ R(ch)=1 ∧ M^r(ch)={1}}
 g_credit(ch) ; g_trandit(ch);
 {obs({ch}, {ch}) ∗ credit(ch) ∗ trandit(ch)}
 fork(
   {obs({}, {ch}) ∗ credit(ch)}
   cserver(ch)
   {obs({}, {})});
 {obs({ch}, {}) ∗ trandit(ch)}
 client(ch)
ens : {obs({}, {})}}
```

Figure 7 Verification of the program in Figure 6, where $a\ ?\ b : c$ evaluates to b if the value of a is true, and otherwise to c.

```
routine server(channel s){     routine client(channel s){     routine main(){
 while(true){                   ch' := new_channel();           s := new_channel();
   (req, ch') := receive(s);    send(s, (request(), ch'));      fork(server(s));
   send(ch', process(req))      receive(ch')                    fork(client(s));
 }}                            }                                client(s)}
```

Figure 8 A server keeps serving clients.

NewChannel

$$\{\mathsf{true}\}\ \mathsf{new\_channel}\ \{\lambda ch.\ \mathsf{channel}(ch, M, M') \wedge \mathsf{R}(ch){=}r \wedge \mathsf{M}^{\mathsf{r}}(ch){=}R \wedge \mathsf{S}(ch){=}b\}$$

Receive

$$\{\mathsf{obs}(O, I) * \mathsf{channel}(ch, M, M') * (\mathsf{S}(ch) \vee \mathsf{credit}(ch)) \wedge ch \prec O \wedge ch \prec^{r} I \wedge (\neg\mathsf{S}(ch) \vee (O{=}\{\![\,]\!\} \wedge \forall ch'{\in}I.\ ch'{=}ch))\}\ \mathsf{receive}(ch)$$
$$\{\lambda m.\ \mathsf{obs}(O{\uplus}M'(m), I{-}\{\![ch]\!\}) * \mathsf{channel}(ch, M, M') * M(m)\}$$

Trandits

$$\{\mathsf{obs}(O, I) * \mathsf{channel}(ch, M, M')\}\ \mathsf{g\_trandits}(ch)$$
$$\{\lambda\_.\ \mathsf{obs}(O, I{\uplus}\{\![ch^{\infty}]\!\}) * \mathsf{channel}(ch, M, M') * \mathsf{trandit}^{\infty}(ch)\}$$

Figure 9 The updated proof rules ensuring deadlock-freedom of server channels, where S is a function that given a channel specifies whether that channel is a server channel or not, and o^{∞} represents an infinite number of occurrences of o.

ch_{min}. Since $\mathsf{size}(ch_{min}){=}0$ and there exists a credit for ch_{min} in the system held by the waiting thread, by Invariant 3, either 1) there exists a thread having an obligation of ch_{min} that is waiting for a channel ch whose level is lower than the level of ch_{min}, or 2) there exists a message m in a channel ch through which an obligation of ch_{min} is transferred, i.e. $0{<}\mathsf{M}'(ch)(m)(ch_{min})$ which by Invariant 4 implies $\mathsf{R}(ch_{min}){\in}\mathsf{M}^{\mathsf{r}}(ch)$, where $\mathsf{R}(ch_{min})$ denotes the level of ch_{min}. The first case contradicts minimality of the level of ch_{min}, because in this case ch is a non-server channel, since that thread is waiting for ch while it has some obligations. In the second case, since $0{<}\mathsf{size}(ch)$ and $\mathsf{M}^{\mathsf{r}}(ch){\neq}\{\![\,]\!\}$, by Invariant 2, there exists a thread t having an importer channel ch that is waiting for a channel ch_1 whose level is lower than the level of ch_{min} (because ch imports an obligation of level of ch_{min}). Since $0{<}\mathsf{size}(ch)$ we know $ch_1{\neq}ch$. Accordingly, ch_1 cannot be a server channel (because t is trying to receive from ch_1 while it has an importer channel ch which is not equal to ch_1), which is a contradiction.

The proof rules which need to be updated are shown in Figure 9. As shown in Rule NewChannel, when a channel is created it must be specified whether this channel is a server channel or not, which is denoted by a function S. As shown in Rule Receive, if a thread tries to receive from a server channel ch the bag of the obligations of this thread must be empty and the bag of the importer channels of this thread must only contain ch. Note that a thread does not need to spend a credit for waiting on a server channel. We also introduce a new ghost command $\mathsf{g\_trandits}(ch)$, shown in Rule Trandits, which produces an infinite number of transferring credits of ch, denote by $\mathsf{trandit}^{\infty}(ch)$, by loading an infinite number of ch, denoted by ch^{∞}, onto the importers. The verification of the program in Figure 8 using these rules is shown in Figure 10, where the server does not need any credit for waiting on the server channel s. Note that this verification ensures neither termination nor deadlock-freedom. It actually ensures that if this program has a thread waiting for the non-server channel ch' it also has a running thread, not waiting for any channel.

```
$\mathsf{Ms}(s) ::= \lambda m.\ \mathsf{channel}(\mathsf{snd}(m), \lambda\_.\ \mathsf{trandit}(s), \lambda\_.\ \{\!\!\{\}\!\!\}) \land \mathsf{M^r}(\mathsf{snd}(m)){=}\{\!\!\{\}\!\!\}$
$\mathsf{M's} ::= \lambda m.\ \{\!\!\{\mathsf{snd}(m)\}\!\!\}$

routine server(channel $s$){
req : $\{\mathsf{obs}(\{\!\!\{\}\!\!\}, \{\!\!\{s^\infty\}\!\!\}) * \mathsf{channel}(s, \mathsf{Ms}(s), \mathsf{M's}) \land \mathsf{M^r}(s){=}\{\!\!\{1\}\!\!\} \land \mathsf{S}(s)\}$
 while(true){
 inv : $\{\mathsf{obs}(\{\!\!\{\}\!\!\}, \{\!\!\{s^\infty\}\!\!\})\}$
   $(req, ch')$ := receive($s$);
   $\{\mathsf{obs}(\{\!\!\{ch'\}\!\!\}, \{\!\!\{s^\infty\}\!\!\}{-}\{\!\!\{s\}\!\!\}) * \mathsf{channel}(ch', \lambda\_.\ \mathsf{trandit}(s), \lambda\_.\ \{\!\!\{\}\!\!\}) \land \mathsf{M^r}(ch'){=}\{\!\!\{\}\!\!\}\}$
   g_trandit($s$);
   $\{\mathsf{obs}(\{\!\!\{ch'\}\!\!\}, \{\!\!\{s^\infty\}\!\!\}) * \mathsf{trandit}(s)\}$
   send($ch'$, process($req$))
   $\{\mathsf{obs}(\{\!\!\{\}\!\!\}, \{\!\!\{s^\infty\}\!\!\})\}$
 }
ens : $\{\mathsf{false}\}$}

routine client(channel $s$){
req : $\{\mathsf{obs}(\{\!\!\{\}\!\!\}, \{\!\!\{\}\!\!\}) * \mathsf{channel}(s, \mathsf{Ms}(s), \mathsf{M's}) * \mathsf{trandit}(s) \land \mathsf{M^r}(s){=}\{\!\!\{1\}\!\!\}\}$
 $ch'$ := new_channel();
 $\{\mathsf{obs}(\{\!\!\{\}\!\!\}, \{\!\!\{\}\!\!\}) * \mathsf{trandit}(s) * \mathsf{channel}(ch', \lambda\_.\ \mathsf{trandit}(s), \lambda\_.\ \{\!\!\{\}\!\!\}) \land \mathsf{R}(ch'){=}1 \land \mathsf{M^r}(ch'){=}\{\!\!\{\}\!\!\}\}$
 g_credit($ch'$);
 $\{\mathsf{obs}(\{\!\!\{ch'\}\!\!\}, \{\!\!\{\}\!\!\}) * \mathsf{trandit}(s) * \mathsf{credit}(ch')\}$
 send($s$, (request(), $ch'$));
 $\{\mathsf{obs}(\{\!\!\{\}\!\!\}, \{\!\!\{\}\!\!\}) * \mathsf{credit}(ch')\}$
 receive($ch'$)
 $\{\mathsf{obs}(\{\!\!\{\}\!\!\}, \{\!\!\{\}\!\!\}) * \mathsf{trandit}(s)\}$
ens : $\{\mathsf{obs}(\{\!\!\{\}\!\!\}, \{\!\!\{\}\!\!\})\}$}

routine main(){
req : $\{\mathsf{obs}(\{\!\!\{\}\!\!\}, \{\!\!\{\}\!\!\}\}$
 $s$ := new_channel();
 $\{\mathsf{obs}(\{\!\!\{\}\!\!\}, \{\!\!\{\}\!\!\}) * \mathsf{channel}(s, \mathsf{Ms}(s), \mathsf{M's}) \land \mathsf{M^r}(s){=}\{\!\!\{1\}\!\!\}\}$
 g_trandits($s$);
 $\{\mathsf{obs}(\{\!\!\{\}\!\!\}, \{\!\!\{s^\infty\}\!\!\}) * \mathsf{trandit}^\infty(s)\}$
 fork(
   $\{\mathsf{obs}(\{\!\!\{\}\!\!\}, \{\!\!\{s^\infty\}\!\!\})\}$
   server($s$));
 $\{\mathsf{obs}(\{\!\!\{\}\!\!\}, \{\!\!\{\}\!\!\}) * \mathsf{trandit}^\infty(s)\}$
 fork(client($s$));
 $\{\mathsf{obs}(\{\!\!\{\}\!\!\}, \{\!\!\{\}\!\!\}) * \mathsf{trandit}^\infty(s)\}$
 client($s$)
ens : $\{\mathsf{obs}(\{\!\!\{\}\!\!\}, \{\!\!\{\}\!\!\})\}$}
```

Figure 10 Verification of the program in Figure 8.

```
routine server(channel $s$){
 while(true){
   $(thr, ch')$ := receive($s$);
   $ch$ := new_channel();
   send($ch'$, (through, $ch$));
   fork(cserver($ch$))
 }}
```

```
routine client(channel $s$){
 $ch'$ := new_channel();
 send($s$, (through, $ch'$));
 $(thr, ch)$ := receive($ch'$);
 send($ch$, (request(), $ch'$));
 $(res, ch)$ := receive($ch'$);
 send($ch$, (done, $ch'$))}
```

```
routine main()
{
 $s$ := new_channel();
 fork(server($s$));
 fork(client($s$));
 client($s$)
}
```

Figure 11 A server keeps serving clients through separate conditional channels.

Using the proposed proof rules it is also possible to verify some other variations of client-server programs such as the one shown in Figure 11, where a server infinitely serves clients' request through separate conditional channels (see Appendix A illustrating the proof of this program). Note that the definition of the function cserver in this figure is similar to the one shown in Figure 6.

3 Transferring Obligations Through Notifications

In this section we introduce a mechanism which allows obligations to be transferred from a thread notifying a condition variable (CV) to the notified thread. The main idea behind this mechanism is based on this unique feature of condition variables that when a sender thread notifies a CV, if there is a receiver thread waiting for that CV, the receiver thread *immediately* receives this notification. Accordingly, if there exists a thread waiting for a condition variable v, it is safe to discharge the transferred obligations of v when v is notified and load these obligations to the receiver thread as it is notified. Note that a notification on a condition variable is lost if there is no thread waiting for that condition variable.

The number of threads waiting for a condition variable can be tracked by using the approach introduced by Hamin et al. [15, 16], which modularly verifies deadlock-freedom of programs synchronized by monitors. Since this approach only allows permissions to be transferred from a notifying thread to the one notified, it cannot verify some interesting programs such as a particular implementation of fair readers-writers locks, shown in Figure 19, and dining philosophers (see Appendix B). In the following we first review this approach and then we extend this approach such that it also allows transferring of obligations, enabling it to verify a wider range of programs such as the ones we just mentioned.

3.1 Verifying Monitors

Hamin et al. [15, 16] introduced a modular approach for verifying deadlock-freedom of programs synchronized by condition variables (CVs), where executing a command wait(v, l) on a CV v, which is associated with a lock l, releases l and suspends the running thread, and executing a command notify(v)/notifyAll(v) wakes up one/all thread(s) waiting for CV v, if any. This approach ensures absence of deadlock by making sure that for any CV v for which a thread is waiting there is a thread obliged to fulfill an obligation for v which only waits for waitable objects whose levels are lower than the level of v. In this approach when a thread acquires a lock l, the total number of waiting threads, and the total number of obligations of any CV v associated with l, denoted by $Wt(v)$ and $Ot(v)$ respectively, can be mentioned in the proof of that thread. In order to ensure the mentioned constraint this approach makes sure that 1) if a command wait(v, l) is executed then $0 < Ot(v)$, i.e. there is an obligation of v in the system, 2) if an obligation of v is discharged then after this discharge the invariant

NewLock

$$\{\mathsf{true}\}\ \mathsf{newlock}\ \{\lambda l.\ \mathsf{ulock}(l, \{\!\!\{\}\!\!\}, \{\!\!\{\}\!\!\}) \wedge \mathsf{R}(l){=}r\}$$

InitLock

$$\{\mathsf{ulock}(l,\ Wt, Ot) \ast inv(Wt, Ot) \ast \mathsf{obs}(O)\}\ \mathsf{g\_initl}(l)\ \{\lambda\_.\ \mathsf{lock}(l) \ast \mathsf{obs}(O) \wedge \mathsf{I}(l){=}inv\}$$

Acquire

$$\{\mathsf{lock}(l) \ast \mathsf{obs}(O) \wedge l \prec O\}\ \mathsf{acquire}(l)$$
$$\{\lambda\_.\ \exists Wt, Ot.\ \mathsf{locked}(l,\ Wt, Ot) \ast \mathsf{I}(l)(Wt, Ot) \ast \mathsf{obs}(O \uplus \{\!\!\{l\}\!\!\})\}$$

Release

$$\{\mathsf{locked}(l,\ Wt, Ot) \ast \mathsf{I}(l)(Wt, Ot) \ast \mathsf{obs}(O \uplus \{\!\!\{l\}\!\!\})\}\ \mathsf{release}(l)\ \{\lambda\_.\ \mathsf{lock}(l) \ast \mathsf{obs}(O)\}$$

NewCV

$$\{\mathsf{true}\}\ \mathsf{new\_cvar}\ \{\lambda v.\ \mathsf{ucond}(v) \wedge \mathsf{R}(v){=}r\}$$

InitCV

$$\{\mathsf{ucond}(v) \ast \mathsf{ulock}(l,\ Wt, Ot)\}\ \mathsf{g\_initc}(v)\ \{\lambda\_.\ \mathsf{cond}(v, M) \ast \mathsf{ulock}(l,\ Wt, Ot) \wedge \mathsf{L}(v){=}l\}$$

Wait

$$\{\mathsf{cond}(v, M) \ast \mathsf{locked}(l,\ Wt, Ot) \ast \mathsf{I}(l)(Wt \uplus \{\!\!\{v\}\!\!\}, Ot) \ast \mathsf{obs}(O \uplus \{\!\!\{l\}\!\!\}) \wedge$$
$$l{=}\mathsf{L}(v) \wedge v \prec O \wedge l \prec O \wedge \mathsf{enoughObs}(v,\ Wt \uplus \{\!\!\{v\}\!\!\}, Ot)\}\ \mathsf{wait}(v, l)$$
$$\{\lambda\_.\ \mathsf{cond}(v, M) \ast \mathsf{obs}(O \uplus \{\!\!\{l\}\!\!\}) \ast \exists Wt', Ot'.\ \mathsf{locked}(l,\ Wt', Ot') \ast \mathsf{I}(l)(Wt', Ot') \ast M\}$$

Notify

$$\{\mathsf{cond}(v, M) \ast \mathsf{locked}(\mathsf{L}(v),\ Wt, Ot) \ast (Wt(v){=}0 \vee M)\}\ \mathsf{notify}(v)$$
$$\{\lambda\_.\ \mathsf{cond}(v, M) \ast \mathsf{locked}(\mathsf{L}(v),\ Wt{-}\{\!\!\{v\}\!\!\}, Ot)\}$$

NotifyAll

$$\{\mathsf{cond}(v, M) \ast \mathsf{locked}(\mathsf{L}(v),\ Wt, Ot) \ast (\mathop{\ast}_{i:=1}^{Wt(v)} M)\}\ \mathsf{notifyAll}(v)$$
$$\{\lambda\_.\ \mathsf{cond}(v, M) \ast \mathsf{locked}(\mathsf{L}(v),\ Wt[v{:=}0], Ot)\}$$

ChargeObligation

$$\{\mathsf{obs}(O) \ast \mathsf{ulock/locked}(\mathsf{L}(v),\ Wt, Ot)\}\ \mathsf{g\_chrg}(v)$$
$$\{\lambda\_.\ \mathsf{obs}(O \uplus \{\!\!\{v\}\!\!\}) \ast \mathsf{ulock/locked}(\mathsf{L}(v),\ Wt, Ot \uplus \{\!\!\{v\}\!\!\})\}$$

DischargeObligation

$$\{\mathsf{obs}(O) \ast \mathsf{ulock/locked}(\mathsf{L}(v),\ Wt, Ot) \wedge \mathsf{enoughObs}(v,\ Wt, Ot{-}\{\!\!\{v\}\!\!\})\}\ \mathsf{g\_disch}(v)$$
$$\{\lambda\_.\ \mathsf{obs}(O{-}\{\!\!\{v\}\!\!\}) \ast \mathsf{ulock/locked}(\mathsf{L}(v),\ Wt, Ot{-}\{\!\!\{v\}\!\!\})\}$$

Figure 12 Proof rules verifying deadlock-freedom of monitors, where $Wt(v)$ and $Ot(v)$ denote the total number of threads waiting for v and the total number of obligations for v, respectively; the parameter M in the permission cond of a condition variable denotes the permissions which are transferred from the thread notifying that condition variable to the one(s) notified; $\mathsf{L}(v)$ denotes the lock associated with the condition variable v; $\mathsf{enoughObs}(v, Wt, Ot) \Leftrightarrow (Wt(v){>}0 \Rightarrow Ot(v){>}0)$; and $v \prec O \Leftrightarrow \forall o{\in}O.\ \mathsf{R}(v) < \mathsf{R}(o)$.

$0 < Wt(v) \Rightarrow 0 < Ot(v)$ holds, i.e. if there is a thread waiting for v then after this discharge there are still some obligations for v in the system, and 3) a thread executes a command $\mathsf{wait}(v, l)$ only if the level of v is lower than the levels of the obligations of that thread.

A program in this approach can be successfully verified if each lock associated with some CVs has an appropriate invariant such that for any CV v associated with that lock this invariant implies $0 < Wt(v) \Rightarrow 0 < Ot(v)$. Accordingly, in this approach each lock invariant is parametrized over the bags Wt and Ot, which map all CVs associated with that lock to the number of their waiting threads and obligations, respectively.

The proof rules proposed in this approach are shown in Figure 12. As shown in Rule NewLock, when a lock l is created an arbitrary level is assigned to that lock and an uninitialized lock permission $\mathsf{ulock}(l, \{\!\}, \{\!\})$ is produced. The second and the third parameters of this permission are two bags mapping the CVs associated with l to their number of waiting threads and obligations, respectively. As shown in Rule InitLock, this uninitialized lock permission can be converted to a (*duplicable and leakable*) lock permission $\mathsf{lock}(l)$ if the assertion resulting from applying the invariant of that lock, denoted by $\mathsf{I}(l)$, to the bags stored in the permission ulock is consumed (the permissions described by the invariant of this lock are transferred from the thread to the lock). As shown in Rule Acquire, when a thread acquires this lock the permissions described by the invariant of this lock are transferred from the lock to the thread. Additionally, a permission $\mathsf{locked}(l, Wt, Ot)$ is provided for the thread, where Wt and Ot are two bags mapping the CVs associated with l to their number of waiting threads and obligations, respectively, and are existentially quantified in the postcondition. Note that to prevent circular dependencies the precondition of this rule enforces that the level of l be lower than the levels of the obligations of the acquiring thread. Additionally, this lock is added to the bag of the obligations of this thread. As shown in rule Release, when this lock is released it is discharged from the bag of the obligations and the assertion resulting from applying the invariant of this lock to the bags stored in the permission locked is consumed. Additionally, the permission locked is converted to a permission lock.

As shown in Rule NewCV, when a CV is created an arbitrary level is assigned to it and an uninitialized permission ucond for that CV is produced. As shown in Rule InitCV, this permission can be converted to a (*duplicable and leakable*) permission cond if a lock is associated to this CV, denoted by $\mathsf{L}(v)$. Additionally, the transferred permissions of this CV, which are transferred from the notifying thread to the one notified, are also specified in this rule, denoted by M in the permission cond. These permissions are consumed when a command $\mathsf{notify}(v)$ is executed (if there is a thread waiting for v; see the precondition of Rule Notify), and are produced when a command $\mathsf{wait}(v, l)$ is executed (see the postcondition of Rule Wait). Note that $\mathsf{notifyAll}(v)$ transfers $Wt(v)$ instances of these permissions, denoted by $\mathop{*}\limits_{i:=1}^{Wt(v)} M$ (see the precondition of Rule NotifyAll). As shown in Rule Wait, when a command $\mathsf{wait}(v, l)$ is executed, since l is going to be released and the number of threads waiting for v is going to be increased, the result of applying the invariant of lock l to bags $Wt \uplus \{\!v\!\}$ and Ot must be consumed, where Wt and Ot are the bags stored in the permission locked of l. Additionally, the level of v must be lower than the levels of all obligations of the thread except for l. Note that the level of l must be lower than the levels of these obligations, too, since when the thread is woken up it tries to reacquire l. As previously mentioned, the precondition of this rule also makes sure that $0 < Ot(v)$, which is enforced by the invariant $\mathsf{enoughObs}(v, Wt \uplus \{\!v\!\}, Ot)$. This invariant follows from $\mathsf{I}(l)(Wt \uplus \{\!v\!\}, Ot)$ if the invariant of l is properly defined such that for any CV v' associated with l and any Wt' and Ot', we have $\mathsf{I}(l)(Wt, Ot) \Rightarrow \mathsf{enoughObs}(v', Wt, Ot)$. Lastly the precondition of this command makes sure that v is associated with lock l, which is enforced by $\mathsf{L}(v) = l$. As

InitCV

$$\{\mathsf{ucond}(v) * \mathsf{ulock}(l,\ Wt, Ot)\}\ \mathsf{g\_initc}(v)$$
$$\{\lambda\_.\ \mathsf{cond}(v, M, M') * \mathsf{ulock}(l,\ Wt, Ot) \wedge \mathsf{L}(v){=}l\}$$

Wait

$$\{\mathsf{cond}(v, M, M') * \mathsf{locked}(l,\ Wt, Ot) * \mathsf{I}(l)(Wt\uplus\{\!\!\{v\}\!\!\}, Ot) * \mathsf{obs}(O\uplus\{\!\!\{l\}\!\!\}) \wedge$$
$$l{=}\mathsf{L}(v) \wedge v{\prec}O \wedge l{\prec}O\uplus M' \wedge \mathsf{enoughObs}(v,\ Wt\uplus\{\!\!\{v\}\!\!\}, Ot)\}\ \mathsf{wait}(v, l)$$
$$\{\lambda\_.\ \mathsf{cond}(v, M, M') * \mathsf{obs}(O\uplus\{\!\!\{l\}\!\!\}\uplus M') * \exists\, Wt', Ot'.\ \mathsf{locked}(l,\ Wt', Ot') * \mathsf{I}(l)(Wt', Ot') * M\}$$

Notify

$$\{\mathsf{obs}(O\uplus(0{<}\,Wt(v)\ ?\ M' : \{\!\!\}\!\!\})) * \mathsf{cond}(v, M, M') * \mathsf{locked}(\mathsf{L}(v),\ Wt, Ot) * (Wt(v){=}0 \vee M)\}$$
$$\mathsf{notify}(v)\ \{\lambda\_.\ \mathsf{obs}(O) * \mathsf{cond}(v, M, M') * \mathsf{locked}(\mathsf{L}(v),\ Wt{-}\{\!\!\{v\}\!\!\}, Ot)\}$$

NotifyAll

$$\{\mathsf{cond}(v, M, \{\!\!\}\!\!\}) * \mathsf{locked}(\mathsf{L}(v),\ Wt, Ot) * (\mathop{*}_{i:=1}^{Wt(v)} M)\}\ \mathsf{notifyAll}(v)$$
$$\{\lambda\_.\ \mathsf{cond}(v, M, \{\!\!\}\!\!\}) * \mathsf{locked}(\mathsf{L}(v),\ Wt[v{:=}0], Ot)\}$$

Figure 13 New proof rules verifying deadlock-freedom of monitors allowing transferring obligations through notifications, where the parameter M' in the permission cond of a condition variable denotes the obligations which are transferred from the thread notifying that condition variable to the one notified.

shown in Rules Notify/NotifyAll, when a CV v is notified, one/all instance(s) of v is/are removed from the bag Wt stored in the permission locked of the lock associated with v, if any. Unlike the Leino et al. approach [33], this approach has no notion of credits and an obligation for a CV v is loaded/unloaded when that obligation is also loaded/unloaded onto/from the bag Ot stored in the permission locked of the lock associated with v, as shown in Rules ChargeObligation and DischargeObligation. However, an obligation for v is discharged only if after this discharge we have $0 < Wt(v) \Rightarrow 0 < Ot(v)$, which is enforced by the invariant enoughObs in the precondition of the rule DischargeObligation.

3.2 Transferring Obligations Through Notifications

In this subsection we extend the Hamin et al. approach [15] such that it also allows obligations to be transferred from the notifying thread to the one notified. To this end we make sure that 1) when a thread notifies a CV v the transferred obligations of v, which are transferred through a notification on v, are discharged from the thread only if there is a thread waiting for v which is notified, that is $0{<}\,Wt(v)$, 2) when a thread waits for a CV v the transferred obligations of v are loaded onto the bag of obligations of that thread as that thread is notified, and 3) when a thread waits for a CV v the level of the lock associated with v is lower than the levels of the transferred obligations of v, too, since when the thread wakes up, where these obligations are loaded onto the bag of the obligations of the thread, this lock must be reacquired. For the sake of simplicity, any CV v on which notifyAll(v) is performed must have no transferred obligations. The proof rules which need to be updated are shown in Figure 13.

It can be proved that any program verified by the mentioned proof rules, where the verification starts from an empty bag of obligations and also ends with such bag, never deadlocks, i.e. it always has a running thread, not waiting for any waitable object such as a condition variable or a lock, until it terminates. We know that for any waitable object o all

```
routine new_mutex(){
 l := new_lock;
 v := new_cvar;
 mutex(l:=l, v:=v,
   b:=new_bool(false))
}
```

```
routine enter_cs(mutex m){
 acquire(m.l);
 while(m.b)
   wait(m.v, m.l);
 m.b := true;
 release(m.l)}
```

```
routine exit_cs(mutex m){
 acquire(m.l);
 m.b := false;
 notify(m.v);
 release(m.l)
}
```

Figure 14 Mutexes.

```
routine main(){
 m := new_mutex();
 fork(
   while(1){
      enter_cs(m);  /* CS */
      exit_cs(m)});
 enter_cs(m);  /* CS */
 exit_cs(m)}
```

Figure 15 The main thread is starved if it is scheduled only when the forked thread is in the CS.

of these proof rules preserve the invariant $0 < Wt(o) \Rightarrow 0 < Ot(o)$, where $Wt(o)$ and $Ot(o)$ denote the total number of waiting threads and obligations for o in the system, respectively. Note that this invariant holds even when some obligations are transferred because these obligations are immediately transferred from the notifying thread to the one notified. Now consider a deadlocked state, where each thread of a verified program is waiting for an object. Among all of these objects take the one having a minimal wait level, namely o_{min}. By the invariant above, there exists a thread having an obligation for o_{min} that is waiting for an object whose level is lower than the level of o_{min}, which contradicts minimality of the level of o_{min}.

3.3 Fair Mutexes

In this section we show how our extension helps to verify a fair implementation of a mutex, in which threads are synchronized by CVs. Before introducing this implementation, consider a simple (unfair) implementation of a mutex, shown in Figure 14. In this implementation a mutex consists of a boolean variable b, indicating whether the critical section (CS) is executed by any thread or not, a lock l, protecting this variable from concurrent accesses, and a CV v, preventing threads from entering the CS if there is a thread running that CS. As shown in the routine enter_cs, when a thread tries to enter a CS, protected by a mutex m, it first

```
routine new_mutex(){
 l := new_lock;
 v := new_cvar;
 mutex(l:=l, v:=v,
   b:=new_bool(false),
   w:=new_int(0))
}
```

```
routine enter_cs(mutex m){
 acquire(m.l);
 if(m.b){
   m.w := m.w+1;
   wait(m.v, m.l)}
 else
   m.b := true;
 release(m.l)}
```

```
routine exit_cs(mutex m){
 acquire(m.l);
 m.b := false;
 if(0<m.w){
   m.w := m.w−1;
   m.b := true;
   notify(m.v)}
 release(m.l)}
```

Figure 16 Fair mutexes on top of fair monitors.

```
$\mathsf{mutex}(\mathsf{mutex}\ m, \mathsf{waitobj}\ o) = \mathsf{lock}(m.l) * \mathsf{cond}(m.v, \mathsf{true}, \{\!\{m.v\}\!\}) \wedge$
 $\mathsf{I}(m.l){=}\mathsf{linv}(m) \wedge \mathsf{L}(m.v){=}m.l \wedge \mathsf{R}(m.l) < \mathsf{R}(m.v) \wedge o{=}m.v$

$\mathsf{linv}(\mathsf{mutex}\ m) =$
 $\lambda Wt.\ \lambda Ot.\ \exists b, w.\ m.b \mapsto b * m.w \mapsto w \wedge Wt(m.v){=}w \wedge (b\ ?\ 0 < Ot(v) : Wt(v) = 0)$

routine new_mutex(){
req : {true}
 $l$ := new_lock;
 $\{\mathsf{ulock}(l, \{\!\{\}\!\}, \{\!\{\}\!\}) \wedge \mathsf{R}(l){=}r{-}1\}$
 $v$ := new_cvar; g_initc($v$);
 $\{\mathsf{ulock}(l, \{\!\{\}\!\}, \{\!\{\}\!\}) * \mathsf{cond}(v, \mathsf{true}, \{\!\{v\}\!\}) \wedge \mathsf{R}(v){=}r \wedge \mathsf{L}(v){=}l\}$
 $m$ := mutex($l{:=}l, v{:=}v, b{:=}$new_bool(false)$, w{:=}$new_int(0)); g_initl($l$); $m$
ens : $\{\lambda m.\ \exists o.\ \mathsf{mutex}(m, o) \wedge \mathsf{R}(o){=}r\}\}$

routine enter_cs(mutex $m$){
req : $\{\mathsf{obs}(O) * \mathsf{mutex}(m, o) \wedge o \prec O\}$
 acquire($m.l$);
 $\{\mathsf{obs}(O{\uplus}\{\!\{m.l\}\!\}) * \mathsf{mutex}(m, o) * \exists Wt, Ot.\ \mathsf{locked}(m.l, Wt, Ot) * \mathsf{linv}(m)(Wt, Ot)\}$
 if($m.b$){
   $m.w := m.w{+}1$;
   $\{\mathsf{obs}(O{\uplus}\{\!\{m.l\}\!\}) * \mathsf{mutex}(m, o) * \exists Wt, Ot.\ \mathsf{locked}(m.l, Wt, Ot) * \mathsf{linv}(m)(Wt{\uplus}\{\!\{m.v\}\!\}, Ot)\}$
   wait($m.v, m.l$)
   $\{\mathsf{obs}(O{\uplus}\{\!\{m.l, m.v\}\!\}) * \mathsf{mutex}(m, o) * \exists Wt, Ot.\ \mathsf{locked}(m.l, Wt, Ot) * \mathsf{linv}(m)(Wt, Ot)\}\}$
 else{
   $m.b$ := true; g_chrg($m.v$)
   $\{\mathsf{obs}(O{\uplus}\{\!\{m.l, m.v\}\!\}) * \mathsf{mutex}(m, o) * \exists Wt, Ot.\ \mathsf{locked}(m.l, Wt, Ot) * \mathsf{linv}(m)(Wt, Ot)\}$
 };
 $\{\mathsf{obs}(O{\uplus}\{\!\{m.l, m.v\}\!\}) * \mathsf{mutex}(m, o) * \exists Wt, Ot.\ \mathsf{locked}(m.l, Wt, Ot) * \mathsf{linv}(m)(Wt, Ot)\}$
 release($m.l$)
ens : $\{\mathsf{obs}(O{\uplus}\{\!\{o\}\!\}) * \mathsf{mutex}(m, o)\}\}$

routine exit_cs(mutex $m$){
req : $\{\mathsf{obs}(O{\uplus}\{\!\{o\}\!\}) * \mathsf{mutex}(m, o) \wedge o \prec O\}$
 acquire($m.l$);
 $\{\mathsf{obs}(O{\uplus}\{\!\{m.v, m.l\}\!\}) * \mathsf{mutex}(m, o) * \exists Wt, Ot.\ \mathsf{locked}(m.l, Wt, Ot) * \mathsf{linv}(m)(Wt, Ot)\}$
 $m.b$ := false;
 if($0{<}m.w$){
   $m.w := m.w{-}1$; $m.b$ := true; notify($m.v$)
   $\{\mathsf{obs}(O{\uplus}\{\!\{m.l\}\!\}) * \mathsf{mutex}(m, o) * \exists Wt, Ot.\ \mathsf{locked}(m.l, Wt, Ot) * \mathsf{linv}(m)(Wt, Ot)\}\}$
 else{
   g_disch($m.v$)
   $\{\mathsf{obs}(O{\uplus}\{\!\{m.l\}\!\}) * \mathsf{mutex}(m, o) * \exists Wt, Ot.\ \mathsf{locked}(m.l, Wt, Ot) * \mathsf{linv}(m)(Wt, Ot)\}\}$;
 release($m.l$)
ens : $\{\mathsf{obs}(O) * \mathsf{mutex}(m, o)\}\}$
```

Figure 17 Verification of the fair mutexes implementation shown in Figure 16.

acquires the mutex's lock and while there is a thread running the CS it releases that lock and suspends itself. If that thread is notified (and reacquires the mutex's lock) while there is no thread running the CS it changes the value of the variable b to true, preventing other threads from entering the CS, and releases the mutex's lock. Before leaving the critical section, as shown in routine exit_cs, this thread acquires the mutex's lock, changes the value of the variable b to false, allowing other threads to enter the CS, notifies the condition variable of the mutex, waking up a waiting thread, if any, and finally releases the mutex's lock.

However, one problem with this implementation is that a thread might be in *starvation*; it might infinitely wait for entering the CS. For example, consider the program in Figure 15, where the main thread is starved if it is scheduled only when the forked thread is in the CS. In this situation, when the forked thread exits the CS, since $m.b$=false, this thread without waiting for $m.v$ again changes $m.b$ to true and enters the CS. To address this problem a new variable w can be added to the structure of the mutex, tracking the number of threads waiting for that mutex. Introducing this variable, the operations enter_cs and exit_cs can be updated, as shown in Figure 16. As shown in the routine enter_cs(m), when a thread tries to enter a CS, if the CS is currently executed by another thread ($m.b$=true) this thread increases the variable $m.w$ and waits for a notification on $m.v$. Otherwise, this threads changes $m.b$ to true and continue its execution. As shown in the routine exit_cs(m), when a thread leaves a CS it first changes $m.b$ to false and if there is a thread waiting for this CS it decreases the number of the waiting threads, changes $m.b$ to true and notifies a thread waiting for $m.v$. Having this implementation, the forked thread in Figure 15 cannot enter the critical section if the main thread is already waiting to enter. Note that this mutex is fair under the assumption that the monitor primitives are fair, i.e. the lock and condition variable wait queues are FIFO.

This implementation can be verified against the expected specifications, shown in Figure 17, if an obligation of $m.v$ is transferred from the thread leaving the CS to the next thread entering the CS. This transfer is necessary because one of the desired invariants of such program is $m.b \Rightarrow 0 < Ot(m.v)$, that is if the CS is executed by any thread there exists an obligation of $m.v$ in the system. Since in the fair implementation of exit_cs before notifying $m.v$ the variable $m.b$ is changed to true, an obligation of $m.v$ must be loaded onto the system. This obligation can be transferred to the notified thread, which is going to enter the CS. Note that this transfer is sound since it is only performed when $0 < m.w$, that is there is a thread waiting for $m.v$ which immediately receives the transferred obligation.

3.4 Fair Readers-Writers Locks

In this section we show how our extension makes it possible to verify a fair implementation of a readers-writers lock, which is synchronized by CVs. Before that, consider a naive implementation of a readers-writers lock (writers-preference), shown in Figure 18, which can be verified by the Hamin et al. [15] approach. This lock consists of three variables aw, ww, and ar, keeping track of the total number of active writers, waiting writers, and active readers respectively, a lock l, protecting these variables from concurrent accesses, a condition variable v_w, preventing writers from writing while other threads are reading or writing, and a condition variable v_r, preventing readers from reading while there is another thread writing or waiting to write.

However, in this implementation due to the same reason mentioned in the previous section a writer might be starved by other writers, where for example a writer continuously releases the writing lock and without waiting for the CV v_w immediately reacquires it.

```
routine new_rdwr(){
 $l$ := new_lock();
 $v_w$ := new_cvar;
 $v_r$ := new_cvar;
 rdwr($l$:=$l$, $v_w$:=$v_w$, $v_r$:=$v_r$,
   $aw$ := new_int(0), $ww$ := new_int(0),
   $ar$ := new_int(0))}

routine acquire_write(rdwr $b$){
 acquire($b.l$);
 while($b.aw$+$b.ar$>0){
   $b.ww$ := $b.ww$+1;
   wait($b.v_w$, $b.l$);
   $b.ww$ := $b.ww$−1 };
 $b.aw$ := $b.aw$+1;
 release($b.l$)}

routine release_write(rdwr $b$){
 acquire($b.l$);
 $b.aw$ := $b.aw$−1;
 notify($b.v_w$);
 if($b.ww$ = 0)
   notifyAll($b.v_r$);
 release($b.l$)}
```

```
routine acquire_read(rdwr $b$){
 acquire($b.l$);
 while($b.aw$+$b.ww$>0)
   wait($b.v_r$, $b.l$);
 $b.ar$ := $b.ar$+1;
 release($b.l$)}

routine release_read(rdwr $b$){
 acquire($b.l$);
 $b.ar$ := $b.ar$−1;
 notify($b.v_w$);
 release($b.l$)}

routine main(){
 $rw$ := new_rdwr();
 fork(
   while(1){
     acquire_read($rw$); /* reading ... */
     release_read($rw$)
   );
 while(1){
   acquire_write($rw$); /* writing ... */
   release_write($rw$)
 }}
```

Figure 18 Readers-writers locks synchronized by condition variables.

A solution to solve this problem is that when a writer releases the writing lock, if there is a waiting writer, in addition to notifying that waiting writer, it also increases the number of active writers. In other words, if there is a waiting writer in the system, the number of the active writers is increased in advance by the thread releasing the writing lock (or the reading lock) and not by the thread acquiring the writing lock. A fair implementation of the readers-writers lock (writers-preference) following this idea along with the details of the specifications of each routine is shown in Figure 19[6]. When a readers-writers lock b is created, a permission $\mathsf{rw}(b, O_w, O_r)$ is provided, where O_w and O_r are two bags of waitable objects whose levels are in an arbitrary client-specified range R. A thread can acquire a writing lock of b if the levels of objects in O_w are lower than the levels of all obligations of that thread. When that lock is acquired the objects in O_w are loaded onto the bag of the obligations of this thread, which are discharged when this thread releases this lock. Similar to a writing lock, a reading lock of b can be acquired if the levels of the objects in O_r are lower than the levels of the obligations of the reading thread. When this lock is acquired the objects in O_r are loaded which are discharged when this lock is released. Note that in this program one of the desired invariants is $b.ar + b.aw \leqslant Ot(b.v_w)$. Accordingly, since the variable tracking the number of active writers (aw) is increased in the thread notifying v_w and not in the notified one, it is necessary to load an obligation of v_w onto the notifying thread and then transfer this obligation to the notified thread, through the notification.

[6] We inserted assert(e) commands, shorthand for while($\neg e$){} (loop forever if e is false), to simplify the proof. They can be eliminated using ghost state [27]. The verification of this program without using the assert commands can be found in [17].

```
$\mathsf{rw}(\mathsf{rdwr}\ b, \mathsf{bag}\langle\mathsf{waitobj}\rangle\ O_w, \mathsf{bag}\langle\mathsf{waitobj}\rangle\ O_r) =$
 $\mathsf{lock}(b.l) * \mathsf{cond}(b.v_r, \mathsf{true}, \{\!\{\}\!\}) * \mathsf{cond}(b.v_w, \mathsf{true}, \{\!\{b.v_w\}\!\}) \land \mathsf{I}(ch.l){=}\mathsf{linv}(b) \land$
 $\mathsf{R}(b.l) < \mathsf{R}(b.v_w) < \mathsf{R}(b.v_r) \land O_w{=}\{\!\{b.v_r, b.v_w\}\!\} \land O_r{=}\{\!\{b.v_w\}\!\}$

$\mathsf{linv}(\mathsf{rdwr}\ b) =$
 $\lambda Wt.\ \lambda Ot.\ \exists ar, aw, ww.\ b.ar \mapsto ar * b.aw \mapsto aw * b.ww \mapsto ww \land \mathsf{L}(b.v_r){=}\mathsf{L}(b.v_w){=}b.l \land$
 $0 \leqslant ar \land 0 \leqslant aw \land 0 \leqslant ww \land Wt(b.v_w){=}ww \land$
 $ar + aw \leqslant Ot(b.v_w) \land (Wt(b.v_w) = 0 \lor 0 < ar + aw) \land$
 $aw + ww \leqslant Ot(b.v_r) \land (Wt(b.v_r) = 0 \lor 0 < aw + ww)$
```

```
routine new_rdwr()
req : $\{R \cong_< \mathbb{Q}\}$
ens : $\{\lambda b.\ \exists O_w, O_r.\ \mathsf{rw}(b, O_w, O_r) \land$
 $\mathsf{levels}(O_w) \subseteq R \land \mathsf{levels}(O_r) \subseteq R\}$
{$l$ := new_lock();
 $v_w$ := new_cvar; g_initc($v_w$);
 $v_r$ := new_cvar; g_initc($v_r$);
 $b$ := rdwr($l$:=$l$, $v_w$:=$v_w$, $v_r$:=$v_r$,
   $aw$:=new_int(0), $ww$:=new_int(0),
   $ar$:=new_int(0)); g_initl($l$);
 $b$}

routine acquire_write(rdwr $b$)
req : $\{\mathsf{obs}(O) * \mathsf{rw}(b, O_w, O_r) \land O_w \prec O\}$
ens : $\{\mathsf{obs}(O \uplus O_w) * \mathsf{rw}(b, O_w, O_r)\}$
{acquire($b.l$); g_chrg($b.v_r$);
 if($b.aw$+$b.ar$>0){
   $b.ww$ := $b.ww$+1;
   wait($b.v_w$, $b.l$)
 }
 else{ $b.aw$ := $b.aw$+1; g_chrg($b.v_w$)};
 release($b.l$)}

routine release_write(rdwr $b$)
req : $\{\mathsf{obs}(O \uplus O_w) * \mathsf{rw}(b, O_w, O_r) \land O_w \prec O\}$
ens : $\{\mathsf{obs}(O) * \mathsf{rw}(b, O_w, O_r)\}$
{acquire($b.l$);
 assert($0 < b.aw$);
 $b.aw$ := $b.aw$−1;
 if($b.ww > 0$){
   notify($b.v_w$);
   $b.ww$ := $b.ww$−1;
   $b.aw$ := $b.aw$+1
 }
 else{ notifyAll($b.v_r$); g_disch($b.v_w$)};
 g_disch($b.v_r$);
 release($b.l$)}
```

```
routine acquire_read(rdwr $b$)
req : $\{\mathsf{obs}(O) * \mathsf{rw}(b, O_w, O_r) \land O_r \prec O\}$
ens : $\{\mathsf{obs}(O \uplus O_r) * \mathsf{rw}(b, O_w, O_r)\}$
{acquire($b.l$);
 while($b.aw$+$b.ww$>0)
   wait($b.v_r$, $b.l$);
 $b.ar$ := $b.ar$+1; g_chrg($b.v_w$);
 release($b.l$)}

routine release_read(rdwr $b$)
req : $\{\mathsf{obs}(O \uplus O_r) * \mathsf{rw}(b, O_w, O_r) \land O_r \prec O\}$
ens : $\{\mathsf{obs}(O) * \mathsf{rw}(b, O_w, O_r)\}$
{acquire($b.l$);
 assert($0 < b.ar$);
 $b.ar$ := $b.ar$−1;
 if($b.ar = 0 \land b.ww > 0$){
   notify($b.v_w$);
   $b.ww$ := $b.ww$−1;
   $b.aw$ := $b.aw$+1
 } else g_disch($b.v_w$);
 release($b.l$)}

routine main()
req : $\{\mathsf{obs}(\{\!\{\}\!\})\}$
ens : $\{\mathsf{obs}(\{\!\{\}\!\})\}$
{$rw$ := new_rdwr();
 fork(
   while(1){
      acquire_read($rw$); /* reading ... */
      release_read($rw$)
   }
 );
while(1){
   acquire_write($rw$); /* writing ... */
   release_write($rw$)
 }
}
```

Figure 19 Fair readers-writers locks on top of fair monitors, where $R \cong_< \mathbb{Q}$ indicates that R is order-isomorphic to the rational numbers.

4 Related Work

Permissions

One common approach to prove safety properties of a program is assigning permissions to the threads of that program and to make sure that each thread only performs actions for which that thread has permissions [3]. This approach has been adopted by concurrent separation logic [37], where a thread can access a heap location only if it owns that location. This logic has been extended to handle dynamic thread creation [11, 19, 13], rely/guarantee [48, 8], reentrant locking [12], and channels [18, 40].

Jung et al. [27] proposed a concurrent separation logic, namely *Iris*, for reasoning about safety of concurrent programs, as the logic in logical relations, and to reason about type systems and data abstraction, among other things. In this logic user-defined protocols on shared state are expressed through *partial commutative monoids* and are enforced through *invariants*. However, the Iris program logic and many other logics such as [41, 36] only prove per-thread safety (i.e. no thread ever crashes): their adequacy theorems state that the program does not reach a state where some thread cannot make a step. This works because these logics do not consider blocking constructs, where a thread may legitimately be stuck temporarily. It follows that these logics do not support programs that use primitive blocking operations. Recently, an extension of Iris, namely *Iron* [1], exploits a notion of obligation to prove absence of resource leaks, but not deadlock-freedom. The adequacy theorem of this logic only considers the state reached by a program after it is completely finished (i.e. all threads have reduced to a value), and it proves that in that state all resources have been freed.

Deadlock

Several approaches to verify termination [35, 14, 43], total correctness [4], and lock-freedom [20] of concurrent programs have been proposed. These approaches are only applicable to non-blocking algorithms, where the suspension of one thread cannot lead to the suspension of other threads. Consequently, they cannot be used to verify deadlock-freedom of programs using condition variables or channels, where the suspension of a notifying/sending thread might cause a waiting thread to be infinitely blocked. In [39] a compositional approach to verify termination of multi-threaded programs is introduced, where *rely-guarantee reasoning* is used to reason about each thread individually while there are some assertions about other threads. In this approach a program is considered to be terminating if it does not have any infinite computations. As a consequence, it is not applicable to programs using condition variables because a waiting thread that is never notified cannot be considered as a terminating thread. There are some other works on verifying deadlock-freedom and starvation-freedom of concurrent objects with partial methods, which do not return under certain circumstances such as acquiring a held lock [34]. In addition to locks these approaches allows to verify other concurrent objects such as sets, stacks and queues. However, this approach is not applicable to condition variables because of *lost notifications*, i.e. a notification on a condition variable is lost if no thread is waiting for that condition variable. Note that releasing a lock, pushing an item into stack, and enqueueing an item when there is no thread waiting for the related concurrent object is not lost, since the next thread, which tries to acquire/pop/dequeue the concurrent object, will not be blocked.

There are also some other approaches addressing some common synchronization bugs of programs in the presence of condition variables. In [49], for example, an approach to identify some potential problems of concurrent programs using condition variables is presented.

However, it does not take the order of execution of theses commands into account. In other words, it might accept an undesired execution trace where the waiting thread is scheduled after the notifying thread, that might lead the waiting thread to be infinitely suspended. [28] uses Petri nets to identify some common problems in multithreaded programs such as data races, lost signals, and deadlocks. However the model introduced for condition variables in this approach only covers the communication of two threads and it is not clear how it deals with programs having more than two threads communicating through condition variables. Recently, [6, 9] have introduced an approach ensuring that every thread synchronizing under a set of condition variables eventually exits the synchronization block if that thread eventually reaches that block. This approach succeeds in verifying one of the applications of condition variables, namely the buffer. However, since this approach is not modular and relies on a Petri net analysis tool to solve the termination problem, it suffers from a long verification time when the size of the state space is increased, such that the verification of a buffer application having 20 producer and 18 consumer threads, for example, takes more than two minutes.

There are several verification techniques and type systems to check deadlock-freedom of programs that either synchronize via locks [10, 32, 47] or communicate via messages [7, 30]. Kobayashi [30, 29] proposed a type system for deadlock-free processes, ensuring that a well-typed process that is annotated with a finite capability level is deadlock-free. He extended channel types with the notion of usages, describing how often and in which order a channel is used for input and output. In his type system, which works in the context of π-calculus, the send operation is synchronous; i.e. the thread sending on a channel is suspended until the sent message is received by another thread. However, this approach and other approaches, such as [31, 5, 38], verifying deadlock-freedom in the context of π-calculus are not straight forwardly applicable to imperative programming languages.

Obligations

Inspired by the notion of capabilities [30, 29] and implicit dynamic frames [46, 44, 45], Leino et al. [33] later integrated deadlock prevention into a verification system for an object-oriented and imperative programing language. In this approach each thread trying to receive a message from a channel must spend one credit for that channel, where a credit for a channel is obtained if a thread is obliged to discharge an obligation for that channel. A thread can discharge an obligation for a channel if it either sends a message on that channel or delegates that obligation to another thread. This approach supports asynchronous send operations, where sending on a channel does not suspend the sender thread and there might be a state where a message is sent but not received by any thread. The notion of obligations is used in other verification approaches, which verify deadlock-freedom of semaphores [21], monitors [15, 16] and channels in a separation logic-based system [23, 24], and finite blocking in non-terminating programs [2]. However, these approaches allow obligations to be transferred only when a thread is forked. In other words, unlike permissions which can be transfered through synchronizations, transferring obligations through synchronizations in these approaches is forbidden. In this paper we provide two mechanisms that allow these approaches to transfer obligations, along with permissions, through synchronization. Our approach can be used to verify deadlock-freedom of imperative programs where some obligation must be transferred through notifications and channels (with asynchronous send operations).

5 Conclusion

This paper introduces two techniques to transfer obligations through synchronization, while ensuring that there is no state where the transferred obligations are lost, i.e. where they are discharged from the sender thread and not loaded onto the receiver thread yet. These techniques allow the obligation-based verification approaches, which modularly verify deadlock-freedom and liveness properties of programs, to transfer obligations, along with permissions, between threads, enabling them to verify a wider range of interesting programs, where obligations must be transferred through synchronizations. We encoded the proposed proof rules in the VeriFast program verifier and succeeded in verifying deadlock-freedom of a number of interesting programs, such as some variations of client-server programs, fair readers-writers locks and dining philosophers, which cannot be modularly verified without such transfer. Integrating the two mechanisms introduced in this paper is an area of future work. Additionally, designing a new variant of Iris for programs with primitive blocking constructs on top of which our presented approach can be built is another important area of future work.

References

1 Aleš Bizjak, Daniel Gratzer, Robbert Krebbers, and Lars Birkedal. Iron: Managing Obligations in Higher-Order Concurrent Separation Logic. *To appear in POPL 2019: ACM SIGPLAN Symposium on Principles of Programming Languages, Lissabon, Portugal*, 2019.

2 Pontus Boström and Peter Müller. *Modular verification of finite blocking in non-terminating programs*, volume 37. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

3 John Boyland. Checking interference with fractional permissions. In *International Static Analysis Symposium*, pages 55–72. Springer, 2003.

4 Pedro da Rocha Pinto, Thomas Dinsdale-Young, Philippa Gardner, and Julian Sutherland. Modular Termination Verification for Non-blocking Concurrency. In *ESOP*, pages 176–201, 2016.

5 Ornela Dardha and Simon J Gay. A new linear logic for deadlock-free session-typed processes. In *International Conference on Foundations of Software Science and Computation Structures*, pages 91–109. Springer, 2018.

6 Pedro de Carvalho Gomes, Dilian Gurov, and Marieke Huisman. Specification and Verification of Synchronization with Condition Variables. In *International Workshop on Formal Techniques for Safety-Critical Systems*, pages 3–19. Springer, 2016.

7 Mariangiola Dezani-Ciancaglini, Dimitris Mostrous, Nobuko Yoshida, and Sophia Drossopoulou. Session types for object-oriented languages. In *European Conference on Object-Oriented Programming*, pages 328–352. Springer, 2006.

8 Xinyu Feng. Local rely-guarantee reasoning. In *ACM SIGPLAN Notices*, volume 44, pages 315–327. ACM, 2009.

9 Pedro de C Gomes, Dilian Gurov, Marieke Huisman, and Cyrille Artho. Specification and verification of synchronization with condition variables. *Science of Computer Programming*, 163:174–189, 2018.

10 Colin S Gordon, Michael D Ernst, and Dan Grossman. Static lock capabilities for deadlock freedom. In *Proceedings of the 8th ACM SIGPLAN workshop on Types in language design and implementation*, pages 67–78. ACM, 2012.

11 Alexey Gotsman, Josh Berdine, Byron Cook, Noam Rinetzky, and Mooly Sagiv. Local reasoning for storable locks and threads. In *Asian Symposium on Programming Languages And Systems*, pages 19–37. Springer, 2007.

12 Christian Haack, Marieke Huisman, and Clément Hurlin. Reasoning about Java's reentrant locks. In *Asian Symposium on Programming Languages And Systems*, pages 171–187. Springer, 2008.

13 Christian Haack and Clément Hurlin. Separation logic contracts for a Java-like language with fork/join. In *International Conference on Algebraic Methodology and Software Technology*, pages 199–215. Springer, 2008.

14 Jafar Hamin and Bart Jacobs. Modular verification of termination and execution time bounds using separation logic. In *Information Reuse and Integration (IRI), 2016 IEEE 17th International Conference on*, pages 110–117. IEEE, 2016.

15 Jafar Hamin and Bart Jacobs. Deadlock-Free Monitors. In *European Symposium on Programming*, pages 415–441. Springer, 2018.

16 Jafar Hamin and Bart Jacobs. Deadlock-free monitors: extended version. *TR CW712, Department of Computer Science, KU Leuven, Belgium. Full version available at* *http://www.cs.kuleuven.be/publicaties/rapporten/cw/CW712.abs.html*, 2018.

17 Jafar Hamin and Bart Jacobs. Deadlock-Free Monitors and Channels. Zenodo, http://doi.org/10.5281/zenodo.3241454, 2019. doi:10.5281/zenodo.3241454.

18 Tony Hoare and Peter O'Hearn. Separation logic semantics for communicating processes. *Electronic Notes in Theoretical Computer Science*, 212:3–25, 2008.

19 Aquinas Hobor, Andrew W Appel, and Francesco Zappa Nardelli. Oracle semantics for concurrent separation logic. In *European Symposium on Programming*, pages 353–367. Springer, 2008.

20 Jan Hoffmann, Michael Marmar, and Zhong Shao. Quantitative reasoning for proving lock-freedom. In *Logic in Computer Science (LICS), 2013 28th Annual IEEE/ACM Symposium on*, pages 124–133. IEEE, 2013.

21 Bart Jacobs. Provably live exception handling. In *Proceedings of the 17th Workshop on Formal Techniques for Java-like Programs*, page 7. ACM, 2015.

22 Bart Jacobs. VeriFast 18.02. Zenodo, http://doi.org/10.5281/zenodo.1182724, 2018. doi:10.5281/zenodo.1182724.

23 Bart Jacobs, Dragan Bosnacki, and Ruurd Kuiper. Modular termination verification. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 37. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

24 Bart Jacobs, Dragan Bosnacki, and Ruurd Kuiper. Modular termination verification of single-threaded and multithreaded programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 40(3):12, 2018.

25 Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. *NASA Formal Methods*, 6617:41–55, 2011.

26 Bart Jacobs, Jan Smans, and Frank Piessens. A quick tour of the VeriFast program verifier. *Programming Languages and Systems*, pages 304–311, 2010.

27 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. *ACM SIGPLAN Notices*, 50(1):637–650, 2015.

28 Krishna M Kavi, Alireza Moshtaghi, and Deng-Jyi Chen. Modeling multithreaded applications using Petri nets. *International Journal of Parallel Programming*, 30(5):353–371, 2002.

29 Naoki Kobayashi. A type system for lock-free processes. *Information and Computation*, 177(2):122–159, 2002.

30 Naoki Kobayashi. A new type system for deadlock-free processes. In *International Conference on Concurrency Theory*, pages 233–247. Springer, 2006.

31 Naoki Kobayashi and Cosimo Laneve. Deadlock analysis of unbounded process networks. *Information and Computation*, 252:48–70, 2017.

32 Duy-Khanh Le, Wei-Ngan Chin, and Yong-Meng Teo. An expressive framework for verifying deadlock freedom. In *International Symposium on Automated Technology for Verification and Analysis*, pages 287–302. Springer, 2013.

33 K Rustan M Leino, Peter Müller, and Jan Smans. Deadlock-free channels and locks. In *European Symposium on Programming*, pages 407–426. Springer, 2010.

34 Hongjin Liang and Xinyu Feng. Progress of concurrent objects with partial methods. *Proceedings of the ACM on Programming Languages*, 2(POPL):20, 2017.

35 Hongjin Liang, Xinyu Feng, and Zhong Shao. Compositional verification of termination-preserving refinement of concurrent programs. In *Proceedings of the Joint Meeting of the*

Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), page 65. ACM, 2014.

36 Aleksandar Nanevski, Ruy Ley-Wild, Ilya Sergey, and Germán Andrés Delbianco. Communicating state transition systems for fine-grained concurrent resources. In *ESOP*, volume 8410 of *Lecture Notes in Computer Science*, pages 290–310. Springer, 2014.

37 Peter W O'Hearn. Resources, concurrency, and local reasoning. *Theoretical computer science*, 375(1-3):271–307, 2007.

38 Luca Padovani. Type-Based Analysis of Linear Communications. *Behavioural Types: from Theory to Tools*, page 193, 2017.

39 Corneliu Popeea and Andrey Rybalchenko. Compositional Termination Proofs for Multi-threaded Programs. In *TACAS*, volume 12, pages 237–251. Springer, 2012.

40 David Pym and Chris Tofts. A calculus and logic of resources and processes. *Formal Aspects of Computing*, 18(4):495–517, 2006.

41 Azalea Raad, Jules Villard, and Philippa Gardner. Colosl: Concurrent local subjective logic. In *European Symposium on Programming Languages and Systems*, pages 710–735. Springer, 2015.

42 John 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. IEEE, 2002.

43 Reuben NS Rowe and James Brotherston. Automatic cyclic termination proofs for recursive procedures in separation logic. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs*, pages 53–65. ACM, 2017.

44 Jan Smans, Bart Jacobs, and Frank Piessens. Implicit dynamic frames. In *Proceedings of the 10th ECOOP Workshop on Formal Techniques for Java-like Programs*, pages 1–12, 2008.

45 Jan Smans, Bart Jacobs, and Frank Piessens. Implicit dynamic frames: Combining dynamic frames and separation logic. In *European Conference on Object-Oriented Programming*, pages 148–172. Springer, 2009.

46 Jan Smans, Bart Jacobs, and Frank Piessens. Implicit dynamic frames. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 34(1):2, 2012.

47 Kohei Suenaga. Type-based deadlock-freedom verification for non-block-structured lock primitives and mutable references. In *Asian Symposium on Programming Languages and Systems*, pages 155–170. Springer, 2008.

48 Viktor Vafeiadis and Matthew Parkinson. A marriage of rely/guarantee and separation logic. In *International Conference on Concurrency Theory*, pages 256–271. Springer, 2007.

49 Chao Wang and Kevin Hoang. Precisely Deciding Control State Reachability in Concurrent Traces with Limited Observability. In *VMCAI*, pages 376–394. Springer, 2014.

A Proof of Conditional Server Channels

In this section the verification of the program shown in Figure 11 is illustrated in Figures 20 and 21.

B Proof of Dining Philosophers

Transferring obligations through notifications allows to verify some other interesting programs such as some variants of *dining philosophers*, where a number of philosophers sit at a round table, think, get hungry, and eat if their neighbors are not eating (due to the limited number of forks). An implementation of this program is shown in Figures 22[7], and 23, where a

[7] For simplicity, in the proof of this program we avoid writing the heap ownership permissions

```
Mch′ ::= $\lambda m.$ channel(snd($m$), Mch, M′ch) ∗ trandit(snd($m$)) ∧ R(snd($m$))=1 ∧
 $\mathsf{M^r}$(snd($m$))=⦃1⦄
M′ch′ ::= $\lambda m.$ ⦃snd($m$)⦄

Mch ::= $\lambda m.$ channel(snd($m$), Mch′, M′ch′) ∗ (fst($m$)=done ? true : trandit(snd($m$))) ∧
   $\mathsf{M^r}$(snd($m$))=⦃1⦄ ∧ ¬S(snd($m$))
M′ch ::= $\lambda m.$ fst($m$)=done ? ⦃⦄ : ⦃snd($m$)⦄

Ms ::= $\lambda m.$ channel(snd($m$), Mch′, M′ch′) ∗ trandit(snd($m$))∧$\mathsf{M^r}$(snd($m$))=⦃1⦄∧¬S(snd($m$))
M′s ::= $\lambda m.$ ⦃snd($m$)⦄

routine server(channel $s$){
req : {obs(⦃⦄, ⦃$s^\infty$⦄) ∗ channel($s$, Ms, M′s) ∧ $\mathsf{M^r}$($s$)=⦃1⦄ ∧ S($s$)}
 while(true){
 inv : {obs(⦃⦄, ⦃$s^\infty$⦄)}
   ($thr$, $ch'$) := receive($s$);
   {obs(⦃$ch'$⦄, ⦃$s^\infty$⦄) ∗ channel($ch'$, Mch′, M′ch′) ∗ trandit($ch'$) ∧ $\mathsf{M^r}$($ch'$)=⦃1⦄ ∧ ¬S($ch'$)}
   $ch$ := new_channel();
   {obs(⦃$ch'$⦄, ⦃$s^\infty$⦄) ∗ trandit($ch'$) ∗ channel($ch$, Mch, M′ch) ∧ R($ch$)=1 ∧ $\mathsf{M^r}$($ch$)=⦃1⦄ ∧
    ¬S($ch$)}
   g_credit($ch$); g_trandit($ch$);
   {obs(⦃$ch'$, $ch$⦄, ⦃$s^\infty$, $ch$⦄) ∗ trandit($ch'$) ∗ credit($ch$) ∗ trandit($ch$)}
   send($ch'$, (through, $ch$));
   {obs(⦃⦄, ⦃$s^\infty$, $ch$⦄) ∗ credit($ch$)}
   fork(cserver($ch$))
   {obs(⦃⦄, ⦃$s^\infty$⦄)} }
ens : {false}}

routine client(channel $s$){
req : {obs(⦃⦄, ⦃⦄) ∗ channel($s$, Ms, M′s) ∗ trandit($s$)}
 $ch'$ := new_channel();
 {obs(⦃⦄, ⦃⦄) ∗ trandit($s$) ∗ channel($ch'$, Mch′, M′ch′) ∧ R($ch'$)=1 ∧ $\mathsf{M^r}$($ch'$)=⦃1⦄ ∧ ¬S($ch'$)}
 g_credit($ch'$); g_trandit($ch'$);
 {obs(⦃$ch'$⦄, ⦃$ch'$⦄) ∗ trandit($s$) ∗ credit($ch'$) ∗ trandit($ch'$)}
 send($s$, (through, $ch'$));
 {obs(⦃⦄, ⦃$ch'$⦄) ∗ credit($ch'$)}
 ($thr$, $ch$) := receive($ch'$); if($thr$≠through) abort();
 {obs(⦃$ch$⦄, ⦃⦄) ∗ channel($ch$, Mch, M′ch) ∗ trandit($ch$) ∧ R($ch$)=1 ∧ $\mathsf{M^r}$($ch$)=⦃1⦄}
 g_credit($ch'$); g_trandit($ch'$);
 {obs(⦃$ch$, $ch'$⦄, ⦃$ch'$⦄) ∗ trandit($ch$) ∗ credit($ch'$) ∗ trandit($ch'$)}
 send($ch$, (request(), $ch'$));
 {obs(⦃⦄, ⦃$ch'$⦄) ∗ credit($ch'$)}
 ($res$, $ch_1$) := receive($ch'$); if($res$=through ∨ $ch$≠$ch_1$) abort();
 {obs(⦃$ch$⦄, ⦃⦄) ∗ trandit($ch$)}
 send($ch$, (done, $ch'$))
ens : {obs(⦃⦄, ⦃⦄)}}
```

Figure 20 Verification of the program in Figure 11 (part one of two).

```
routine main(){
req : $\{\mathsf{obs}(⦃⦄, ⦃⦄)\}$
 $s$ := new_channel();
 $\{\mathsf{obs}(⦃⦄, ⦃⦄) * \mathsf{channel}(s, \mathsf{Ms}, \mathsf{M's}) \wedge \mathsf{M^r}(s)=⦃1⦄ \wedge \mathsf{S}(s)\}$
 g_trandits($s$);
 $\{\mathsf{obs}(⦃⦄, ⦃s^\infty⦄) * \mathsf{trandit}^\infty(s)\}$
 fork(
   $\{\mathsf{obs}(⦃⦄, ⦃s^\infty⦄)\}$
   server($s$));
 $\{\mathsf{obs}(⦃⦄, ⦃⦄) * \mathsf{trandit}^\infty(s)\}$
 fork(client($s$));
 $\{\mathsf{obs}(⦃⦄, ⦃⦄) * \mathsf{trandit}^\infty(s)\}$
 client($s$)
ens : $\{\mathsf{obs}(⦃⦄, ⦃⦄)\}$}
```

Figure 21 Verification of the program in Figure 11 (part two of two).

dining_philosophers structure consists of *phs*, a circular doubly linked list of philosopher, *size*, the size of this list, and a lock (l), where a philosopher structure consists of *pre*, a pointer to the previous philosopher, *next*, a pointer to the next philosopher, a condition variable v, and a state which can be Thinking, Hungry, or Eating. Calling new_dining_philosophers(*size*) creates *size* philosophers which are initially thinking. Calling pickup(dp, i) makes the ith philosopher in dp start eating if he/she is already hungry and none of its neighbors are eating. Calling putdown(dp, i) makes the ith philosopher in dp stop eating if he/she is already eating, and also makes this philosopher's neighbors eat if they are hungry and their neighbors are not eating.

The specification of the routine new_dining_philosophers in this figure indicates that creating a number of dining philosophers dp produces a permission dp(dp, cvs), where cvs is a list of condition variables associated with philosophers whose levels are assigned arbitrarily. A philosopher can try to start eating only if the level of the CV associated with that philosopher is lower than the levels of the obligations of the running thread. When this philosopher starts eating the CV associated with his/her left and right neighbors are loaded onto the bag of the obligations of the running thread. These obligations are discharged when this philosopher stops eating.

One desired invariant in this program is that the number of obligations for a CV associated with a philosopher ph is greater than the number of his/her neighbors which are eating, that is $\mathsf{st}(pre.state) + \mathsf{st}(next.state) \leqslant Ot(ph.v)$, where $\mathsf{st}(\text{Eating})=1$ and $\mathsf{st}(\text{Thinking})=\mathsf{st}(\text{Hungry})=0$. Since the state of a hungry philosopher ph waiting in a suspended thread t is changed to Eating in a thread t' where its neighbor stops eating, an obligation of $ph.v$ must be loaded onto the bag of the obligations of t' and be transferred to t as t' notifies t. This requires transferring obligations through notifications which is possible using our proposed extension. Accordingly, as shown in Figure 22, this program can be verified if for any condition variable v of a philosopher ph, when v is notified the obligations $⦃ph.pre.v, ph.next.v⦄$ are transferred.

$\mathsf{dp}(\mathsf{dining\_philosophers}\ dp,\ \mathsf{list}\langle\mathsf{waitobj}\rangle\ cvs) =$
$\mathsf{lock}(dp.l) * \mathsf{phs\_cvs}(dp.phs, dp.phs.pre, cvs) \wedge$
$\mathsf{I}(dp.l)=\mathsf{linv}(dp) \wedge \mathsf{L}(dp.l)=\mathsf{new\_dp} \wedge \mathsf{size}(cvs)=dp.size \wedge \forall 0\leqslant i\leqslant size.\ \mathsf{R}(r_l) < \mathsf{R}(rs[i])$

$\mathsf{phs\_cvs}(ph, ph', cvs) =$
$ph=ph'\ ?\ cvs=[ph.v] :\ \exists cvs',\ cvs=[ph.v :: cvs'] * \mathsf{phs\_cvs}(ph.next, ph', cvs')$

$\mathsf{linv}(\mathsf{dp}\ dp) = \lambda Wt.\ \lambda Ot.\ \mathsf{philosophers}(dp.phs, dp.phs.pre, Wt, Ot, dp.l)$

$\mathsf{philosophers}(ph, ph', Wt, Ot, l) = \mathsf{philosopher}(ph, Wt, Ot, l, ph.pre, ph.next) *$
$ph=ph'\ ?\ \mathsf{true} : \mathsf{philosophers}(ph.next, ph', Wt, Ot, l)$

$\mathsf{philosopher}(ph, Wt, Ot, l, pre, next) = \mathsf{cond}(ph.v, \mathsf{true}, \{\!\{pre.v, next.v\}\!\}) \wedge$
$pre=ph.pre \wedge next=ph.next \wedge ph\neq pre \wedge ph\neq next \wedge pre\neq next \wedge$
$next.pre=ph \wedge pre.next=ph \wedge \mathsf{L}(ph.v)=l \wedge 0 \leqslant Wt(ph.v) \wedge Wt(ph.v) \leqslant 1 \wedge$
$(ph.state=\text{Hungry} \vee Wt(ph.v) \leqslant 0) \wedge (ph.state\neq\text{Hungry} \vee 0 < Wt(ph.v)) \wedge$
$(Wt(ph.v) \leqslant 0 \vee 0 < \mathsf{st}(pre.state) + \mathsf{st}(next.state)) \wedge$
$\mathsf{st}(pre.state) + \mathsf{st}(next.state) \leqslant Ot(ph.v)$

$\mathsf{st}(state) ::= state=\text{Eating}\ ?\ 1 : 0$

```
routine new_dining_philosophers(int $size$){
req : $\{\forall 0\leqslant i<size.\ r_l < rs[i] \wedge \mathsf{size}(rs)=size \wedge 2 < size\}$
ens : $\{\lambda dp.\ \mathsf{dp}(dp, cvs) \wedge \mathsf{size}(cvs)=size \wedge \mathsf{R}(dp.l)=r_l \wedge \forall 0\leqslant i<size.\ \mathsf{R}(cvs[i])=rs[i]\}$
 $phs$ := philosopher($state$:=new_cell(Thinking), $v$:=new_cvar,
   $pre$:=new_cell(null), $next$:=new_cell(null));
 $ph1$ := philosopher($state$:=new_cell(Thinking), $v$:=new_cvar,
   $pre$:=new_cell(null), $next$:=new_cell(null));
 $ph2$ := philosopher($state$:=new_cell(Thinking), $v$:=new_cvar,
   $pre$:=new_cell(null), $next$:=new_cell(null));
 $phs.pre$ := $ph2$; $phs.next$ := $ph1$;
 $ph1.pre$ := $phs$; $ph1.next$ := $ph2$;
 $ph2.pre$ := $ph1$; $phs.next$ := $phs$;
 $l$ := new_lock; $i$ := 3
 while($i$<$size$){
   $ph$:=philosopher($state$:=new_cell(Thinking), $v$:=new_cvar, $pre$:=$phs$, $next$:=$phs.next$);
   $phs.next.pre$ := $ph$;
   $phs.next$ = $ph$;
   $i$ := $i$+1
 };
 dining_philosophers($phs$:=$phs$, $l$:=$l$, $size$:=$size$)}
```

Figure 22 Dining philosophers (part one of two).

```
routine pickup(dining_philosophers $dp$, int $i$){
req : $\{\mathsf{obs}(O) * \mathsf{dp}(dp, cvs) \land cvs[i] \prec O \land 0 \leqslant i < \mathsf{size}(cvs)\}$
ens : $\{\mathsf{obs}(O \uplus \{\!\![cvs[(i{+}n{-}1)\%n], cvs[(i{+}1)\%n]]\!\!\}) * \mathsf{dp}(dp, cvs) \land n{=}\mathsf{size}(cvs)\}$
 acquire($dp.l$);
 $ph$ := $dp.phs$;
 $j$ := new_int(0);
 while($j < i$){
   $ph$ := $ph.next$;
   $j$ := $j{+}1$
 };
 if($ph.state \neq$ Thinking)
   abort;
 $ph.state$ := Hungry;
 if($ph.next.state{=}$Eating $\lor$ $ph.pre.state{=}$Eating)
   wait($ph.v, dp.l$)
 else
   $ph.state$ := Eating;
 release($dp.l$)}

routine putdown(dining_philosophers $dp$, int $i$){
req : $\{\mathsf{obs}(O) * \mathsf{dp}(dp, cvs) \land 0 \leqslant i < \mathsf{size}(cvs)\}$
ens : $\{\mathsf{obs}(O - \{\!\![cvs[(i{+}n{-}1)\%n], cvs[(i{+}1)\%n]]\!\!\}) * \mathsf{dp}(dp, cvs) \land n{=}\mathsf{size}(cvs)\}$
 acquire($dp.l$);
 $ph$ := $dp.phs$;
 $j$ := new_int(0);
 while($j < i$){
   $ph$ := $ph.next$;
   $j$ := $j{+}1$
 };
 if($ph.state \neq$ Eating)
   abort;
 $ph.state$ := Thinking;
 test_and_notify($ph.next$);
 test_and_notify($ph.pre$);
 release($dp.l$)}

routine test_and_notify(philosopher $ph$)
{if($ph.next.state{\neq}$Eating $\land$ $ph.pre.state{\neq}$Eating $\land$ $ph.state{=}$Hungry){
   $ph.state$ := Eating;
   notify($ph.v$)
 }}
```

Figure 23 Dining philosophers (part two of two).

$$
\begin{array}{l}
c \in \mathit{Commands},\ e \in \mathit{Expressions},\ z \in \mathbb{Z},\ b \in \mathit{Booleans},\ x \in \mathit{Variables} \\
e ::= z \mid x \mid e_1{+}e_2 \mid (e_1, e_2) \mid \mathsf{fst}(e) \mid \mathsf{snd}(e) \\
\qquad \mid \mathsf{true} \mid e_1 = e_2 \mid e_1 \leqslant e_2 \mid \neg e \\
c ::= \mathsf{val}(e) \mid \mathsf{let}(x, c_1, c_2) \mid \mathsf{fork}(c) \mid \mathsf{while}(c) \mid \mathsf{if}(e, c_1, c_2) \\
\qquad \mid \mathsf{new\_channel}(b) \mid \mathsf{send}(e_1, e_2) \mid \mathsf{receive}(e) \mid \mathsf{wait}(e) \mid \mathsf{nop}
\end{array}
$$

Figure 24 Syntax of the programming language.

C Transferring Obligations Through Channels: Soundness Proof

In this appendix we provide a formalization and soundness proof for the approach introduced in Section 2. However, unfortunately, there are a few technical differences between this formalization and the one proposed in Section 2 such that in this formalization the ghost information, such as level and transferred permissions and obligations, are associated with channel addresses via the channel permissions rather than via global functions[8]. The proof rules associated with this formalization and the verification of the program in Figure 3, proved using these rules, are shown in Sections C.4 and C.6, respectively.

C.1 Syntax and Semantics of Programs

The syntax of our programming language is defined in Figure 24, where $\mathsf{val}(e)$ is a command that simply yields the value of e as its result and has no side effects, $\mathsf{let}(x, c_1, c_2)$ is syntactic sugar for $x{:=}c_1; c_2$, $\mathsf{fork}(c)$ creates a thread executing c, $\mathsf{while}(c)$ keeps executing c while c evaluates to true, $\mathsf{if}(e, c_1, c_2)$ executes c_1 if e evaluates to true and otherwise it executes c_2, send and receive are used for sending and receiving on channels, and wait, which cannot be used by programmers, indicates that the related thread has tried to receive from an empty channel. Additionally, instead of defining three ghost commands g_credit, g_trandit, and g_trandits, we define a single ghost command nop which is inserted into the program for verification purposes and has no effect on the program's behavior. The expressions used in the syntax of programs can be evaluated and substituted as shown in Figure 25.

The small step semantics, defined in Figure 27, relates two *configurations*, defined in Figure 26. A configuration consists of a heap, which maps a channel identifier to the list of messages of that channel; a thread table, which maps a thread identifier to the pair command-context related to that thread; and a list of server channels.

C.2 Syntax and Semantics of Assertions

The syntax of assertions is defined in Figure 28[9]. Note that the *location* of a channel ch consists of the *obligation* of ch and the permissions and the obligations which are transferred through a specific message sent on ch, denoted by $\mathsf{M}(ch)$ and $\mathsf{M}'(ch)$. Also note that the

[8] The reason is to make this formalization consistent with the one in Appendix D which is machine-checked with Coq, where ghost information is associated with lock and condition variable addresses via the lock and cond permissions. However, we believe one way to formalize the precise approach of Section 3 would be to define assertions as functions from ghost information to separating conjunctions of chunks. In the soundness proof, one would track these as partial functions whose domain is the set of allocated addresses. The functions passed into the assertions would be *totalizations* of these partial functions. An assertion is true if it is true for all totalizations of the functions.

[9] Note that we use a shallow embedding: assertions have no variables; to model quantifications, we use meta-level functions from values to assertions.

$v \in \mathit{Values} ::= z \mid b \mid (v, v)$
$[\![]\!] \in \mathit{Expressions} \to \mathit{Values}$

$[\![z]\!]{=}z$
$[\![x]\!]{=}0$
$[\![e_1{+}e_2]\!]{=}[\![e_1]\!]{+}[\![e_2]\!]$
$[\![(e_1, e_2)]\!]{=}([\![e_1]\!], [\![e_2]\!])$
$[\![\mathsf{true}]\!]{=}\mathsf{true}$
$[\![(e_1 = e_2)]\!]{=}([\![e_1]\!] = [\![e_2]\!])$
$[\![(e_1 \leqslant e_2)]\!]{=}([\![e_1]\!] \leqslant [\![e_2]\!])$
$[\![(\neg e)]\!]{=}(\neg[\![e]\!])$

$$[\![\mathsf{fst}(e)]\!] = \begin{cases} [\![e_1]\!] & \text{if } e{=}(e_1, e_2) \\ 0 & \text{otherwise} \end{cases}$$

$$[\![\mathsf{snd}(e)]\!] = \begin{cases} [\![e_2]\!] & \text{if } e{=}(e_1, e_2) \\ 0 & \text{otherwise} \end{cases}$$

$z[v/x] = z$
$x[v/x'] = x \quad$ if $x \neq x'$
$x[v/x] = v$
$(e_1{+}e_2)[v/x] = e_1[v/x]{+}e_2[v/x]$
$(e_1, e_2)[v/x] = (e_1[v/x], e_2[v/x])$
$\mathsf{true}[v/x] = \mathsf{true}$
$(e_1 = e_2)[v/x] = (e_1[v/x] = e_2[v/x])$
$(e_1 \leqslant e_2)[v/x] = (e_1[v/x] \leqslant e_2[v/x])$
$(\neg e)[v/x] = (\neg e[v/x])$
$\mathsf{fst}(e)[v/x] = \mathsf{fst}(e[v/x])$
$\mathsf{snd}(e)[v/x] = \mathsf{snd}(e[v/x])$

$\mathsf{val}(e)[v/x] = \mathsf{val}(e[v/x])$
$\mathsf{let}(x, c_1, c_2)[v/x] = \mathsf{let}(x, c_1, c_2)$
$\mathsf{let}(x, c_1, c_2)[v/x'] = \mathsf{let}(x, c_1[v/x'], c_2[v/x']) \quad$ if $x \neq x'$
$\mathsf{fork}(c)[v/x] = \mathsf{fork}(c[v/x])$
$\mathsf{while}(c)[v/x] = \mathsf{while}(c[v/x])$
$\mathsf{if}(e, c_1, c_2)[v/x] = \mathsf{if}(e[v/x], c_1[v/x], c_2[v/x])$
$\mathsf{new\_channel}(b)[v/x] = \mathsf{new\_channel}(b)$
$\mathsf{send}(e_1, e_2)[v/x] = \mathsf{send}(e_1[v/x], e_2[v/x])$
$\mathsf{receive}(e)[v/x] = \mathsf{receive}(e[v/x])$
$\mathsf{nop}[v/x] = \mathsf{nop}$

Figure 25 Evaluation of expressions and substitution of expressions and commands.

$$\begin{array}{l}
m \in \mathit{Messages} = \mathit{Values} \\
\mathit{Addresses} = \mathbb{Z} \\
\mathit{ThreadIds} = \mathbb{Z} \\
h \in \mathit{Heaps} = \mathit{Addresses} \rightharpoonup \mathit{Lists}(\mathit{Messages}) \\
\xi \in \mathit{Contexts} ::= \mathsf{done} \mid \mathsf{let}'(x, c, \xi) \mid \mathsf{while}'(c, \xi) \\
\theta \in \mathit{ThreadConfigurations} ::= (c; \xi) \\
t \in \mathit{ThreadTables} = \mathit{ThreadIds} \rightharpoonup \mathit{ThreadConfigurations} \\
s \in \mathit{ServerChannelsSets} = \mathit{Sets}(\mathit{Addresses}) \\
\kappa \in \mathit{Configurations} = \mathit{Heaps} \times \mathit{ThreadTables} \times \mathit{ServerChannelsSets}
\end{array}$$

Figure 26 Configurations.

$$\begin{array}{l}
(t[id{:=}\mathsf{new\_channel}(b); \xi], h[z{:=}\varnothing], s) \rightsquigarrow (t[id{:=}\mathsf{val}(z); \xi], h[z{:=}[]], b{=}\mathsf{true}\ ?\ s{\cup}\{z\} : s) \\
(t[id{:=}\mathsf{send}(e_1, e_2); \xi], h[\llbracket e_1 \rrbracket{:=}M], s) \rightsquigarrow (t[id{:=}\mathsf{tt}; \xi], h[\llbracket e_1 \rrbracket{:=}M{\cdot}[\llbracket e_2 \rrbracket]], s) \\
(t[id{:=}\mathsf{receive}(e); \xi], h[\llbracket e \rrbracket{:=}[m]{\cdot}M], s) \rightsquigarrow (t[id{:=}\mathsf{val}(m); \xi], h[\llbracket e \rrbracket{:=}M], s) \\
(t[id{:=}\mathsf{receive}(e); \xi], h[\llbracket e \rrbracket{:=}[]], s) \rightsquigarrow (t[id{:=}\mathsf{wait}(e); \xi], h[\llbracket e \rrbracket{:=}[]], s) \\
(t[id{:=}\mathsf{wait}(e); \xi], h[\llbracket e \rrbracket{:=}[m]{\cdot}M], s) \rightsquigarrow (t[id{:=}\mathsf{val}(m); \xi], h[\llbracket e \rrbracket{:=}M], s) \\
(t[id{:=}\mathsf{fork}(c); \xi, id'{:=}\varnothing], h, s) \rightsquigarrow (t[id{:=}\mathsf{tt}; \xi, id'{:=}c; \mathsf{done}], h, s) \\
(t[id{:=}\mathsf{let}(x, c_1, c_2); \xi], h, s) \rightsquigarrow (t[id{:=}c_1; \mathsf{let}'(x, c_2, \xi)], h, s) \\
(t[id{:=}\mathsf{val}(e); \mathsf{let}'(x, c, \xi)], h, s) \rightsquigarrow (t[id{:=}c[\llbracket e \rrbracket/x]; \xi], h, s) \\
(t[id{:=}\mathsf{val}(e); \mathsf{done}], h, s) \rightsquigarrow (t[id{:=}\varnothing], h, s) \\
(t[id{:=}\mathsf{if}(e, c_1, c_2); \xi], h, s) \rightsquigarrow (t[id{:=}c_1; \xi], h, s) \quad \text{if } \llbracket e \rrbracket = \mathsf{true} \\
(t[id{:=}\mathsf{if}(e, c_1, c_2); \xi], h, s) \rightsquigarrow (t[id{:=}c_2; \xi], h, s) \quad \text{if } \llbracket e \rrbracket \neq \mathsf{true} \\
(t[id{:=}\mathsf{while}(c); \xi], h, s) \rightsquigarrow (t[id{:=}c; \mathsf{while}'(c, \xi)], h, s) \\
(t[id{:=}\mathsf{val}(e); \mathsf{while}'(c, \xi)], h, s) \rightsquigarrow (t[id{:=}c; \mathsf{while}'(c, \xi)], h, s) \quad \text{if } \llbracket e \rrbracket = \mathsf{true} \\
(t[id{:=}\mathsf{val}(e); \mathsf{while}'(c, \xi)], h, s) \rightsquigarrow (t[id{:=}\mathsf{tt}; \xi], h, s) \quad \text{if } \llbracket e \rrbracket \neq \mathsf{true} \\
(t[id{:=}\mathsf{nop}; \xi], h, s) \rightsquigarrow (t[id{:=}\mathsf{tt}; \xi], h, s)
\end{array}$$

Figure 27 Semantics of programs, where tt stands for val(0), and $[m]$ represents a list with one element m, and $M_1{\cdot}M_2$ appends two lists M_1 and M_2.

$$
\begin{array}{l}
n \in \mathbb{N} \\
\mathbb{NF} ::= n \mid \infty \\
\mathit{Bags}(A) = A \to \mathbb{NF} \\
\mathit{Indexes} = \mathbb{Z} \\
\mathit{Arguments} = \mathbb{Z} \\
r \in \mathit{Levels} = \mathbb{R} \\
o \in \mathit{Obligations} = \mathit{Addresses} \times \mathit{Levels} \times \mathit{Bags}(\mathit{Levels}) \times \mathit{Booleans} \\
l \in \mathit{Locations} = \\
\quad \mathit{Obligations} \times (\mathit{Indexes} \times \mathit{Lists}(\mathit{Arguments})), \mathit{Messages} \to \mathit{Bags}(\mathit{Obligations}) \\
O, I \in \mathit{Bags}(\mathit{Obligations}) \\
b \in \mathit{Booleans} \\
\hat{v} \in \mathit{AValues} ::= z \mid r \mid b \mid l \mid o \mid O \\
\alpha \in \mathit{AValues} \to \mathit{Assertions} \\
a \in \mathit{Assertions} ::= \mathsf{channel}(l) \mid \mathsf{credit}(z) \mid \mathsf{trandit}(z) \mid \mathsf{trandit}^{\infty}(z) \mid \mathsf{obs}(O, I) \\
\qquad\qquad\qquad\quad \mid b \mid a_1 \wedge a_2 \mid a_1 \vee a_2 \mid a_1 * a_2 \mid a_1 \mathrel{-\!\!*} a_2 \mid \forall \alpha \mid \exists \alpha \\
\\
\mathsf{pt} : \mathit{PredicateTables} = \mathit{Indexes} \to \mathit{Lists}(\mathit{Arguments}) \to \mathit{Messages} \to \mathit{Assertions} \\
\mathsf{O} : \mathit{Locations} \to \mathit{Obligations} \text{ , where } \mathsf{O}((A, R, M^r, S), M, M') = (A, R, M^r, S) \\
\mathsf{A} : \mathit{Locations} \to \mathit{Addresses} \text{ , where } \mathsf{A}((A, R, M^r, S), M, M') = A \\
\mathsf{R} : \mathit{Locations} \to \mathit{Levels} \text{ , where } \mathsf{R}((A, R, M^r, S), M, M') = R \\
\mathsf{M^r} : \mathit{Locations} \to \mathit{Bags}(\mathit{Levels}) \text{ , where } \mathsf{M^r}((A, R, M^r, S), M, M') = M^r \\
\mathsf{S} : \mathit{Locations} \to \mathit{Booleans} \text{ , where } \mathsf{S}((A, R, M^r, S), M, M') = S \\
\mathsf{M} : \mathit{Locations} \to \mathit{Messages} \to \mathit{Assertions} \\
\quad \text{where } \mathsf{M}((A, R, M^r, S), (M_1, M_2), M') = \mathsf{pt}(M_1, M_2) \\
\mathsf{M}' : \mathit{Locations} \to \mathit{Messages} \to \mathit{Bags}(\mathit{Obligations}) \text{ , where } \mathsf{M}'((A, R, M^r, S), M, M') = M' \\
\mathsf{Ro} : \mathit{Obligations} \to \mathit{Levels} \text{ , where } \mathsf{Ro}(A, R, M^r, S) = R
\end{array}
$$

Figure 28 Syntax of assertions.

$$
\begin{array}{l}
p \in \mathit{PermissionHeaps} = \mathit{Locations} \rightharpoonup \{\mathsf{channel}\}\\
\mathit{Option}(A) ::= s \mid \varnothing \text{ , where } s \in A\\
\tilde{O}, \tilde{I} \in \mathit{Option}(\mathit{Bags}(\mathit{Obligations}))\\
C, T \in \mathit{Bags}(\mathit{Addresses})
\end{array}
$$

$$
\begin{array}{l}
p, \tilde{O}, \tilde{I}, C, T \models \mathsf{channel}(l) \quad \Leftrightarrow \quad p(l) = \mathsf{channel}\\
p, \tilde{O}, \tilde{I}, C, T \models \mathsf{credit}(z) \quad \Leftrightarrow \quad 0 < C(z)\\
p, \tilde{O}, \tilde{I}, C, T \models \mathsf{trandit}(z) \quad \Leftrightarrow \quad 0 < T(z)\\
p, \tilde{O}, \tilde{I}, C, T \models \mathsf{trandit}^{\infty}(z) \quad \Leftrightarrow \quad T(z) = \infty\\
p, \tilde{O}, \tilde{I}, C, T \models \mathsf{obs}(O, I) \quad \Leftrightarrow \quad \tilde{O} = O \wedge \tilde{I} = I\\
p, \tilde{O}, \tilde{I}, C, T \models b \quad \Leftrightarrow \quad b = \mathsf{true}\\
p, \tilde{O}, \tilde{I}, C, T \models a_1 \wedge a_2 \quad \Leftrightarrow \quad p, \tilde{O}, \tilde{I}, C, T \models a_1 \wedge p, \tilde{O}, \tilde{I}, C, T \models a_2\\
p, \tilde{O}, \tilde{I}, C, T \models a_1 \vee a_2 \quad \Leftrightarrow \quad p, \tilde{O}, \tilde{I}, C, T \models a_1 \vee p, \tilde{O}, \tilde{I}, C, T \models a_2\\
p, \tilde{O}, \tilde{I}, C, T \models a_1 * a_2 \quad \Leftrightarrow \quad \exists p_1, p_2, \tilde{O}_1, \tilde{O}_2, \tilde{I}_1, \tilde{I}_2, C_1, C_2, T_1, T_2.\\
\quad p{=}p_1 \uplus p_2 \wedge \tilde{O}{=}\tilde{O}_1 \tilde{\uplus} \tilde{O}_2 \wedge \tilde{I}{=}\tilde{I}_1 \tilde{\uplus} \tilde{I}_2 \wedge C{=}C_1 \uplus C_2 \wedge T{=}T_1 \uplus T_2 \wedge\\
\quad p_1, \tilde{O}_1, \tilde{I}_1, C_1, T_1 \models a_1 \wedge p_2, \tilde{O}_2, \tilde{I}_2, C_2, T_2 \models a_2\\
p, \tilde{O}, \tilde{I}, C, T \models a_1 \mathrel{-\!\!*} a_2 \quad \Leftrightarrow \quad \forall p_1, \tilde{O}_1, \tilde{I}_1, C_1, T_1.\ p_1, \tilde{O}_1, \tilde{I}_1, C_1, T_1 \models a_1 \wedge\\
\quad \tilde{O} \perp \tilde{O}_1 \wedge \tilde{I} \perp \tilde{I}_1 \Rightarrow (p \uplus p_1), (\tilde{O} \tilde{\uplus} \tilde{O}_1), (\tilde{I} \tilde{\uplus} \tilde{I}_1), (C \uplus C_1), (T \uplus T_1) \models a_2\\
p, \tilde{O}, \tilde{I}, C, T \models \forall \alpha \quad \Leftrightarrow \quad \forall \hat{v} {\in} \mathit{AValues}.\ p, \tilde{O}, \tilde{I}, C, T \models \alpha(\hat{v})\\
p, \tilde{O}, \tilde{I}, C, T \models \exists \alpha \quad \Leftrightarrow \quad \exists \hat{v} {\in} \mathit{AValues}.\ p, \tilde{O}, \tilde{I}, C, T \models \alpha(\hat{v})
\end{array}
$$

$$
a_1 \vdash a_2 \quad \Leftrightarrow \quad (\forall p, \tilde{O}, \tilde{I}, C, T.\ p, \tilde{O}, \tilde{I}, C, T \models a_1 \Rightarrow p, \tilde{O}, \tilde{I}, C, T \models a_2)
$$

where for any $p_1, p_2 \in A \rightharpoonup B$ and $\tilde{O}_1, \tilde{O}_2 \in \mathit{Option}(\mathit{Bags}(A))$ and $B_1, B_2 \in \mathit{Bags}(A)$

$$
\tilde{O}_1 \perp \tilde{O}_2 \quad \Leftrightarrow \quad \tilde{O}_1 = \varnothing \vee \tilde{O}_2 = \varnothing
$$

$$
p_1 \uplus p_2 = \lambda v. \begin{cases} p_1(v) & \text{if } p_2(v){=}\varnothing \\ p_2(v) & \text{otherwise} \end{cases} \qquad \tilde{O}_1 \tilde{\uplus} \tilde{O}_2 = \begin{cases} \tilde{O}_1 & \text{if } \tilde{O}_2{=}\varnothing \\ \tilde{O}_2 & \text{if } \tilde{O}_1{=}\varnothing \\ \text{undefined} & \text{otherwise} \end{cases}
$$

$$
B_1 \uplus B_2 = \lambda v. \begin{cases} \infty & \text{if } B_1(v){=}\infty \text{ or } B_2(v){=}\infty \\ B_1(v) + B_2(v) & \text{otherwise} \end{cases}
$$

Figure 29 Satisfaction relation.

permissions which are transferred through a specific message sent on ch, denoted by $\mathsf{M}(ch)$, are specified through an index (as well as the required arguments) pointing to a table in which each element is a function that given a list of arguments and a message returns an assertion. This makes it possible for the predicates specifying these permissions to recursively refer to themselves or to each other, as in Figure 20[10]. The obligation of a location ch, denoted by $\mathsf{O}(ch)$, consists of the address of that location, denoted by $\mathsf{A}(ch)$, as well as other related information such as the level of ch, denoted by $\mathsf{R}(ch)$; the bag of the levels of the obligations which are possibly imported by ch, denoted by $\mathsf{M}^{\mathsf{r}}(ch)$; and whether ch is a server channel or not, denoted by $\mathsf{S}(ch)$.

The proposed assertions describe some ghost resources, namely p, $\tilde{O}$, $\tilde{I}$, C, T, that keep track of allocated channels, and the current thread's obligations, importers, credits, and trandits, respectively, shown in Figure 29.

C.3 Weakest Precondition of Commands

The weakest precondition of a command c for $n{>}0$ steps w.r.t. a postcondition Q (with a given predicate table, specified by pt), denoted by $\mathsf{wp}_{n,pt}(c, Q)$ is defined in Figure 30. Note that $\mathsf{wp}(c, Q)_{0,pt} = \mathsf{true}$. Also note that for the sake of simplicity the index pt is elided. Having this definition, we define the weakest precondition of a context and the weakest precondition of a command-context as shown in Definitions 1 and 2. Having these definitions, we can prove some auxiliary lemmas, shown in Lemmas 4, 5, 6, and 7, which are used to prove Theorem 13.

▶ **Definition 1** (Weakest Precondition of a Context).

$$\mathsf{wpx}_n(\xi) = \begin{cases} \lambda\_.\ \mathsf{obs}(\{\!\!\}, \{\!\!\}) & \textit{if } \xi{=}\mathsf{done} \\ \lambda v.\ \mathsf{wp}_n(c[v/x], \mathsf{wpx}_n(\xi')) & \textit{if } \xi{=}\mathsf{let}'(x, c, \xi') \\ \lambda v.\ v \neq \mathsf{true}\ ?\ \mathsf{wp}_n(\mathsf{tt}, \mathsf{wpx}_n(\xi')) : \mathsf{wp}_n(c, \mathsf{wpx}_{n-1}(\xi)) & \textit{if } \xi{=}\mathsf{while}'(c, \xi') \end{cases}$$

▶ **Definition 2** (Weakest Precondition of a command-context).

$$\mathsf{wpcx}_n(c, \xi) = \mathsf{wp}_n(c, \mathsf{wpx}_n(\xi))$$

▶ **Lemma 3** (Weakening Postcondition).

$$\begin{aligned}&\forall n, c, Q, Q', p, \tilde{O}, \tilde{I}, C, T. \quad p, \tilde{O}, \tilde{I}, C, T \models \mathsf{wp}_n(c, Q) \land (\forall z.\ Q(z) \vdash Q'(z)) \Rightarrow \\ &\forall n' {\leqslant} n.\ p, \tilde{O}, \tilde{I}, C, T \models \mathsf{wp}_{n'}(c, Q')\end{aligned}$$

Proof. By induction on n and case analysis of c. ◀

▶ **Lemma 4** (Weakest Precondition of new_channel).

$$\begin{aligned}&\forall b, \xi, p, O, I, C, T. \\ &p, O, I, C, T \models \mathsf{wpcx}_n(\mathsf{new\_channel}(b), \xi) \Rightarrow \forall z.(\forall l.\ \mathsf{A}(l) = z \Rightarrow p(l) = \varnothing) \Rightarrow \\ &\quad \exists r, M^r, M, M', ch.\ p[ch{:=}\mathsf{channel}], O, I, C, T \models \mathsf{wpcx}_n(\mathsf{val}(z), \xi) \land \\ &\quad \mathsf{A}(ch) = z \land \mathsf{R}(ch){=}r \land \mathsf{M}^{\mathsf{r}}(ch){=}M^r \land \mathsf{M}(ch){=}M \land \mathsf{M}'(ch){=}M' \land \mathsf{S}(ch){=}b\end{aligned}$$

[10] An alternative approach is to use a step-indexed domain of assertions, as in Iris [27]. There, ▶ *Assertions* could be used instead of *Indexes* × *Lists*(*Arguments*), where ▶ is Iris's *guard* for *guarded recursive definitions.*

$$
\begin{array}{l}
\mathsf{wp} \in \mathit{WeakestPreconditions} = \\
\quad \mathit{Commands} \to (\mathit{Values} \to \mathit{Assertions}) \to \mathbb{N} \to \mathit{PredicateTables} \to \mathit{Assertions}
\end{array}
$$

$$
\begin{array}{l}
\mathsf{levels}(O) = \{\!\{\mathsf{Ro}(o) \mid o \in O\}\!\} \\
o \prec' R \Leftrightarrow \forall r{\in}R.\ \mathsf{Ro}(o) < r \\
o \prec O \Leftrightarrow o \prec' \mathsf{levels}(O) \\
o \prec^{r} I \Leftrightarrow \forall o'{\in}I.\ o{=}o' \vee o \prec' \mathsf{M^r}(o')
\end{array}
$$

$$
\begin{array}{l}
\mathsf{wp}_n(\mathsf{val}(e), Q) = Q(\llbracket e \rrbracket) \\
\mathsf{wp}_n(\mathsf{new\_channel}(b), Q) = \forall z.\ \exists r, M^r, M, M'.\ \mathsf{channel}((z, r, M^r, b), M, M') \mathrel{-\!\!*} Q(z) \\
\mathsf{wp}_n(\mathsf{send}(e_1, e_2)) = \exists O, I, ch, m.\ (\mathsf{obs}(O, I) * \mathsf{channel}(ch) * \mathsf{M}(ch)(m) * \\
\quad (\mathsf{M^r}(ch){=}\{\!\{\}\!\} \vee \mathsf{trandit}(\mathsf{A}(ch))) \wedge \mathsf{levels}(\mathsf{M}'(ch)(m)){\subseteq}\mathsf{M^r}(ch) \wedge \mathsf{A}(ch) = \llbracket e_1 \rrbracket \wedge m = \llbracket e_2 \rrbracket) \\
\quad * ((\mathsf{obs}(O{-}\{\!\{\mathsf{O}(ch)\}\!\}{-}\mathsf{M}'(ch)(m), I) * \mathsf{channel}(ch)) \mathrel{-\!\!*} Q(\mathsf{tt})) \\
\mathsf{wp}_n(\mathsf{receive}(e)) = \exists O, I, ch.\ (\mathsf{obs}(O, I) * \mathsf{channel}(ch) * ((\mathsf{S}(ch) \vee \mathsf{credit}(\mathsf{A}(ch))) \wedge \\
\quad \mathsf{O}(ch){\prec}O \wedge \mathsf{O}(ch){\prec^{r}}I \wedge (\neg\mathsf{S}(ch) \vee (O{=}\{\!\{\}\!\} \wedge \forall o{\in}I.\ o{=}\mathsf{O}(ch))) \wedge \mathsf{A}(ch) = \llbracket e \rrbracket)) * \\
\quad \forall m.\ ((\mathsf{obs}(O{\uplus}\mathsf{M}'(ch)(m), I{-}\{\!\{\mathsf{O}(ch)\}\!\}) * \mathsf{M}(ch)(m)) \mathrel{-\!\!*} Q(m)) \\
\mathsf{wp}_n(\mathsf{wait}(e)) = \mathsf{wp}_n(\mathsf{receive}(e)) \\
\mathsf{wp}_n(\mathsf{fork}(c), Q) = \exists O_1, O_2, I_1, I_2.\ \mathsf{obs}(O_1{\uplus}O_2, I_1{\uplus}I_2) * (\mathsf{obs}(O_1, I_1) \mathrel{-\!\!*} Q(\mathsf{tt})) * \\
\quad (\mathsf{obs}(O_2, I_2) \mathrel{-\!\!*} \mathsf{wp}_{n-1}(c, \lambda\_.\ \mathsf{obs}(\{\!\{\}\!\}, \{\!\{\}\!\}))) \\
\mathsf{wp}_n(\mathsf{if}(e, c_1, c_2), Q) = (\llbracket e \rrbracket = \mathsf{true})\ ?\ \mathsf{wp}_{n-1}(c_1, Q) : \mathsf{wp}_{n-1}(c_2, Q) \\
\mathsf{wp}_n(\mathsf{while}(c), Q) = \mathsf{wp}_{n-1}(c, (\lambda vl.\ vl{\neq}\mathsf{true}\ ?\ Q(\mathsf{tt}) : \mathsf{wp}_{n-1}(\mathsf{while}(c), Q))) \\
\mathsf{wp}_n(\mathsf{nop}) = \\
\boxed{\text{as } \mathsf{g\_credit}} \\
\quad (\exists O, I, ch.\ \mathsf{obs}(O, I) * \mathsf{channel}(ch) * \\
\qquad ((\mathsf{obs}(O{\uplus}\{\!\{\mathsf{O}(ch)\}\!\}, I) * \mathsf{channel}(ch) * \mathsf{credit}(\mathsf{A}(ch))) \mathrel{-\!\!*} Q(\mathsf{tt}))) \vee \\
\boxed{\text{as } \mathsf{g\_trandit}} \\
\quad (\exists O, I, ch.\ \mathsf{obs}(O, I) * \mathsf{channel}(ch) * \\
\qquad ((\mathsf{obs}(O, I{\uplus}\{\!\{\mathsf{O}(ch)\}\!\}) * \mathsf{channel}(ch) * \mathsf{trandit}(\mathsf{A}(ch))) \mathrel{-\!\!*} Q(\mathsf{tt}))) \vee \\
\boxed{\text{as } \mathsf{g\_trandits}} \\
\quad (\exists O, I, ch.\ \mathsf{obs}(O, I) * \mathsf{channel}(ch) * \\
\qquad ((\mathsf{obs}(O, I{\uplus}(\lambda o.\ o{=}\mathsf{O}(ch)\ ?\ \infty : 0)) * \mathsf{channel}(ch) * \mathsf{trandit}^{\infty}(\mathsf{A}(ch))) \mathrel{-\!\!*} Q(\mathsf{tt})))
\end{array}
$$

Figure 30 Weakest precondition, where tt stands for 0.

▶ **Lemma 5** (Weakest Precondition of send).

$$
\begin{array}{l}
\forall n, e_1, e_2, \xi, p, O, I, C, T. \quad p, O, I, C, T \models \mathsf{wpcx}_n(\mathsf{send}(e_1, e_2), \xi) \Rightarrow \\
\quad \exists p_1, p_2, C_1, C_2, T_1, T_2, T_e, ch, m.\ p = p_1 \uplus p_2 \land C = C_1 \uplus C_2 \land T = T_1 \uplus T_2 \land T_e = T_2(\llbracket e_1 \rrbracket) \\
\quad \land\ \mathsf{A}(ch) = \llbracket e_1 \rrbracket \land m = \llbracket e_2 \rrbracket \land (\mathsf{M}^{\mathsf{r}}(ch) = \{\!\!\{\}\!\!\} \lor 0 < T_e) \land \mathsf{levels}(\mathsf{M}'(ch)(m)) \subseteq \mathsf{M}^{\mathsf{r}}(ch) \land \\
\quad p_2(ch) = \mathsf{channel} \land \\
\quad p_1, \varnothing, \varnothing, C_1, T_1 \models \mathsf{M}(ch)(m) \land \\
\quad (\mathsf{M}^{\mathsf{r}}(ch) = \{\!\!\{\}\!\!\} \Rightarrow p_2, O - \{\!\!\{\mathsf{O}(ch)\}\!\!\} - \mathsf{M}'(ch)(m), I, C_2, T_2 \models \mathsf{wpcx}_n(\mathsf{tt}, \xi)) \land \\
\quad (\mathsf{M}^{\mathsf{r}}(ch) \neq \{\!\!\{\}\!\!\} \Rightarrow p_2, O - \{\!\!\{\mathsf{O}(ch)\}\!\!\} - \mathsf{M}'(ch)(m), I, C_2, T_2[\llbracket e_1 \rrbracket := T_e - 1] \models \mathsf{wpcx}_n(\mathsf{tt}, \xi))
\end{array}
$$

▶ **Lemma 6** (Weakest Precondition of receive).

$$
\begin{array}{l}
\forall n, e, \xi, p, O, I, C, T. \quad p, O, I, C, T \models \mathsf{wpcx}_n(\mathsf{receive}(e), \xi) \Rightarrow \\
\quad \exists ch.\ p(ch) = \mathsf{channel} \land (S(ch) = \mathsf{true} \lor 0 < C(\llbracket e \rrbracket)) \land \mathsf{O}(ch) \prec O \land \mathsf{O}(ch) \prec^r I \land \\
\quad (\neg \mathsf{S}(ch) \lor (O = \{\!\!\{\}\!\!\} \land \forall o \in I.\ o = \mathsf{O}(ch))) \land \mathsf{A}(e) = \llbracket ch \rrbracket \land \\
\quad \forall m, p_1, C_1, T_1.\ p_1, \varnothing, \varnothing, C_1, T_1 \models \mathsf{M}(ch)(m) \Rightarrow \\
\qquad p \uplus p_1, O \uplus \mathsf{M}'(ch)(m), I - \{\!\!\{\mathsf{O}(ch)\}\!\!\}, C[\llbracket e \rrbracket := C(\llbracket e \rrbracket) - 1] \uplus C_1, T \uplus T_1 \models \mathsf{wpcx}_n(\mathsf{val}(m), \xi)
\end{array}
$$

▶ **Lemma 7** (Weakest Precondition of fork).

$$
\begin{array}{l}
\forall n, c, \xi, p, O, I, C, T. \quad p, O, I, C, T \models \mathsf{wpcx}_n(\mathsf{fork}(c), \xi) \Rightarrow \\
\exists p_1, p_2, O_1, O_2, I_1, I_2, C_1, C_2, T_1, T_2. \\
\quad p = p_1 \uplus p_2 \land O = O_1 \uplus O_2 \land I = I_1 \uplus I_2 \land C = C_1 \uplus C_2 \land T = T_1 \uplus T_2 \land \\
\quad p_1, O_1, I_1, C_1, T_1 \models \mathsf{wpcx}_n(\mathsf{tt}, \xi) \land \\
\quad p_2, O_2, I_2, C_2, T_2 \models \mathsf{wp}_{n-1}(c, \lambda\_.\mathsf{obs}(\{\!\!\{\}\!\!\}, \{\!\!\{\}\!\!\}))
\end{array}
$$

C.4 Correctness of Commands

We define *correctness of commands*, as shown in Definition 8, ensuring that each proposed proof rule, where $\mathsf{correct}_{pt}(P, c, Q)$ is abbreviated as $\{P\}\ c\ \{Q\}$, respects the definition of the weakest precondition. Having this definition we prove the proposed proof rules, ensuring deadlock freedom of importer channels, as well as some other necessary proof rules shown in Theorems 9, 10, and 11.

▶ **Definition 8** (Correctness of Commands). *A command is correct w.r.t a precondition P and a postcondition Q if and only if P implies the weakest precondition of that command w.r.t Q.*

$$\mathsf{correct}_{pt}(P, c, Q) \Leftrightarrow \forall n.\ P \Rightarrow \mathsf{wp}_{n,pt}(c, Q)$$

▶ **Theorem 9** (Rule Sequential Composition).

$$\mathsf{correct}(P, c_1, Q) \land (\forall z.\ \mathsf{correct}(Q(z), c_2[z/x], R)) \Rightarrow \mathsf{correct}(P, \mathsf{let}(x, c_1, c_2), R)$$

▶ **Theorem 10** (Rule Consequence).

$$\mathsf{correct}(P, c, Q) \land (P' \vdash P) \land (\forall z.\ Q(z) \vdash Q'(z)) \Rightarrow \mathsf{correct}(P', c, Q')$$

▶ **Theorem 11** (Rule Frame).

$$\mathsf{correct}(P, c, Q) \Rightarrow \mathsf{correct}(P * F, c, \lambda z.\ Q(z) * F)$$

As previously mentioned, since in this formalization ghost information is associated with channel addresses via the channel permissions rather than via global functions, we provide a new version of the proof rules, proposed in Section 2, associated with this formalization as shown in Figure 31.

NewChannel

$$\{\mathsf{true}\}\ \mathsf{new\_channel}\ \{\lambda a.\ \mathsf{channel}((a, r, R, b), (M_{index}, M_{args}), M')\}$$

Send

$$\{\mathsf{obs}(O, I) * \mathsf{channel}(ch) * \mathsf{M}(ch)(m) * (\mathsf{M}^{\mathsf{r}}(ch){=}\lBrace\rBrace \vee \mathsf{trandit}(a)) \wedge$$
$$\mathsf{levels}(\mathsf{M}'(ch)(m)) \subseteq \mathsf{M}^{\mathsf{r}}(ch) \wedge \mathsf{A}(ch) = a\}\ \mathsf{send}(a, m)$$
$$\{\lambda\_.\ \mathsf{obs}(O{-}\lBrace\mathsf{O}(ch)\rBrace{-}\mathsf{M}'(ch)(m), I) * \mathsf{channel}(ch)\}$$

Receive

$$\{\mathsf{obs}(O, I) * \mathsf{channel}(ch) * (\mathsf{S}(ch) \vee \mathsf{credit}(a)) \wedge \mathsf{O}(ch) \prec O \wedge \mathsf{O}(ch) \prec^{r} I \wedge$$
$$(\neg\mathsf{S}(ch) \vee (O{=}\lBrace\rBrace \wedge \forall o{\in}I.\ o{=}\mathsf{O}(ch))) \wedge \mathsf{A}(ch) = a\}\ \mathsf{receive}(a)$$
$$\{\lambda m.\ \mathsf{obs}(O{\uplus}\mathsf{M}'(ch)(m), I{-}\lBrace\mathsf{O}(ch)\rBrace) * \mathsf{channel}(ch) * \mathsf{M}(ch)(m)\}$$

Credit

$$\{\mathsf{obs}(O) * \mathsf{channel}(ch)\}\ \mathsf{nop}\ \{\lambda\_.\ \mathsf{obs}(O{\uplus}\lBrace\mathsf{O}(ch)\rBrace) * \mathsf{channel}(ch) * \mathsf{credit}(\mathsf{A}(ch))\}$$

Trandit

$$\{\mathsf{obs}(O, I) * \mathsf{channel}(ch)\}\ \mathsf{nop}\ \{\lambda\_.\ \mathsf{obs}(O, I{\uplus}\lBrace\mathsf{O}(ch)\rBrace) * \mathsf{channel}(ch) * \mathsf{trandit}(\mathsf{A}(ch))\}$$

Trandits

$$\{\mathsf{obs}(O, I) * \mathsf{channel}(ch)\}\ \mathsf{nop}\ \{\lambda\_.\ \mathsf{obs}(O, I{\uplus}\lBrace\mathsf{O}(ch)^{\infty}\rBrace) * \mathsf{channel}(ch) * \mathsf{trandit}^{\infty}(\mathsf{A}(ch))\}$$

Figure 31 The proof rules ensuring deadlock-freedom of importer channels, where ghost information is associated with channel addresses via the channel permissions.

C.5 Validity of a Configuration

We define *validity of a configuration*, shown in Definition 12, and prove that 1) starting from a valid configuration, all the subsequent configurations of the execution are also valid (Theorem 13), 2) a valid configuration is not deadlocked (Theorem 14), and 3) if a program c is verified by the proposed proof rules, where the verification starts from empty bags of obligations and importers and ends with such bags too, then the initial configuration, where the heap is empty, denoted by $\mathbf{0}{=}\lambda\_.\varnothing$, and there is only one thread with the command c (and a context done), and the list of server channels is empty, is a valid configuration (Theorem 16).

▶ **Definition 12** (Validity of a Configuration). *A configuration (t, h, s) is valid for n steps, denoted by* $\mathsf{valid}_n(t, h, s)$*, if there exists a set of* augmented threads *A, consisting of the identifier (id), the program (c), the context (ξ), the permission heap (p), the obligations (O), the importers (I), the credits (C), and the trandits (T) associated with each thread such that all of the following conditions hold:*

1. $\forall id, c, \xi.\ t(id) = (c; \xi) \Leftrightarrow \exists p, O, I, C, T.\ (id, c, \xi, p, O, I, C, T) \in A$
2. $\forall (id_1, c_1, \xi_1, p_1, O_1, I_1, C_1, T_1) \in A, (id_2, c_2, \xi_2, p_2, O_2, I_2, C_2, T_2) \in A.\ id_1 = id_2 \Rightarrow$ $(id_1, c_1, \xi_1, p_1, O_1, I_1, C_1, T_1) = (id_2, c_2, \xi_2, p_2, O_2, I_2, C_2, T_2)$
3. $\forall l_1, l_2.\ \mathsf{Pt}(l_1) \neq \varnothing \wedge \mathsf{Pt}(l_2) \neq \varnothing \wedge \mathsf{A}(l_1) = \mathsf{A}(l_1) \Rightarrow l_1 = l_2$
4. $\forall l.\ \mathsf{Pt}(l) \neq \varnothing \Rightarrow h(\mathsf{A}(l)) \neq \varnothing$ *and* $\forall z.\ (\forall l.\ \mathsf{A}(l) = z \Rightarrow \mathsf{Pt}(l) = \varnothing) \Rightarrow h(z) = \varnothing$
5. $\forall ch.\ \mathsf{Pt}(ch){=}\mathsf{channel} \Rightarrow$
   a. $\mathsf{M}^{\mathsf{r}}(ch) \neq \lBrace\rBrace \Rightarrow \mathsf{Tt}(\mathsf{A}(ch)) + \mathsf{size}_h(ch) \leqslant \mathsf{It}(ch) \wedge$
   b. $\forall m \in \mathsf{queue}_h(ch).\ \mathsf{levels}(\mathsf{M}'(ch)(m)) \subseteq \mathsf{M}^{\mathsf{r}}(ch) \wedge$

c. $\mathsf{S}(ch) = \mathsf{false} \Rightarrow$
$\mathsf{Ct}(\mathsf{A}(ch)) \leqslant \mathsf{Ot}(ch) + \mathsf{size}_h(ch) + \sum_{ch' \text{ where } \mathsf{Pt}(ch')=\mathsf{channel}} \sum_{m \in \mathsf{queue}_h(ch')} \mathsf{M}'(ch')(m)(ch)$

6. $\forall (id, c, \xi, p, O, I, C, T) \in A.\ p, O, I, C, T \models \mathsf{wpcx}_n(c, \xi)$

7. $\forall z \in s.\ \exists ch.\ \mathsf{Pt}(ch) = \mathsf{channel} \wedge \mathsf{S}(ch) = \mathsf{true} \wedge \mathsf{A}(ch) = z$

8. $\forall ch.\ \mathsf{Pt}(ch) = \mathsf{channel} \wedge \mathsf{S}(ch) = \mathsf{true} \Rightarrow \mathsf{A}(ch) \in s$

where

- $\mathsf{size}_h(ch)$ *returns the number of the messages in the channel ch, i.e.* $|h(\mathsf{A}(ch))|$
- $\mathsf{queue}_h(ch)$ *returns the messages in the channel ch, i.e.* $h(\mathsf{A}(ch))$
- $\mathsf{levels}(O)$ *returns the levels of the obligations in O, i.e.* $\{\!\!\{r \mid (a, r) \in O\}\!\!\}$
- $\mathsf{Pt} = \biguplus_{(id,c,\xi,p,O,I,C,T) \in A} p$ *and* $\mathsf{Wt} = \biguplus_{(id,\mathsf{wait}(l),\xi,p,O,I,C,T) \in A} \{\!\!\{l\}\!\!\}$
- $\mathsf{Ct} = \biguplus_{(id,c,\xi,p,O,I,C,T) \in A} C$ *and* $\mathsf{Ot} = \biguplus_{(id,c,\xi,p,O,I,C,T) \in A} O$
- $\mathsf{Tt} = \biguplus_{(id,c,\xi,p,O,I,C,T) \in A} T$ *and* $\mathsf{It} = \biguplus_{(id,c,\xi,p,O,I,C,T) \in A} I$

▶ **Theorem 13** (Small Steps Preserve Validity of Configurations). *Each step of the execution preserves validity of configurations.*

$$(t, h, s) \rightsquigarrow (t', h', s') \wedge \mathsf{valid}_{n+1}(t, h, s) \Rightarrow \mathsf{valid}_n(t', h', s')$$

Proof. By case analysis of the small step relation $\rightsquigarrow$.
Case $(t[id{:=}\mathsf{new\_channel}(b); \xi], h[z{:=}\varnothing], s) \rightsquigarrow (t[id{:=}\mathsf{val}(z); \xi], h[z{:=}[]], b{=}\mathsf{true}\ ?\ s{\cup}\{z\} : s)$:
By $\mathsf{valid}(t[id{:=}\mathsf{new\_channel}(b); \xi], h[z{:=}\varnothing], s)$ we have an augmented thread set A consisting of an element $(id, \mathsf{new\_channel}(b), \xi, p, O, I, C, T)$ which satisfies all the conditions in the definition of validity of configurations, including $p, O, I, C, T \models \mathsf{wpcx}(\mathsf{new\_channel}(b), \xi)$. $\mathsf{valid}(t[id{:=}\mathsf{val}(z); \xi], h[ch{:=}[]], b{=}\mathsf{true}\ ?\ s{\cup}\{z\} : s)$ holds because by Lemma 4 there exists an augmented thread set $A'{=}A{-}(id, \mathsf{new\_channel}(b), \xi, p, O, I, C, T){\cup}(id, \mathsf{val}(z), \xi, p[l{:=}\mathsf{channel}]$ $, O, I, C, T)$ which satisfies all the conditions in the definition of validity of configurations.

Case $(t[id{:=}\mathsf{send}(e_1, e_2); \xi], h[[\![ch]\!]{:=}M], s) \rightsquigarrow (t[id{:=}\mathsf{tt}; \xi], h[[\![e_1]\!]{:=}M.[[\![e_2]\!]]], s)$:
By $\mathsf{valid}(t[id{:=}\mathsf{send}(e_1, e_2); \xi], h[[\![ch]\!]{:=}M], s)$ we have an augmented thread set A consisting of an element $(id, \mathsf{send}(e_1, e_2), \xi, p, O, I, C, T)$ which satisfies all the conditions in the definition of validity of configurations, including $p, O, I, C, T \models \mathsf{wpcx}(\mathsf{send}(e_1, e_2), \xi)$. $\mathsf{valid}(t[id{:=}\mathsf{tt}; \xi],$ $h[[\![e_1]\!]{:=}M.[[\![e_2]\!]]], s)$ holds because by Lemma 5 there exists an augmented thread set $A'{=}A{-}(id, \mathsf{send}(e_1, e_2), \xi, p, O', I', C, T){\cup}$ $(id, \mathsf{tt}, \xi, p_2, O{\uplus}\mathsf{M}'(ch)(m), I{-}\{\!\!\{ch\}\!\!\}, C_2, T_2)$ which satisfies all the conditions in the definition of validity of configurations.

Case $(t[id{:=}\mathsf{receive}(e); \xi], h[[\![e]\!]{:=}[m].M], s) \rightsquigarrow (t[id{:=}\mathsf{val}(m); \xi], h[[\![e]\!]{:=}M], s)$:
By $\mathsf{valid}(t[id{:=}\mathsf{receive}(e); \xi], h[[\![e]\!]{:=}[m].M], s)$ we have an augmented thread set A consisting of an element $(id, \mathsf{receive}(e), \xi, p, O, I, C, T)$ which satisfies all the conditions in the definition of validity of configurations, including $p, O, I, C, T \models \mathsf{wpcx}(\mathsf{receive}(e), \xi)$. $\mathsf{valid}(t[id{:=}\mathsf{val}(m); \xi],$ $h[[\![e]\!]{:=}M], s)$ holds because by Lemma 6 there exists an augmented thread set $A'{=}A{-}(id,$ $\mathsf{receive}(e), \xi, p, O, I, C, T){\cup}(id, \mathsf{val}(m), \xi, p_2, O{\uplus}\mathsf{M}'(ch)(m), I{-}\{\!\!\{ch\}\!\!\}, C_2, T_2)$ which satisfies all the conditions in the definition of validity of configurations.

Case $(t[id{:=}\mathsf{fork}(c); \xi, id'{:=}\varnothing], h, s) \rightsquigarrow (t[id{:=}\mathsf{val}(\mathsf{tt}); \xi, id'{:=}c; \mathsf{done}], h, s)$:
By $\mathsf{valid}_n(t[id{:=}\mathsf{fork}(c); \xi, id'{:=}\varnothing]; \xi, h, s)$ we have an augmented thread set A consisting of an element $(id, \mathsf{fork}(c), \xi, p, O, I, C, T)$ which satisfies all the conditions in the definition of validity of configurations, including $p, O, I, C, T \models \mathsf{wpcx}_n(\mathsf{fork}(c), \xi)$. $\mathsf{valid}(t[id{:=}\mathsf{val}(\mathsf{tt}); \xi, id'{:=}c;$ $\mathsf{done}], h, s)$ holds because by Lemma 7 there exists an augmented thread set $A'{=}A{-}(id, \mathsf{fork}(c)$ $, \xi, p, O, I, C, T){\cup}(id, \mathsf{tt}, \xi, p_1, O_1, I_1, C_1, T_1){\cup}(id', c, \mathsf{done}, p_2, O_2, I_2, C_2, T_2)$ which satisfies all the conditions in the definition of validity of configurations.

Case $(t[id{:=}\mathsf{let}(x,c_1,c_2);\xi],h,s) \rightsquigarrow (t[id{:=}c1;\mathsf{let}'(x,c_2,\xi)],h,s)$:
By $\mathsf{valid}_n(t[id{:=}\mathsf{let}(x,c_1,c_2);\xi],h)$ we have an augmented thread set A consisting of an element $(id,\mathsf{let}(x,c_1,c_2),\xi,p,O,g)$ which satisfies all the conditions in the definition of validity of configurations, including $p,O,I,C,T \models \mathsf{wpcx}_n(\mathsf{let}(x,c_1,c_2),\xi)$. Since $\mathsf{wpcx}_n(\mathsf{let}(x,c_1,c_2),\xi) = \mathsf{wp}_{n-1}(c_1,\lambda z.\ \mathsf{wp}_{n-1}(c_2[z/x],Q))$, we have $p,O,I,C,T \models \mathsf{wp}_{n-1}(c_1,\lambda z.\ \mathsf{wp}_{n-1}(c_2[z/x],Q))$. Consequently $\mathsf{valid}(t[id{:=}c1;\mathsf{let}'(x,c_2,\xi)],h,s)$ holds because there exists an augmented thread set $A'{=}A{-}(id,\mathsf{let}(x,c_1,c_2),\xi,p,O,I,C,T) \cup (id,c_1,\mathsf{let}'(x,c_2,\xi),p,O,I,C,T)$ which satisfies all the conditions in the definition of validity of configurations. The rest of the cases can be proved similarly. ◀

▶ **Theorem 14** (A Valid Configuration Is Not Deadlocked). *If a valid configuration has some threads then either all threads in this configuration are waiting for some server channels, or there exists a thread in this configuration which is not waiting for an empty channel.*

$$\mathsf{valid}_n(t,h,s) \wedge \exists id.\ t(id) \neq \varnothing \Rightarrow \mathsf{NotDeadlock}(t,h,s)$$

where $\mathsf{NotDeadlock}(t,h,s) \Leftrightarrow \mathsf{AllWaitingToServe}(t,s) \vee \exists id'.\ \neg\mathsf{is\_waiting}(\mathsf{fst}(t(id')),h)$*, where*

- $\mathsf{AllWaitingToServe}(t,s) \Leftrightarrow \forall id.\ t(id) \neq \varnothing \Rightarrow \exists e.\ \mathsf{fst}(t(id)) = \mathsf{wait}(e) \wedge [\![e]\!] \in s$
- $\mathsf{is\_waiting}(c,h) \Leftrightarrow \exists e.\ c = \mathsf{wait}(e) \wedge h([\![e]\!]) = []$.

Proof. By contradiction; we assume that all threads in t are waiting for some empty channels where some of these channels are not server channels, i.e. $\forall id.\ \exists e.\ \mathsf{fst}(t(id)){=}\mathsf{wait}(e) \wedge h(e){=}[] \wedge \exists id',e'.\ \mathsf{fst}(t(id')){=}\mathsf{wait}(e') \wedge [\![e']\!] \notin s$. Since (t,h,s) is a valid configuration and all threads in this configuration are waiting for a channel, there exists a set of valid augmented threads A from which we produce a valid bag $G = \mathsf{valid\_bag}(A)$, where $\mathsf{valid\_bag}$ maps any element $(id,\mathsf{wait}(e),\xi,p,O,I,C,T) \in A$ to an element $([\![e]\!],\mathsf{Addresses}(O),\mathsf{Addresses}(I))$ where $\mathsf{Addresses}(O) = \{\!\{\mathsf{A}(o) \mid o \in O\}\!\}$. By Lemma 15, we have $G{=}\{\!\{\}\!\}$, implying $A{=}\{\}$, implying $t{=}\mathbf{0}$ which contradicts the hypothesis of the theorem.

Note that in the definition of validity of a configuration we also keep track of all locations whose addresses are allocated, which makes it possible to provide the functions R, M^r, and S, mapping channel addresses to their ghost information, for Lemma 15. Additionally, the hypotheses H_2, H_3, H_4, and $H_5 \vee H_6$ in Lemma 15 are met as follows. For each element $(id,\mathsf{wait}(e),\xi,p,O,I,C,T) \in A$ we have $p,O,I,C,T \models \mathsf{wpcx}(\mathsf{wait}(e),\xi)$, which implies $0 < C([\![e]\!])$ and there exists a channel ch with address $[\![e]\!]$ such that $ch \prec O$ (which implies H_2) and $ch \prec^r I$ (which implies H_3), and $\mathsf{S}(ch) = \mathsf{true} \Rightarrow O = \{\!\{\}\!\} \wedge \forall o_1.0 < I(o_1) \Rightarrow o_1 = ch$ (which implies H_4). Additionally, by $0 < C([\![e]\!])$ we have $0 < \mathsf{Ct}([\![e]\!])$, which (by $4.c$ in validity of configurations) implies if $\mathsf{S}(ch) = \mathsf{false}$ either 1) $0 < \mathsf{Ot}(ch)$ (which implies H_5), or 2) there exists a message m in a channel ch' through which an obligation of ch is transferred, i.e. $0 < \mathsf{size}(ch')$ and $m \in \mathsf{queue}_h(ch')$ and $0{<}\mathsf{M}'(ch')(m)(ch)$. By $m \in \mathsf{queue}_h(ch')$ and $4.b$ we have $\mathsf{levels}(\mathsf{M}'(ch')(m)) \subseteq \mathsf{M}^{\mathsf{r}}(ch')$, which by $\{\!\{ch\}\!\}{\in}\mathsf{M}'(ch')(m)$ implies $\mathsf{R}(ch) \in \mathsf{M}^{\mathsf{r}}(ch')$. Additionally, by $\mathsf{M}^{\mathsf{r}}(ch') \neq \{\!\{\}\!\}$, and $0 < \mathsf{size}(ch')$, and $4.a$ we have $0 < \mathsf{It}(ch')$ (which implies H_6). ◀

Lemma 15 ensures that in any state of the execution if all the desired invariants are respected then it is impossible that all threads of the program are waiting for some empty channels where some of these channels are not server channels. In this lemma G is a bag of waitable object-obligations-importers triples such that each element t of G is associated with a thread in a state of the execution, where the first element of t is associated with the address of the object for which t is waiting, the second element is associated with the addresses of obligations of t, and the third element is associated with the addresses of importers of t.

▶ **Lemma 15** (A Valid Bag of Augmented Threads Is Not Deadlocked).

$$\begin{aligned}&\forall\, G : \mathit{Bags}(\mathit{Addresses} \times \mathit{Bags}(\mathit{Addresses}) \times \mathit{Bags}(\mathit{Obligations})),\\ &\quad R : \mathit{Addresses} \to \mathit{Levels},\\ &\quad M^r : \mathit{Addresses} \to \mathit{Bags}(\mathit{Levels}),\\ &\quad S : \mathit{Addresses} \to \mathit{Booleans}.\\ &H_1 \land \forall (o, O, I) \in G.\ H_2 \land H_3 \land H_4 \land (H_5 \lor H_6) \Rightarrow G = \{\!\!\}\end{aligned}$$

where

- $H_1 : \exists (o, O, I) \in G.\ S(o) = \mathsf{false}$
- $H_2 : o \prec O$
- $H_3 : o \prec^r I$
- $H_4 : S(o) = \mathsf{true} \Rightarrow O = \{\!\!\} \land \forall o_1.\ 0 < I(o_1) \Rightarrow o_1 = o$
- $H_5 : S(o) = \mathsf{false} \Rightarrow 0 < \mathsf{Ot}(o)$
- $H_6 : S(o) = \mathsf{false} \Rightarrow \exists o_1.\ R(o) \in \mathsf{M^r}(o_1) \land \mathsf{Wt}(o_1) = 0 \land 0 < \mathsf{It}(o_1)$

  where $\mathsf{Wt} = \biguplus_{(o,O,I)\in G} \{o\}$ *and* $\mathsf{Ot} = \biguplus_{(o,O,I)\in G} O$ *and* $\mathsf{It} = \biguplus_{(o,O,I)\in G} I$

Proof. By H_1 we know $\exists (o_m, O_1, I_1) \in G$ where $S(o_m) = \mathsf{false}$ and $\forall (o, O, I) \in G.\ S(o) = \mathsf{false} \Rightarrow R(o_m) \leqslant R(o)$. By $(H_5 \lor H_6)$ there are two cases: 1) $\exists (o_2, \{o_m\}\uplus O_2, I_2) \in G$, or 2) $\exists (o_3, O_3, \{o_1\}\uplus I_3) \in G.\ R(o_m) \in M^r(o_1) \land \mathsf{Wt}(o_1) = 0$. In the first case by H_4 we have $S(o_2) = \mathsf{false}$, which implies $R(o_m) \leqslant R(o_2)$, that contradicts the hypothesis H_2, i.e $o_2 \prec \{o_m\}\uplus O_2$. In the second case by $\mathsf{Wt}(o_1) = 0$ we have $o_1 \neq o_3$ (because $0 < \mathsf{Wt}(o_3)$), and consequently by H_4 we have $S(o_3) = \mathsf{false}$, which implies $R(o_m) \leqslant R(o_3)$, that contradicts the hypothesis H_3, i.e. $o_3 \prec^r \{o_1\}\uplus I_3$ (because $R(o_m) \in M^r(o_1)$). ◀

▶ **Theorem 16** (The Initial Configuration Is Valid).

$$\mathsf{correct}(\mathsf{obs}(\{\!\!\}, \{\!\!\}), c, \lambda\_.\mathsf{obs}(\{\!\!\}, \{\!\!\})) \Rightarrow \forall n, id.\ \mathsf{valid}_n(\mathbf{0}[id{:=}c; \mathsf{done}], \mathbf{0}, [])$$

Proof. The goal is achieved because there exists an augmented thread set $A{=}[(id, c, \mathsf{done}, \mathbf{0}, \mathit{0}, \mathit{0}, \mathit{0}, \mathit{0})]$, such that all the conditions in the definition of validity of configurations are met, where $\mathbf{0} = \lambda\_.\varnothing$ and $\mathit{0} = \lambda\_.0$. ◀

C.6 An Example Proof

In this section we show how the program in Figure 3 can be verified using the proof rules in Figure 31, as shown in Figure 32.

D Transferring Obligations Through Notifications: Soundness Proof

In this appendix we provide a formalization and soundness proof, machine-checked with Coq[11], for the approach introduced in Section 3. However, unfortunately, there are a few technical differences between this formalization and the system of Section 3 such that in this formalization the ghost information, such as level and transferred permissions and obligations, are associated with lock and condition variable addresses via the lock and cond permissions

[11] The soundness proof of the second mechanism, machine-checked in Coq, can be found in [17].

```
$\mathsf{ob}(a) ::= (a, 1, \{\!\{1\}\!\}, \mathsf{false})$
$\mathsf{ob}'(a) ::= (a, 1, \{\!\{\}\!\}, \mathsf{false})$
$\mathsf{loc}(a) ::= (\mathsf{ob}(a), (\mathsf{Mch}, []), \lambda m.\ \{\!\{\mathsf{ob}'(\mathsf{snd}(m))\}\!\})$
$\mathsf{loc}'(a) ::= (\mathsf{ob}'(a), (\mathsf{Mch}', []), \lambda\_.\ \{\!\{\}\!\})$
$\mathsf{pt}(\mathsf{Mch}, args) ::= \lambda m.\ \mathsf{channel}(\mathsf{loc}'(\mathsf{snd}(m)))$
$\mathsf{pt}(\mathsf{Mch}', args) ::= \lambda m.\ \mathsf{true}$

$\mathbf{routine}\ \mathsf{server}(\mathsf{channel}\ a)\{$
$\mathbf{req} : \{\mathsf{obs}(\{\!\{\}\!\}, \{\!\{\mathsf{ob}(a)\}\!\}) * \mathsf{channel}(\mathsf{loc}(a)) * \mathsf{credit}(a)\}$
 $(req, a') := \mathsf{receive}(ch);$
 $\{\mathsf{obs}(\{\!\{\mathsf{ob}'(a')\}\!\}, \{\!\{\}\!\}) * \mathsf{channel}(\mathsf{loc}'(a'))\}$
 $result := \mathsf{process}(req);$
 $\mathsf{send}(a', result)$
$\mathbf{ens} : \{\mathsf{obs}(\{\!\{\}\!\}, \{\!\{\}\!\})\}\}$

$\mathbf{routine}\ \mathsf{client}(\mathsf{channel}\ a)\{$
$\mathbf{req} : \{\mathsf{obs}(\{\!\{\mathsf{ob}(a)\}\!\}, \{\!\{\}\!\}) * \mathsf{channel}(\mathsf{loc}(a)) * \mathsf{trandit}(a)\}$
 $a' := \mathsf{new\_channel}();$
 $\{\mathsf{obs}(\{\!\{\mathsf{ob}(a)\}\!\}, \{\!\{\}\!\}) * \mathsf{channel}(\mathsf{loc}'(a')) * \mathsf{trandit}(a)\}$
 nop; //Rule Credit
 $\{\mathsf{obs}(\{\!\{\mathsf{ob}(a), \mathsf{ob}'(a')\}\!\}, \{\!\{\}\!\}) * \mathsf{trandit}(a) * \mathsf{credit}(a')\}$
 $\mathsf{send}(a, (\mathsf{request}(), a'));$
 $\{\mathsf{obs}(\{\!\{\}\!\}, \{\!\{\}\!\}) * \mathsf{credit}(a')\}$
 $\mathsf{receive}(a')$
$\mathbf{ens} : \{\mathsf{obs}(\{\!\{\}\!\}, \{\!\{\}\!\})\}\}$

$\mathbf{routine}\ \mathsf{main}()\{$
$\mathbf{req} : \{\mathsf{obs}(\{\!\{\}\!\}, \{\!\{\}\!\})\}$
 $a := \mathsf{new\_channel}();$
 $\{\mathsf{obs}(\{\!\{\}\!\}, \{\!\{\}\!\}) * \mathsf{channel}(\mathsf{loc}(a))\}$
 nop; //Rule Credit
 $\{\mathsf{obs}(\{\!\{\mathsf{ob}(a)\}\!\}, \{\!\{\}\!\}) * \mathsf{credit}(a)\}$
 nop; //Rule Trandit
 $\{\mathsf{obs}(\{\!\{\mathsf{ob}(a)\}\!\}, \{\!\{\mathsf{ob}(a)\}\!\}) * \mathsf{credit}(a) * \mathsf{trandit}(a)\}$
 $\mathsf{fork}($
   $\{\mathsf{obs}(\{\!\{\}\!\}, \{\!\{\mathsf{ob}(a)\}\!\}) * \mathsf{credit}(a)\}$
   $\mathsf{server}(a)$
   $\{\mathsf{obs}(\{\!\{\}\!\}, \{\!\{\}\!\})\});$
 $\{\mathsf{obs}(\{\!\{\mathsf{ob}(a)\}\!\}, \{\!\{\}\!\}) * \mathsf{trandit}(a)\}$
 $\mathsf{client}(a)$
$\mathbf{ens} : \{\mathsf{obs}(\{\!\{\}\!\}, \{\!\{\}\!\})\}\}$
```

Figure 32 Verification of the program in Figure 3 using the rules in Figure 31.

$$\begin{array}{l}
c \in \mathit{Commands},\ \ e \in \mathit{Expressions},\ \ z \in \mathbb{Z},\ \ x \in \mathit{Variables} \\
e ::= z \mid x \mid e_1{+}e_2 \mid -e \\
c ::= \mathsf{val}(e) \mid \mathsf{new\_int}(z) \mid \mathsf{lookup}(e) \mid \mathsf{mutate}(e_1, e_2) \\
\quad\quad \mid \mathsf{if}(c, c_1, c_2) \mid \mathsf{while}(c, c_1) \mid \mathsf{let}(x, c_1, c_2) \mid \mathsf{fork}(c) \mid \mathsf{new\_lock} \mid \mathsf{acquire}(e) \mid \mathsf{release}(e) \\
\quad\quad \mid \mathsf{new\_cvar} \mid \mathsf{wait}(e_1, e_2) \mid \mathsf{notify}(e) \mid \mathsf{notifyAll}(e) \\
\quad\quad \mid \mathsf{waiting4lock}(e) \mid \mathsf{waiting4cvar}(e_1, e_2) \mid \mathsf{nop}
\end{array}$$

Figure 33 Syntax of the programming language.

rather than via global functions[12]. The proof rules associated with this formalization and the verification of the program in Figure 16, proved using these rules, are shown in Sections D.4 and D.6, respectively.

D.1 Syntax and Semantics of Programs

We define the syntax of our programming language as indicated in Figure 33. In this syntax an arithmetic expression, e, can be an integer value, z, a variable, x, an addition of two expressions, or a negation of an expression. An integer value can be substituted for a free variable in an expression or a command, and each expression can be evaluated to an integer value, as shown in Figure 34. Commands include commands $\mathsf{val}(e)$ which simply yield the value of e as their result and have no side effects, memory allocations[13], memory reads, memory writes, conditionals, loops, parallel composition, sequential composition, lock creations, lock acquisitions, lock releases, condition variable creations, waits, and notifications. We also define some extra commands waiting4lock, indicating that the related thread is waiting for a lock, and waiting4cvar, indicating that the related thread has executed wait; these are not supposed to appear in the source program and appear only during execution. Additionally, instead of defining all the ghost commands introduced in Section 3, we define a single ghost command nop which is inserted into the program for verification purposes and has no effect on the program's behavior. The small step semantics, defined in Figure 36, relates two *configurations*, defined in Figure 35.

D.2 Syntax and Semantics of Assertions

The syntax of assertions is defined in Figure 37[14]. Note that a *location* l of an object o consists of the *obligation* of o (if o is a lock or a CV); the lock invariant of o (if o is a lock), denoted by $\mathsf{I}(l)$; and the permissions and the obligations which are transferred through a notification on o (if o is a CV), denoted by $\mathsf{M}(l)$ and $\mathsf{M}'(l)$ respectively. Also note that permissions described by invariants of locks as well as permissions which are transferred through notifications are specified through an index (as well as the required arguments)

[12] Note that one way to formalize the precise approach of Section 3 would be to define assertions as functions from ghost information to separating conjunctions of chunks. In the soundness proof, one would track these as partial functions whose domain is the set of allocated addresses. The functions passed into the assertions would be *totalizations* of these partial functions. An assertion is true if it is true for all totalizations of the functions.

[13] Note that $\mathsf{new\_int}(n)$ allocates n consecutive memory locations and returns the address of the first one.

[14] Note that we use a shallow embedding: assertions have no variables; to model quantifications, we use meta-level functions from values to assertions.

$$
\begin{array}{l}
[\![]\!] \in \mathit{Expressions} \to \mathbb{Z} \\
\\
[\![z]\!]{=}z \\
[\![x]\!]{=}0 \\
[\![e_1{+}e_2]\!]{=}[\![e_1]\!]{+}[\![e_2]\!] \\
[\![{-}e]\!]{=}-[\![e]\!] \\
\\
z[z'/x] = z \\
x[z/x'] = x \quad \text{if } x \neq x' \\
x[z/x] = z \\
(e_1{+}e_2)[z/x] = e_1[z/x]{+}e_2[z/x] \\
\\
\mathsf{val}(e)[z/x] = \mathsf{val}(e[z/x]) \\
\mathsf{let}(x, c_1, c_2)[z/x] = \mathsf{let}(x, c_1, c_2) \\
\mathsf{let}(x, c_1, c_2)[z/x'] = \mathsf{let}(x, c_1[z/x'], c_2[z/x']) \quad \text{if } x \neq x' \\
\mathsf{fork}(c)[z/x] = \mathsf{fork}(c[z/x]) \\
\mathsf{new\_lock}[z/x] = \mathsf{new\_lock} \\
\mathsf{new\_int}(z')[z/x] = \mathsf{new\_int}(z') \\
\mathsf{lookup}(e)[z/x] = \mathsf{lookup}(e[z/x]) \\
\mathsf{mutate}(e_1, e_2)[z/x] = \mathsf{mutate}(e_1[z/x], e_2[z/x]) \\
\mathsf{acquire}(e)[z/x] = \mathsf{acquire}(e[z/x]) \\
\mathsf{release}(e)[z/x] = \mathsf{release}(e[z/x]) \\
\mathsf{new\_cvar}[z/x] = \mathsf{new\_cvar} \\
\mathsf{wait}(e_1, e_2)[z/x] = \mathsf{wait}(e_1[z/x], e_2[z/x]) \\
\mathsf{notify}(e)[z/x] = \mathsf{notify}(e[z/x]) \\
\mathsf{notifyAll}(e)[z/x] = \mathsf{notifyAll}(e[z/x]) \\
\mathsf{waiting4lock}(e)[z/x] = \mathsf{waiting4lock}(e[z/x]) \\
\mathsf{waiting4cvar}(e_1, e_2)[z/x] = \mathsf{waiting4cvar}(e_1[z/x], e_2[z/x]) \\
\mathsf{while}(c, c_1)[z/x] = \mathsf{while}(c[z/x], c_1[z/x]) \\
\mathsf{if}(c, c_1, c_2)[z/x] = \mathsf{if}(c[z/x], c_1[z/x], c_2[z/x]) \\
\mathsf{nop}[z/x] = \mathsf{nop}
\end{array}
$$

Figure 34 Evaluation of expressions and substitution of expressions and commands.

$$
\begin{array}{l}
\mathit{Addresses} = \mathbb{Z} \\
\mathit{ThreadIds} = \mathbb{Z} \\
h \in \mathit{Heaps} = \mathit{Addresses} \rightharpoonup \mathbb{Z} \\
\xi \in \mathit{Contexts} ::= \mathsf{done} \mid \mathsf{let}'(x, c, \xi) \mid \mathsf{if}'(c_1, c_2, \xi) \\
\theta \in \mathit{ThreadConfigurations} ::= (c; \xi) \\
t \in \mathit{ThreadTables} = \mathit{ThreadIds} \rightharpoonup \mathit{ThreadConfigurations} \\
\kappa \in \mathit{Configurations} = \mathit{Heaps} \times \mathit{ThreadTables}
\end{array}
$$

Figure 35 Configurations.

$$(t[id{:=}\mathsf{new\_int}(n);\xi], h[z...z{+}n{-}1{:=}\varnothing]) \rightsquigarrow (t[id{:=}\mathsf{val}(z);\xi], h[z...z{+}n{-}1{:=}0])$$

$$(t[id{:=}\mathsf{lookup}(e);\xi], h[\llbracket e\rrbracket{:=}z]) \rightsquigarrow (t[id{:=}\mathsf{val}(z);\xi], h[\llbracket e\rrbracket{:=}z])$$

$$(t[id{:=}\mathsf{mutate}(e_1,e_2);\xi], h) \rightsquigarrow (t[id{:=}\mathsf{tt};\xi], h[\llbracket e_1\rrbracket{:=}\llbracket e_2\rrbracket])$$

$$(t[id{:=}\mathsf{if}(c,c_1,c_2);\xi], h) \rightsquigarrow (t[id{:=}c;\mathsf{if}'(c1,c2,\xi)], h)$$

$$(t[id{:=}\mathsf{val}(e);\mathsf{if}'(c_1,c_2,\xi)], h) \rightsquigarrow (t[id{:=}c_1;\xi], h) \quad \text{if } 0{<}\llbracket e\rrbracket$$

$$(t[id{:=}\mathsf{val}(e);\mathsf{if}'(c_1,c_2,\xi)], h) \rightsquigarrow (t[id{:=}c_2;\xi], h) \quad \text{if } \llbracket e\rrbracket{\leqslant}0$$

$$(t[id{:=}\mathsf{while}(c,c_1);\xi], h) \rightsquigarrow (t[id{:=}\mathsf{if}(c,\mathsf{let}(x,c_1,\mathsf{while}(c,c_1)),\mathsf{tt}),\xi], h)$$

where x is not free in c and c_1

$$(t[id{:=}\mathsf{fork}(c);\xi, id'{:=}\varnothing], h) \rightsquigarrow (t[id{:=}\mathsf{tt};\xi, id'{:=}c;\mathsf{done}], h)$$

$$(t[id{:=}\mathsf{let}(x,c_1,c_2);\xi], h) \rightsquigarrow (t[id{:=}c_1;\mathsf{let}'(x,c_2,\xi)], h)$$

$$(t[id{:=}\mathsf{val}(e);\mathsf{let}'(x,c,\xi)], h) \rightsquigarrow (t[id{:=}c[\llbracket e\rrbracket/x];\xi], h)$$

$$(t[id{:=}\mathsf{val}(e);\mathsf{done}], h) \rightsquigarrow (t[id{:=}\varnothing], h)$$

$$(t[id{:=}\mathsf{new\_lock};\xi], h[z{:=}\varnothing]) \rightsquigarrow (t[id{:=}\mathsf{val}(z);\xi], h[z{:=}1])$$

$$(t[id{:=}\mathsf{acquire}(e);\xi], h[\llbracket e\rrbracket{:=}1]) \rightsquigarrow (t[id{:=}\mathsf{tt};\xi], h[\llbracket e\rrbracket{:=}0])$$

$$(t[id{:=}\mathsf{acquire}(e);\xi], h[\llbracket e\rrbracket{:=}0]) \rightsquigarrow (t[id{:=}\mathsf{waiting4lock}(e);\xi], h[\llbracket e\rrbracket{:=}0])$$

$$(t[id{:=}\mathsf{waiting4lock}(e);\xi], h[\llbracket e\rrbracket{:=}1]) \rightsquigarrow (t[id{:=}\mathsf{tt};\xi], h[\llbracket e\rrbracket{:=}0])$$

$$(t[id{:=}\mathsf{release}(e);\xi], h) \rightsquigarrow (t[id{:=}\mathsf{tt};\xi], h[\llbracket e\rrbracket{:=}1])$$

$$(t[id{:=}\mathsf{new\_cvar};\xi], h[z{:=}\varnothing]) \rightsquigarrow (t[id{:=}\mathsf{val}(z);\xi], h[z{:=}0])$$

$$(t[id{:=}\mathsf{wait}(e_1,e_2);\xi], h) \rightsquigarrow (t[id{:=}\mathsf{waiting4cvar}(e_1,e_2);\xi], h[\llbracket e_2\rrbracket{:=}1])$$

$$(t[id{:=}\mathsf{notify}(e);\xi, id'{:=}\mathsf{waiting4cvar}(e_1,e_2);\xi'], h) \rightsquigarrow$$
$$(t[id{:=}\mathsf{val}(\mathsf{tt});\xi, id'{:=}\mathsf{waiting4lock}(e_2);\xi'], h) \quad \text{if } \llbracket e\rrbracket = \llbracket e_1\rrbracket$$

$$(t[id{:=}\mathsf{notify}(e);\xi], h) \rightsquigarrow (t[id{:=}\mathsf{tt};\xi], h) \quad \text{if } \mathsf{nowaiting}(\llbracket e\rrbracket, t)$$

$$(t[id{:=}\mathsf{notifyAll}(e);\xi], h) \rightsquigarrow (\mathsf{wkup}(\llbracket e\rrbracket, t[id{:=}\mathsf{val}(\mathsf{tt});\xi]), h)$$

$$(t[id{:=}\mathsf{nop};\xi], h) \rightsquigarrow (t[id{:=}\mathsf{tt};\xi], h)$$

where

$$\mathsf{wkup}(z,t) = \lambda id.\ \begin{cases} \mathsf{waiting4lock}(l);\xi & \text{if } t(id) = \mathsf{waiting4cvar}(v,l);\xi \wedge \llbracket v\rrbracket = z \\ t(id) & \text{otherwise} \end{cases}$$

$$\mathsf{nowaiting}(z,t) \Leftrightarrow \nexists id,\xi,l,v.\ \llbracket v\rrbracket{=}z \wedge t(id){=}\mathsf{waiting4cvar}(v,l);\xi$$

Figure 36 Semantics of programs, where tt stands for $\mathsf{val}(0)$.

$$
\begin{array}{l}
\mathit{Bags}(A) = A \rightarrow \mathbb{N} \\
\mathit{Wt}, \mathit{Ot} \in \mathit{Bags}(\mathbb{Z}) \\
\mathit{Indexes} = \mathbb{Z} \\
\mathit{Arguments} = \mathbb{Z} \\
r \in \mathit{Levels} = \mathbb{R} \\
o \in \mathit{Obligations} = \mathit{Addresses} \times \mathit{Levels} \times \mathit{Addresses} \\
l \in \mathit{Locations} = \mathit{Obligations} \times (\mathit{Indexes} \times \mathit{Lists}(\mathit{Arguments})) \times \\
\quad (\mathit{Indexes} \times \mathit{Lists}(\mathit{Arguments})) \times \mathit{Bags}(\mathit{Obligations}) \\
O \in \mathit{Bags}(\mathit{Obligations}) \\
b \in \mathit{Booleans} \\
\hat{v} \in \mathit{AValues} ::= z \mid r \mid b \mid l \mid o \mid O \\
\alpha \in \mathit{AValues} \rightarrow \mathit{Assertions} \\
\pi \in \mathit{Fractions} \\
a \in \mathit{Assertions} ::= l \overset{\pi}{\mapsto} z \mid \mathsf{ulock}(l, \mathit{Wt}, \mathit{Ot}) \mid \mathsf{lock}(l) \mid \mathsf{locked}(l, \mathit{Wt}, \mathit{Ot}) \mid \mathsf{ucond}(l) \mid \mathsf{cond}(l) \\
\qquad\qquad\qquad\quad \mid \mathsf{obs}(O) \mid \mathsf{ctr}(z, n) \mid \mathsf{tic}(z) \\
\qquad\qquad\qquad\quad \mid b \mid a_1 \wedge a_2 \mid a_1 \vee a_2 \mid a_1 * a_2 \mid a_1 \mathrel{-\!\!*} a_2 \mid \forall \alpha \mid \exists \alpha
\end{array}
$$

$$
\begin{array}{l}
\mathsf{pt} : \mathit{PredicateTables} = \mathit{Indexes} \rightarrow \mathit{Lists}(\mathit{Arguments}) \rightarrow \\
\quad \mathit{Bags}(\mathit{Obligations}) \rightarrow \mathit{Bags}(\mathit{Obligations}) \rightarrow \mathit{Assertions} \\
\mathsf{O} : \mathit{Locations} \rightarrow \mathit{Obligations} \text{ , where } \mathsf{O}((A, R, L), I, M, M') = (A, R, L) \\
\mathsf{A} : \mathit{Locations} \rightarrow \mathit{Addresses} \text{ , where } \mathsf{A}((A, R, L), I, M, M') = A \\
\mathsf{R} : \mathit{Locations} \rightarrow \mathit{Levels} \text{ , where } \mathsf{R}((A, R, L), I, M, M') = R \\
\mathsf{L} : \mathit{Locations} \rightarrow \mathit{Addresses} \text{ , where } \mathsf{L}((A, R, L), I, M, M') = L \\
\mathsf{I} : \mathit{Locations} \rightarrow \mathit{Bags}(\mathit{Obligations}) \rightarrow \mathit{Bags}(\mathit{Obligations}) \rightarrow \mathit{Assertions} \\
\quad \text{where } \mathsf{I}((A, R, L), I, M, M') = \mathsf{pt}(\mathsf{fst}(I), \mathsf{snd}(I)) \\
\mathsf{M} : \mathit{Locations} \rightarrow \mathit{Assertions}, \\
\quad \text{where } \mathsf{M}((A, R, L), I, M, M') = \mathsf{pt}(\mathsf{fst}(M), \mathsf{snd}(M), \Lbag\Rbag, \Lbag\Rbag) \\
\mathsf{M}' : \mathit{Locations} \rightarrow \mathit{Bags}(\mathit{Obligations}) \text{ , where } \mathsf{M}'((A, R, L), I, M, M') = M'
\end{array}
$$

Figure 37 Syntax of assertions.

$$
\begin{array}{l}
k \in \mathit{Knowledge} ::= \mathsf{cell}(\pi, z) \mid \mathsf{ulock}(\mathit{Wt}, \mathit{Ot}) \mid \mathsf{lock} \mid \mathsf{locked}(\mathit{Wt}, \mathit{Ot}) \mid \mathsf{ucond} \mid \mathsf{cond} \\
p \in \mathit{PermissionHeaps} = \mathit{Locations} \rightharpoonup \mathit{Knowledge} \\
\mathit{GhostIdentifications} = \mathbb{Z} \\
\mathit{gv} \in \mathit{GhostValues} ::= \mathit{Option}(\mathbb{N}) \times \mathbb{N} \\
g \in \mathit{GhostHeaps} = \mathit{GhostIdentifications} \rightharpoonup \mathit{GhostValues} \\
\mathit{Option}(A) ::= s \mid \varnothing \text{ , where } s \in A \\
\tilde{O} \in \mathit{Option}(\mathit{Bags}(\mathit{Obligations}))
\end{array}
$$

$$
\begin{array}{l}
p, \tilde{O}, g \models l \overset{\pi}{\mapsto} z \quad \Leftrightarrow \quad p(l) = \mathsf{cell}(\pi, z) \\
p, \tilde{O}, g \models \mathsf{ulock}(l, \mathit{Wt}, \mathit{Ot}) \quad \Leftrightarrow \quad p(l) = \mathsf{ulock}(\mathit{Wt}, \mathit{Ot}) \\
p, \tilde{O}, g \models \mathsf{lock}(l) \quad \Leftrightarrow \quad p(l) = \mathsf{lock}() \\
p, \tilde{O}, g \models \mathsf{locked}(l, \mathit{Wt}, \mathit{Ot}) \quad \Leftrightarrow \quad p(l) = \mathsf{locked}(\mathit{Wt}, \mathit{Ot}) \\
p, \tilde{O}, g \models \mathsf{ucond}(l) \quad \Leftrightarrow \quad p(l) = \mathsf{ucond} \\
p, \tilde{O}, g \models \mathsf{cond}(l) \quad \Leftrightarrow \quad p(l) = \mathsf{cond} \\
p, \tilde{O}, g \models \mathsf{obs}(O) \quad \Leftrightarrow \quad \tilde{O} = O \\
p, \tilde{O}, g \models \mathsf{ctr}(z, n) \quad \Leftrightarrow \quad \exists n_1.\, g(z) = (n, n_1) \\
p, \tilde{O}, g \models \mathsf{tic}(z) \quad \Leftrightarrow \quad \exists n, \tilde{n}.\, g(z) = (\tilde{n}, n{+}1) \\
p, \tilde{O}, g \models b \quad \Leftrightarrow \quad b = \mathsf{true} \\
p, \tilde{O}, g \models a_1 \wedge a_2 \quad \Leftrightarrow \quad p, \tilde{O}, g \models a_1 \wedge p, \tilde{O}, g \models a_2 \\
p, \tilde{O}, g \models a_1 \vee a_2 \quad \Leftrightarrow \quad p, \tilde{O}, g \models a_1 \vee p, \tilde{O}, g \models a_2 \\
p, \tilde{O}, g \models a_1 * a_2 \quad \Leftrightarrow \quad \exists p_1, p_2, \tilde{O}_1, \tilde{O}_2, g_1, g_2.\, p{=}p_1{\uplus}p_2 \wedge \tilde{O}{=}\tilde{O}_1{\uplus}\tilde{O}_2 \wedge g{=}g_1{\uplus}g_2 \wedge \\
\quad p_1, \tilde{O}_1, g_1 \models a_1 \wedge p_2, \tilde{O}_2, g_2 \models a_2 \\
p, \tilde{O}, g \models a_1 \mathrel{-\!\!*} a_2 \quad \Leftrightarrow \quad \forall p_1, \tilde{O}_1, g_1.\, p_1, \tilde{O}_1, g_1 \models a_1 \;\Rightarrow \\
\quad \forall p_2, \tilde{O}_2, g_2.\, p_2{=}p{\uplus}p_1 \wedge g_2{=}g{\uplus}g_1 \wedge \tilde{O}_2{=}\tilde{O}{\uplus}\tilde{O}_1 \;\Rightarrow\; p_2, \tilde{O}_2, g_2 \models a_2 \\
p, \tilde{O}, g \models \forall \alpha \quad \Leftrightarrow \quad \forall \hat{v}{\in}\mathit{AValues}.\, p, \tilde{O}, g \models \alpha(\hat{v}) \\
p, \tilde{O}, g \models \exists \alpha \quad \Leftrightarrow \quad \exists \hat{v}{\in}\mathit{AValues}.\, p, \tilde{O}, g \models \alpha(\hat{v})
\end{array}
$$

$$
a_1 \vdash a_2 \quad \Leftrightarrow \quad (\forall p, \tilde{O}, g.\, p, \tilde{O}, g \models a_1 \Rightarrow p, \tilde{O}, g \models a_2)
$$

Figure 38 Satisfaction relation.

$$\begin{aligned}
&O_1, O_2 \in \mathit{Bags}(A)\\
&\tilde{O}_1, \tilde{O}_2 \in \mathit{Option}(A)\\
&gv_1, gv_2 \in \mathit{GhostValues}\\
&g_1, g_2 \in \mathit{GhostIdentifications} \rightharpoonup \mathit{GhostValues}\\
&k_1, k_2 \in \mathit{Knowledge}\\
&p_1, p_2 \in \mathit{PermissionHeaps}
\end{aligned}$$

$$O_1 \uplus O_2 = \lambda v.\ O_1(v) + O_2(v)$$

$$\tilde{O}_1 \uplus \tilde{O}_2 = \begin{cases} \tilde{O}_1 & \text{if } \tilde{O}_2{=}\varnothing \\ \tilde{O}_2 & \text{if } \tilde{O}_1{=}\varnothing \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\tilde{O}_1 \tilde{\uplus} \tilde{O}_2 = \begin{cases} \tilde{O}_1 & \text{if } \tilde{O}_2 = \varnothing \\ \tilde{O}_2 & \text{if } \tilde{O}_1 = \varnothing \\ \tilde{O}_1 \uplus \tilde{O}_2 & \text{otherwise} \end{cases}$$

$$gv_1 \uplus gv_2 = \begin{cases} (\tilde{m}_1 \uplus \tilde{m}_2, n_1{+}n_2) & \text{if } gv_1 = (\tilde{m}_1, n_1) \land gv_2 = (\tilde{m}_2, n_2) \land \\ & (\tilde{m}_1 \uplus \tilde{m}_2 = n \Rightarrow n \geqslant n_1{+}n_2) \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$g_1 \uplus g_2 = \begin{cases} \lambda l.\ g_1(l) \tilde{\uplus} g_2(l) & \text{if } \exists g.\ \forall l.\ g(l) = g_1(l) \tilde{\uplus} g_2(l) \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$k_1 \uplus k_2 = \begin{cases} \mathsf{cell}(\pi{+}\pi', z) & \text{if } k_1{=}\mathsf{cell}(\pi, z) \land k_2{=}\mathsf{cell}(\pi', z) \land \pi{+}\pi' \leqslant 1 \\ \mathsf{lock} & \text{if } k_1{=}\mathsf{lock} \land k_2{=}\mathsf{lock} \\ \mathsf{locked}(\mathit{Wt}, \mathit{Ot}) & \text{if } k_1{=}\mathsf{lock} \land k_2 = \mathsf{locked}(\mathit{Wt}, \mathit{Ot}) \\ \mathsf{locked}(\mathit{Wt}, \mathit{Ot}) & \text{if } k_1 = \mathsf{locked}(\mathit{Wt}, \mathit{Ot}) \land k_2{=}\mathsf{lock} \\ \mathsf{cond} & \text{if } k_1{=}\mathsf{cond} \land k_2{=}\mathsf{cond} \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$p_1 \uplus p_2 = \begin{cases} \lambda l.\ p_1(l) \tilde{\uplus} p_2(l) & \text{if } \exists p.\ \forall l.\ p(l) = p_1(l) \tilde{\uplus} p_2(l) \\ \text{undefined} & \text{otherwise} \end{cases}$$

Figure 39 Operations on ghost resources.

$$\begin{array}{l}
\mathsf{pheap\_heap}(p,h) \Leftrightarrow \\
(\forall z.\ (\forall l.\ \mathsf{A}(l) = z \Rightarrow p(l) = \varnothing) \Rightarrow h(z) = \varnothing) \text{ and} \\
\forall l. \\
\quad p(l) = \varnothing \text{ or} \\
\quad \forall z.\ (\exists \pi.\ p(l) = \mathsf{cell}(\pi, z)) \Rightarrow h(\mathsf{A}(l)) = z \text{ and} \\
\quad (\exists O_1, O_2.\ p(l) = \mathsf{ulock}(O_1, O_2)) \Rightarrow h(\mathsf{A}(l)) = 1 \text{ and} \\
\quad p(l) = \mathsf{lock} \Rightarrow h(\mathsf{A}(l)) = 1 \text{ and} \\
\quad (\exists O_1, O_2.\ p(l) = \mathsf{locked}(O_1, O_2)) \Rightarrow h(\mathsf{A}(l)) = 0 \text{ and} \\
\quad p(l) = \mathsf{cond} \Rightarrow h(\mathsf{A}(l)) \neq \varnothing \text{ and} \\
\quad p(l) = \mathsf{ucond} \Rightarrow h(\mathsf{A}(l)) \neq \varnothing
\end{array}$$

Figure 40 Permission heaps corresponding to concrete heaps.

pointing to a table in which each element is a function that given a list of arguments returns an assertion. This makes it possible to quantify over locations in assertions[15]. The obligation of a location l, denoted by $\mathsf{O}(l)$, consists of the address of that location, denoted by $\mathsf{A}(l)$, as well as other ghost information such as the level of l, denoted by $\mathsf{R}(l)$; and the lock associated with l (if l is the location of a CV), denoted by $\mathsf{L}(l)$.

These assertions describe some ghost resources, namely p, O, and g that keep track of heap locations, obligations, and ghost counters, respectively, shown in Figure 38, where some operations and relations defined on these resources are shown in Figures 39, 40.

D.3 Weakest Precondition of Commands

The weakest precondition of a command c for $n{>}0$ steps w.r.t. a postcondition Q (with a given predicate table, specified by pt), denoted by $\mathsf{wp}_{n,pt}(c, Q)$ is defined in Figures 41 and 42. Note that $\mathsf{wp}(c, Q)_{0,pt} = \mathsf{true}$. Also note that for the sake of simplicity the index pt is elided. Having this definition, we define the weakest precondition of a context and the weakest precondition of a command-context as shown in Definitions 17 and 18. Having these definitions, we can prove some auxiliary lemmas, shown in Lemmas 20, 21, 22, 23, and 24, which are used to prove Theorem 31.

▶ **Definition 17** (Weakest Precondition of a Context).

$$\mathsf{wpx}_n(\xi) = \begin{cases} \lambda\_.\ \mathsf{obs}(\{\!\}) & \textit{if } \xi{=}\mathsf{done} \\ \lambda z.\ \mathsf{wp}_n(c[z/x], \mathsf{wpx}_n(\xi')) & \textit{if } \xi{=}\mathsf{let}'(x, c, \xi') \\ \lambda z.\ 0{<}z\ ?\ \mathsf{wp}_n(c_1, (\mathsf{wpx}_n(\xi'))) : \mathsf{wp}_n(c_2, (\mathsf{wpx}_n(\xi'))) & \textit{if } \xi{=}\mathsf{if}'(c_1, c_2, \xi') \end{cases}$$

▶ **Definition 18** (Weakest Precondition of a command-context).

$$\mathsf{wpcx}_n(c, \xi) = \mathsf{wp}_n(c, \mathsf{wpx}_n(\xi))$$

▶ **Lemma 19** (Weakening Postcondition).

$$p, \tilde{O}, g \models \mathsf{wp}_n(c, Q) \wedge (\forall z.\ Q(z) \vdash Q'(z)) \Rightarrow \forall n' {\leqslant} n.\ p, \tilde{O}, g \models \mathsf{wp}_{n'}(c, Q')$$

[15] An alternative approach is to use a step-indexed domain of assertions, as in Iris [27]. There, ▶ *Assertions* could be used instead of *Indexes* × *Lists*(*Arguments*), where ▶ is Iris's *guard* for *guarded recursive definitions*.

$$
\begin{array}{l}
\mathsf{wp} \in \mathit{WeakestPreconditions} = \\
\quad \mathit{Commands} \rightarrow (\mathbb{Z} \rightarrow \mathit{Assertions}) \rightarrow \mathbb{N} \rightarrow \mathit{PredicateTables} \rightarrow \mathit{Assertions}
\end{array}
$$

$$
\begin{array}{l}
\mathsf{wp}_n(\mathsf{val}(e), Q) = Q(\llbracket e \rrbracket) \\
\mathsf{wp}_n(\mathsf{new_lock}, Q) = \forall z.\ \exists r.\ \mathsf{ulock}(((z, r, z), (0, []), (0, []), \{\!\!\}\!\!\}), \{\!\!\}\!\!\}, \{\!\!\}\!\!\}) \mathrel{-\!\!*} Q(z) \\
\mathsf{wp}_n(\mathsf{acquire}(e), Q) = \exists O, l.\ (\mathsf{lock}(l) * \mathsf{obs}(O) \wedge \mathsf{O}(l) \prec O \wedge \mathsf{A}(l) = \llbracket e \rrbracket) * \\
\quad (\forall \mathit{Wt}, \mathit{Ot}.\ (\mathsf{obs}(O \uplus \{\!\!\{\mathsf{O}(l)\}\!\!\}) * \mathsf{locked}(l, \mathit{Wt}, \mathit{Ot}) * \mathsf{I}(l)(\mathit{Wt}, \mathit{Ot})) \mathrel{-\!\!*} Q(\mathsf{tt})) \\
\mathsf{wp}_n(\mathsf{waiting4lock}(e), Q) = \mathsf{wp}_n(\mathsf{acquire}(e), Q) \\
\mathsf{wp}_n(\mathsf{release}(e), Q) = \exists \mathit{Wt}, \mathit{Ot}, O, l.\ (\mathsf{locked}(l, \mathit{Wt}, \mathit{Ot}) * \mathsf{I}(l)(\mathit{Wt}, \mathit{Ot}) * \mathsf{obs}(O \uplus \{\!\!\{\mathsf{O}(l)\}\!\!\}) \wedge \\
\quad \mathsf{A}(l) = \llbracket e \rrbracket) * ((\mathsf{lock}(l) * \mathsf{obs}(O)) \mathrel{-\!\!*} Q(\mathsf{tt})) \\
\mathsf{wp}_n(\mathsf{new_cvar}, Q) = \forall z.\ \exists r.\ \mathsf{ucond}(((z, r, z), (0, []), (0, []), \{\!\!\}\!\!\})) \mathrel{-\!\!*} Q(z) \\
\mathsf{wp}_n(\mathsf{wait}(e_1, e_2), Q) = \exists \mathit{Wt}, \mathit{Ot}, O, v, l.\ (\mathsf{cond}(v) * \mathsf{locked}(l, \mathit{Wt}, \mathit{Ot}) * \\
\quad \mathsf{I}(l)(\mathit{Wt} \uplus \{\!\!\{\mathsf{A}(v)\}\!\!\}, \mathit{Ot}) * \mathsf{obs}(O \uplus \{\!\!\{\mathsf{O}(l)\}\!\!\}) \wedge \mathsf{L}(v) = \mathsf{A}(l) \wedge \mathsf{O}(v) \prec O \wedge \mathsf{O}(l) \prec O \uplus \mathsf{M}'(v) \wedge \\
\quad \mathsf{enoughObs}(v, \mathit{Wt} \uplus \{\!\!\{\mathsf{A}(v)\}\!\!\}, \mathit{Ot}) \wedge \mathsf{A}(v) = \llbracket e_1 \rrbracket \wedge \mathsf{A}(l) = \llbracket e_2 \rrbracket) * \\
\quad (\forall \mathit{Wt}', \mathit{Ot}'.\ (\mathsf{cond}(v) * \mathsf{locked}(l, \mathit{Wt}', \mathit{Ot}') * \mathsf{I}(l)(\mathit{Wt}', \mathit{Ot}') * \\
\quad \mathsf{obs}(O \uplus \{\!\!\{\mathsf{O}(l)\}\!\!\} \uplus \mathsf{M}'(v)) * \mathsf{M}(v)) \mathrel{-\!\!*} Q(\mathsf{tt})) \\
\mathsf{wp}_n(\mathsf{waiting4cvar}(e_1, e_2), Q) = \exists O, v, l.\ (\mathsf{cond}(v) * \mathsf{lock}(l) * \mathsf{obs}(O) \wedge \\
\quad \mathsf{O}(v) \prec O \wedge \mathsf{O}(l) \prec O \uplus M' \wedge \mathsf{L}(v) = \mathsf{A}(l) \wedge \mathsf{A}(v) = \llbracket e_1 \rrbracket \wedge \mathsf{A}(l) = \llbracket e_2 \rrbracket) * \\
\quad (\forall \mathit{Wt}, \mathit{Ot}.\ (\mathsf{cond}(v) * \mathsf{locked}(l, \mathit{Wt}, \mathit{Ot}) * \mathsf{I}(l)(\mathit{Wt}, \mathit{Ot}) * \\
\quad \mathsf{obs}(O \uplus \{\!\!\{\mathsf{O}(l)\}\!\!\} \uplus \mathsf{M}'(v)) * \mathsf{M}(v)) \mathrel{-\!\!*} Q(\mathsf{tt})) \\
\mathsf{wp}_n(\mathsf{notify}(e), Q) = \exists \mathit{Wt}, \mathit{Ot}, O, v.\ (\mathsf{cond}(v) * \mathsf{locked}(\mathsf{L}(v), \mathit{Wt}, \mathit{Ot}) * \\
\quad \mathsf{obs}(O \uplus (0 < \mathit{Wt}(\mathsf{A}(v))\ ?\ \mathsf{M}'(v) : \{\!\!\}\!\!\})) * (\mathit{Wt}(\mathsf{A}(v)) = 0 \vee \mathsf{M}(v)) \wedge \mathsf{A}(v) = \llbracket e \rrbracket) * \\
\quad ((\mathsf{cond}(v) * \mathsf{locked}(\mathsf{L}(v), \mathit{Wt} - \{\!\!\{\mathsf{A}(v)\}\!\!\}, \mathit{Ot}) * \mathsf{obs}(O)) \mathrel{-\!\!*} Q(\mathsf{tt})) \\
\mathsf{wp}_n(\mathsf{notifyAll}(e), Q) = \exists \mathit{Wt}, \mathit{Ot}, O, v, l.\ (\mathsf{cond}(v) * \mathsf{locked}(l, \mathit{Wt}, \mathit{Ot}) * (\mathop{*}\limits_{i:=1}^{\mathit{Wt}(v)} \mathsf{M}(v)) \wedge \\
\quad \mathsf{M}'(v) = \{\!\!\}\!\!\} \wedge \mathsf{A}(v) = \llbracket e \rrbracket \wedge \mathsf{L}(v) = \mathsf{A}(l)) * ((\mathsf{cond}(v) * \mathsf{locked}(l, \mathit{Wt}[\mathsf{A}(v) := 0], \mathit{Ot})) \mathrel{-\!\!*} Q(\mathsf{tt})) \\
\mathsf{wp}_n(\mathsf{let}(x, c_1, c_2), Q) = \mathsf{wp}_{n-1}(c_1, \lambda z.\ \mathsf{wp}_{n-1}(c_2[z/x], Q)) \\
\mathsf{wp}_n(\mathsf{fork}(c), Q) = \exists O_1, O_2.\ \mathsf{obs}(O_1 \uplus O_2) * (\mathsf{obs}(O_1) \mathrel{-\!\!*} Q(\mathsf{tt})) * (\mathsf{obs}(O_2) \mathrel{-\!\!*} \\
\quad \mathsf{wp}_{n-1}(c, \lambda_.\ \mathsf{obs}(\{\!\!\}\!\!\}))) \\
\mathsf{wp}_n(\mathsf{if}(c, c_1, c_2), Q) = \mathsf{wp}_{n-1}(c, (\lambda z.\ 0 < z\ ?\ \mathsf{wp}_{n-1}(c_1, Q) : \mathsf{wp}_{n-1}(c_2, Q))) \\
\mathsf{wp}_n(\mathsf{while}(c, c_1), Q) = \mathsf{wp}_{n-1}(c, (\lambda z.\ z \leqslant 0\ ?\ Q(\mathsf{tt}) : \\
\quad \mathsf{wp}_{n-1}(c_1, (\lambda_.\ \mathsf{wp}_{n-1}(\mathsf{while}(c, c_1), Q)))))
\end{array}
$$

Figure 41 Weakest precondition, where tt stands for 0 (part one of two).

$$\begin{array}{l}
\mathsf{wp}_{n,pt}(\mathsf{nop}) = \\
\boxed{\text{as g\_initl}} \\
\quad (\exists\, Wt, Ot, O, a, r, I.\ \mathsf{ulock}(((a,r,a),(0,[]),(0,[]),\{\!\!\}),\ Wt, Ot) * pt(I)(Wt, Ot) * \mathsf{obs}(O) * \\
\qquad ((\mathsf{lock}(((a,r,a), I, (0,[]), \{\!\!\})) * \mathsf{obs}(O)) \mathrel{-\!\!*} Q(\mathsf{tt}))) \vee \\
\boxed{\text{as g\_initc}} \\
\quad (\exists\, Wt, Ot, a, r, l, M, M'.\ \mathsf{ucond}((a,r,a),(0,[]),(0,[]),\{\!\!\}) * \mathsf{ulock}(l,\ Wt, Ot) * \\
\qquad ((\mathsf{cond}((a,r,\mathsf{A}(l)),(0,[]), M, M') * \mathsf{ulock}(l,\ Wt, Ot)) \mathrel{-\!\!*} Q(\mathsf{tt}))) \vee \\
\boxed{\text{as g\_load}} \\
\quad (\exists v, l.\ (\mathsf{cond}(v) * \mathsf{ulock/locked}(l,\ Wt, Ot) * \mathsf{obs}(O) \wedge \mathsf{L}(v){=}\mathsf{A}(l)) * \\
\qquad ((\mathsf{cond}(v) * \mathsf{ulock/locked}(l,\ Wt, Ot{\uplus}\{\!\{\mathsf{A}(v)\}\!\}) * \mathsf{obs}(O{\uplus}\{\!\{\mathsf{O}(v)\}\!\})) \mathrel{-\!\!*} Q(\mathsf{tt}))) \\
\boxed{\text{as g\_discharge}} \\
\quad (\exists v, l.\ (\mathsf{cond}(v) * \mathsf{ulock/locked}(l,\ Wt, Ot) * \mathsf{obs}(O) \wedge \mathsf{enoughObs}(v,\ Wt, Ot{-}\{\!\{\mathsf{A}(v)\}\!\}) \wedge \\
\quad \mathsf{L}(v){=}\mathsf{A}(l)) * ((\mathsf{cond}(v) * \mathsf{ulock/locked}(l,\ Wt, Ot{-}\{\!\{\mathsf{A}(v)\}\!\}) * \mathsf{obs}(O{-}\{\!\{\mathsf{O}(v)\}\!\})) \mathrel{-\!\!*} Q(\mathsf{tt}))) \vee \\
\boxed{\text{as g\_new\_ctr}}\ (\forall gv.\ \mathsf{ctr}(gv, 0) \mathrel{-\!\!*} Q(\mathsf{tt})) \vee \\
\boxed{\text{as g\_inc}}\ (\exists n, gv.\ \mathsf{ctr}(gv, n) * ((\mathsf{ctr}(gv, n{+}1) * \mathsf{tic}(gv)) \mathrel{-\!\!*} Q(\mathsf{tt}))) \vee \\
\boxed{\text{as g\_dec}}\ (\exists n, gv.\ \mathsf{ctr}(gv, n) * \mathsf{tic}(gv) * (\mathsf{ctr}(gv, n{-}1) \mathrel{-\!\!*} Q(\mathsf{tt})))
\end{array}$$

Figure 42 Weakest precondition (part two of two).

Proof. By induction on n and case analysis of c. ◀

▶ **Lemma 20** (Weakest Precondition of Wait).

$$\begin{array}{l}
\forall n, e_1, e_2, \xi, p, O, g. \quad p, O, g \models \mathsf{wpcx}_n(\mathsf{wait}(e_1, e_2), \xi) \Rightarrow \\
\exists p_1, p_2, g_1, g_2, O_1, v, l,\ Wt, Ot.\ p{=}p_1{\uplus}p_2 \wedge O{=}O_1{\uplus}\{\!\{\mathsf{O}(l)\}\!\} \wedge g{=}g_1{\uplus}g_2 \wedge \\
\mathsf{A}(v) = [\![e_1]\!] \wedge \mathsf{A}(l) = [\![e_2]\!] \wedge p_1(l){=}\mathsf{locked}(Wt, Ot) \wedge p_1(v){=}\mathsf{cond} \wedge \\
p_2, \varnothing, g_2 \models \mathsf{I}(l)(Wt{\uplus}\{\!\{\mathsf{A}(v)\}\!\}, Ot) \wedge \mathsf{O}(v) \prec O_1 \wedge \mathsf{O}(l) \prec O_1{\uplus}\mathsf{M}'(v) \wedge \mathsf{L}(v){=}\mathsf{A}(l) \wedge \\
\mathsf{enoughObs}(v,\ Wt{\uplus}\{\!\{\mathsf{A}(v)\}\!\}, Ot) \wedge \\
p_1[l{:=}\mathsf{lock}], O_1, g_1 \models \mathsf{wpcx}_n(\mathsf{waiting4cvar}(e_1, e_2), \xi)
\end{array}$$

▶ **Lemma 21** (Weakest Precondition of Notify).

$$\begin{array}{l}
\forall n, e, \xi, p, O, g. \quad p, O, g \models \mathsf{wpcx}_n(\mathsf{notify}(e), \xi) \Rightarrow \exists p_1, p_M, g_1, g_M, O_1, v, l,\ Wt, Ot. \\
p{=}p_1{\uplus}p_M \wedge g{=}g_1{\uplus}g_M \wedge O{=}O_1{\uplus}(0{<}\,Wt(\mathsf{A}(v))\ ?\ \mathsf{M}'(v) : \{\!\!\}) \\
\wedge\ \mathsf{A}(v) = [\![e]\!] \wedge \mathsf{L}(v) = \mathsf{A}(l) \wedge p_1(v){=}\mathsf{cond} \wedge p_1(l){=}\mathsf{locked}(Wt, Ot) \wedge \\
(0{<}\,Wt(\mathsf{A}(v))\ ?\ p_M, \varnothing, g_M \models \mathsf{M}(v) : (p_M{=}\mathbf{0} \wedge g_M{=}\mathbf{0})) \wedge \\
p_1[l{:=}\mathsf{locked}(Wt{-}\{\!\{\mathsf{A}(v)\}\!\}, Ot)], O_1, g_1 \models \mathsf{wpcx}_n(\mathsf{tt}, \xi)
\end{array}$$

▶ **Lemma 22** (Weakest Precondition of waiting4cvar).

$$\begin{array}{l}
\forall n, e_1, e_2, \xi, p, O, g. \quad p, O, g \models \mathsf{wpcx}_n(\mathsf{waiting4cvar}(e_1, e_2), \xi) \Rightarrow \\
\exists v, l.\ p(v){=}\mathsf{cond} \wedge (p(l){=}\mathsf{lock} \vee \exists\, Wt, Ot.\ p(l){=}\mathsf{locked}(Wt, Ot)) \wedge \mathsf{L}(v){=}\mathsf{A}(l) \wedge \mathsf{O}(v) \prec O \\
\wedge\ \mathsf{O}(l) \prec O{\uplus}\mathsf{M}'(v) \wedge \mathsf{A}(v) = [\![e_1]\!] \wedge \mathsf{A}(l) = [\![e_2]\!] \wedge \\
\forall p_M, g_M.\ p_M, \varnothing, g_M \models \mathsf{M}(v) \Rightarrow p{\uplus}p_M, O{\uplus}\mathsf{M}'(v), g{\uplus}g_M \models \mathsf{wpcx}_n(\mathsf{waiting4lock}(e_2), \xi)
\end{array}$$

▶ **Lemma 23** (Weakest Precondition of g_discharge).

$$\begin{array}{l}
\forall n, \xi, p, O, g. \quad p, O, g \models \mathsf{wpcx}_n(\text{g\_discharge}, \xi) \Rightarrow \\
\exists O_1,\ Wt, Ot, v, l.\ O{=}O_1{\uplus}\{\!\{\mathsf{O}(v)\}\!\} \wedge p(l){=}\mathsf{locked}(Wt, Ot) \wedge p(v){=}\mathsf{cond} \wedge \\
\mathsf{enoughObs}(v,\ Wt, Ot{-}\{\!\{\mathsf{A}(v)\}\!\}) \wedge \mathsf{L}(v){=}\mathsf{A}(l) \wedge \\
p[l{:=}\mathsf{locked}(Wt, Ot{-}\{\!\{\mathsf{A}(v)\}\!\})], O_1, g \models \mathsf{wpcx}_n(\mathsf{tt}, \xi)
\end{array}$$

▶ **Lemma 24** (Weakest Precondition of fork).

$$\begin{array}{l}\forall n, c, \xi, p, O, g.\quad p, O, g \models \mathsf{wpcx}_n(\mathsf{fork}(c), \xi) \Rightarrow \\ \exists p_1, p_2, g_1, g_2, O_1, O_2.\ p{=}p_1 \uplus p_2 \wedge g{=}g_1 \uplus g_2 \wedge O{=}O_1 \uplus O_2 \wedge \\ p_1, O_1, g_1 \models \mathsf{wpcx}_n(\mathsf{tt}, \xi) \wedge p_2, O_2, g_2 \models \mathsf{wp}_{n-1}(c, \lambda\_.\mathsf{obs}(\{\!\!\}))\end{array}$$

▶ **Lemma 25** (Frame in Weakest Precondition).

$$\forall n, c, Q, F, p, \tilde{O}, g.\quad p, \tilde{O}, g \models \mathsf{wp}_n(c, Q) * F \Rightarrow \forall n' {\leqslant} n.\ p, \tilde{O}, g \models \mathsf{wp}_{n'}(c, (\lambda z.\ Q(z) * F))$$

Proof. By induction on n and case analysis of c. ◀

D.4 Correctness of Commands

We define *correctness of commands*, as shown in Definition 26, ensuring that each proposed proof rule, where $\mathsf{correct}_{pt}(P, c, Q)$ is abbreviated as $\{P\}\ c\ \{Q\}$, respects the definition of the weakest precondition. Having this definition we prove the proposed proof rules, ensuring deadlock freedom of importer channels, as well as some other necessary proof rules shown in Theorems 27, 28, and 29.

▶ **Definition 26** (Correctness of Commands). *A command is correct w.r.t a precondition P and a postcondition Q if and only if P implies the weakest precondition of that command w.r.t Q.*

$$\mathsf{correct}_{pt}(P, c, Q) \Leftrightarrow \forall n.\ P \Rightarrow \mathsf{wp}_{n,pt}(c, Q)$$

▶ **Theorem 27** (Rule Sequential Composition).

$$\mathsf{correct}(P, c_1, Q) \wedge (\forall z.\ \mathsf{correct}(Q(z), c_2[z/x], R)) \Rightarrow \mathsf{correct}(P, \mathsf{let}(x, c_1, c_2), R)$$

▶ **Theorem 28** (Rule Consequence).

$$\mathsf{correct}(P, c, Q) \wedge (P' \vdash P) \wedge (\forall z.\ Q(z) \vdash Q'(z)) \Rightarrow \mathsf{correct}(P', c, Q')$$

▶ **Theorem 29** (Rule Frame).

$$\mathsf{correct}(P, c, Q) \Rightarrow \mathsf{correct}(P * F, c, \lambda z.\ Q(z) * F)$$

As previously mentioned, since in this formalization ghost information is associated with lock and condition variable addresses via the lock and cond permission rather than via global function, we provide a new version of the proof rules, proposed in Section 3, regarding this formalization as shown in Figure 43.

D.5 Validity of a Configuration

We define *validity of a configuration*, shown in Definition 30, and prove that 1) starting from a valid configuration, all the subsequent configurations of the execution are also valid (Theorem 31), 2) a valid configuration is not deadlocked (Theorem 32), and 3) if a program c is verified by the proposed proof rules, where the verification starts from an empty bag of obligations and ends with such a bag too, then the initial configuration, where the heap is empty, denoted by $\mathbf{0}{=}\lambda\_.\varnothing$, and there is only one thread with the command c (and a context done), is a valid configuration (Theorem 34).

NewLock

$$\{\mathsf{true}\}\ \mathsf{newlock}\ \{\lambda a.\ \mathsf{ulock}(((a,r,a),(0,[]),(0,[]),\{\!\!\{\}\!\!\}),\{\!\!\{\}\!\!\},\{\!\!\{\}\!\!\})\}$$

InitLock

$$\{\mathsf{ulock}(((a,r,a),(0,[]),(0,[]),\{\!\!\{\}\!\!\}),\ Wt,Ot) * pt(I_{index},I_{args})(Wt,Ot) * \mathsf{obs}(O)\}\ \mathsf{nop}$$
$$\{\lambda\_.\ \mathsf{lock}(((a,r,a),(I_{index},I_{args}),(0,[]),\{\!\!\{\}\!\!\})) * \mathsf{obs}(O)\}$$

Acquire

$$\{\mathsf{lock}(l) * \mathsf{obs}(O) \wedge \mathsf{O}(l) \prec O \wedge \mathsf{A}(l) = a_l\}\ \mathsf{acquire}(a_l)$$
$$\{\lambda\_.\ \exists\, Wt,Ot.\ \mathsf{locked}(l,\ Wt,Ot) * \mathsf{I}(l)(\,Wt,Ot) * \mathsf{obs}(O \uplus \{\!\!\{\mathsf{O}(l)\}\!\!\})\}$$

Release

$$\{\mathsf{locked}(l,\ Wt,Ot) * \mathsf{I}(l)(\,Wt,Ot) * \mathsf{obs}(O \uplus \{\!\!\{\mathsf{O}(l)\}\!\!\}) \wedge \mathsf{A}(l) = a_l\}\ \mathsf{release}(a_l)$$
$$\{\lambda\_.\ \mathsf{lock}(l) * \mathsf{obs}(O)\}$$

NewCV

$$\{\mathsf{true}\}\ \mathsf{new\_cvar}\ \{\lambda a.\ \mathsf{ucond}(((a,r,a),(0,[]),(0,[]),\{\!\!\{\}\!\!\}))\}$$

InitCV

$$\{\mathsf{ucond}((a,r,a),(0,[]),(0,[]),\{\!\!\{\}\!\!\}) * \mathsf{ulock}(l,\ Wt,Ot)\}\ \mathsf{nop}$$
$$\{\lambda\_.\ \mathsf{cond}((a,r,\mathsf{A}(l)),(0,[]),(M_{index},M_{args}),M') * \mathsf{ulock}(l,\ Wt,Ot)\}$$

Wait

$$\{\mathsf{cond}(v) * \mathsf{locked}(l,\ Wt,Ot) * \mathsf{I}(l)(\,Wt \uplus \{\!\!\{\mathsf{A}(v)\}\!\!\},Ot) * \mathsf{obs}(O \uplus \{\!\!\{\mathsf{O}(l)\}\!\!\}) \wedge \mathsf{A}(v) = a_v \wedge \mathsf{A}(l) = a_l$$
$$\wedge\ \mathsf{A}(l){=}\mathsf{L}(v) \wedge \mathsf{O}(v){\prec}O \wedge \mathsf{O}(l){\prec}O{\uplus}\mathsf{M}'(v) \wedge \mathsf{enoughObs}(\mathsf{A}(v),\ Wt \uplus \{\!\!\{\mathsf{A}(v)\}\!\!\},Ot)\}\ \mathsf{wait}(a_v,a_l)$$
$$\{\lambda\_.\ \mathsf{cond}(v) * \mathsf{obs}(O \uplus \{\!\!\{\mathsf{O}(l)\}\!\!\} \uplus \mathsf{M}'(v)) * \exists\, Wt',Ot'.\ \mathsf{locked}(l,\ Wt',Ot') * \mathsf{I}(l)(\,Wt',Ot') * \mathsf{M}(v)\}$$

Notify

$$\{\mathsf{obs}(O \uplus (0{<}\,Wt(\mathsf{A}(v))\ ?\ \mathsf{M}'(v) : \{\!\!\{\}\!\!\})) * \mathsf{cond}(v) * \mathsf{locked}(l,\ Wt,Ot) * (\,Wt(\mathsf{A}(v)){=}0 \vee \mathsf{M}(v)) \wedge$$
$$\mathsf{A}(l){=}\mathsf{L}(v) \wedge \mathsf{A}(v) = a_v\}\ \mathsf{notify}(a_v)\ \{\lambda\_.\ \mathsf{obs}(O) * \mathsf{cond}(v) * \mathsf{locked}(l,\ Wt{-}\{\!\!\{\mathsf{A}(v)\}\!\!\},Ot)\}$$

NotifyAll

$$\{\mathsf{cond}(v) * \mathsf{locked}(l,\ Wt,Ot) * (\mathop{\ast}_{i:=1}^{Wt(\mathsf{A}(v))} \mathsf{M}(v)) \wedge \mathsf{M}'(v){=}\{\!\!\{\}\!\!\} \wedge \mathsf{A}(l){=}\mathsf{L}(v) \wedge \mathsf{A}(v) = a_v\}$$
$$\mathsf{notifyAll}(a_v)\ \{\lambda\_.\ \mathsf{cond}(v) * \mathsf{locked}(l,\ Wt[v{:=}0],Ot)\}$$

ChargeObligation

$$\{\mathsf{obs}(O) * \mathsf{cond}(v) * \mathsf{ulock/locked}(l,\ Wt,Ot) \wedge \mathsf{A}(l){=}\mathsf{L}(v)\}\ \mathsf{nop}$$
$$\{\lambda\_.\ \mathsf{obs}(O \uplus \{\!\!\{\mathsf{O}(v)\}\!\!\}) * \mathsf{cond}(v) * \mathsf{ulock/locked}(l,\ Wt,Ot \uplus \{\!\!\{\mathsf{A}(v)\}\!\!\})\}$$

DischargeObligation

$$\{\mathsf{obs}(O) * \mathsf{cond}(v) * \mathsf{ulock/locked}(l,\ Wt,Ot) \wedge \mathsf{enoughObs}(\mathsf{A}(v),\ Wt,Ot{-}\{\!\!\{\mathsf{A}(v)\}\!\!\})$$
$$\wedge\ \mathsf{A}(l){=}\mathsf{L}(v)\}\ \mathsf{nop}\ \{\lambda\_.\ \mathsf{obs}(O{-}\{\!\!\{\mathsf{O}(v)\}\!\!\}) * \mathsf{ulock/locked}(l,\ Wt,Ot{-}\{\!\!\{\mathsf{A}(v)\}\!\!\})\}$$

Figure 43 Proof rules verifying deadlock-freedom of importer monitors, where ghost information is associated with lock and channel addresses via lock and cond permissions.

▶ **Definition 30** (Validity of a Configuration). *A configuration (t, h) is valid for n steps, denoted by $\mathsf{valid}_n(t, h)$, if there exist a list of* augmented threads *A, consisting of the identification (id), the program (c), the context (ξ), the permission heap (p), the ghost resource heap (g) and the obligations (O) associated with each thread; a list of lock-invariant pairs $Linv$, storing the locks which are not held along with their invariants; three permission heaps p_i (associated with the invariants of the locks which are not held), p_l (the part of the permission heap which is leaked), and p_A (the union of all permission heaps in A and p_i as well as p_l); three ghost resource heaps g_i (associated with the invariants of the locks which are not held), g_l (the part of the ghost resource heap which is leaked), g_A (the union of all ghost resource heaps in A and g_i as well as g_l); and $locs$ (the set of locations for which a memory has been allocated), such that all of the following conditions hold:*

1. $\forall id, c, \xi.\ t(id) = (c; \xi) \Leftrightarrow \exists p, O, g.\ (id, c, \xi, p, O, g) \in A$
2. $\forall (id_1, c_1, \xi_1, p_1, O_1, g_1) \in A, (id_2, c_2, \xi_2, p_2, O_2, g_2) \in A.\ id_1 = id_2 \Rightarrow$ $(id_1, c_1, \xi_1, p_1, O_1, g_1) = (id_2, c_2, \xi_2, p_2, O_2, g_2)$
3. $p_A = p_i \uplus p_l \uplus \biguplus_{(id,c,\xi,p,O,g)\in A} p \quad \wedge \quad g_A = g_i \uplus g_l \uplus \biguplus_{(id,c,\xi,p,O,g)\in A} g$
4. $\forall l_1, l_2.\ p_A(l_1) \neq \varnothing \wedge p_A(l_2) \neq \varnothing \wedge \mathsf{A}(l_1) = \mathsf{A}(l_1) \Rightarrow l_1 = l_2$
5. $\forall l.\ p_A(l) \neq \varnothing \Leftrightarrow l \in locs$
6. $\mathsf{pheap\_heap}(p_A, h)$
7. $p_i, \varnothing, g_i \models \mathop{\ast}_{(l,inv)\in Linv} inv$
8. $p_A(l) = \mathsf{lock} \wedge \neg\mathsf{held}_h(l) \Rightarrow (l, \mathsf{I}(l)(Wt_{l,A}, Ot_{l,A})) \in Linv$
9. $(l, inv) \in Linv \Rightarrow p_A(l) = \mathsf{lock} \wedge \neg\mathsf{held}_h(l)$
10. $\forall o \in O_A.\ \exists l.\ \mathsf{O}(l){=}o \wedge (p_A(l){=}\mathsf{cond} \vee p_A(l){=}\mathsf{lock} \vee \exists\, Wt, Ot.\ p_A(l){=}\mathsf{locked}(Wt, Ot))$
11. $p_A(l){=}\mathsf{ulock}(Wt, Ot) \vee p_A(l){=}\mathsf{locked}(Wt, Ot) \Rightarrow\ Wt{=}\,Wt_{l,A} \wedge Ot{=}Ot_{l,A}$
12. $p_A(l){=}\mathsf{lock} \vee p_A(l){=}\mathsf{ulock}(Wt, Ot) \vee p_A(l){=}\mathsf{locked}(Wt, Ot){\Rightarrow}\mathsf{held}_h(l) \Rightarrow l \in O_A$
13. $\forall (id, c, \xi, p, O, g){\in}A.$
    a. $p, O, g \models \mathsf{wpcx}_n(c, \xi)$
    b. $c = \mathsf{waiting4cvar}(e_1, e_2) \Rightarrow \mathsf{enoughObs}(\llbracket e_1\rrbracket, Wt_{\llbracket e_2\rrbracket,A}, Ot_{\llbracket e_2\rrbracket,A})$

where

- $O_A = \biguplus_{(id,c,\xi,p,O,g)\in A} O$
- $Ot_{l,A} = \lambda v.\ \mathsf{L}(v){=}l\ ?\ O_A(v) : 0$ *, and* $Wt_{l,A} = \biguplus_{(id,c,\xi,p,O,g)\in A \wedge \mathsf{waiting\_for}_h(c)=v \wedge \mathsf{L}(v)=l} \{\!\!\{v\}\!\!\}$
- $\mathsf{waiting\_for}_h(c)$ *returns the object for which c is waiting, if any, i.e.*
  $$\mathsf{waiting\_for}_h(c, h) = \begin{cases} \llbracket e_1\rrbracket & \textit{if } c{=}\mathsf{waiting4cond}(e_1, e_2) \\ \llbracket e\rrbracket & \textit{if } c{=}\mathsf{waiting4lock}(e) \wedge h(\llbracket e\rrbracket){\neq}1 \\ \varnothing & \textit{otherwise} \end{cases}$$
- $\mathsf{held}_h(l)$ *returns true if and only if the lock l is held, i.e.* $\mathsf{held}_h(l) \Rightarrow h(\mathsf{A}(l)) \neq 1$

Each item in this definition ensures some properties as follows: (1) and (2) ensure that the list of augmented threads A is correctly associated with the thread tables t, (3) ensures that the union of all permission heaps as well as the union of all ghost resource heaps in A are defined, (4) ensures that any two allocated locations which have the same address are equal, (5) *locs* is the set of locations for which a memory has been allocated (having this set it is possible to map addresses to their ghost information) (6) ensures that p_A corresponds to the concrete heap h, (7) ensures that p_i and g_i model the separating conjunction of the invariants of the locks which are not held, and these invariants do not assert any obligation, (8) ensures that any lock which is not held along with its invariant exist in Linv, (9) ensures that the locks in Linv are not held, (10) ensures that any obligation in A is associated with the address of a condition variable or a lock which has been initialized, (11) ensures that the parameters Wt and Ot stored in the permissions ulock and locked of any lock store the total number of waiting threads and obligations of the condition variables associated with that lock, respectively, (12) ensures that any lock l which is held exists in the union of the obligations of A, (13-a) the permission heap, the obligations, and the ghost resource heap of each thread model the weakest precondition of the command of that thread w.r.t. the postcondition in which there is no obligation, and (13-b) for any condition variable for which a thread is waiting the invariant $\mathsf{enoughObs}$ holds.

▶ **Theorem 31** (Small Steps Preserve Validity of Configurations)**.** *Each step of the execution preserves validity of configurations.*

$$(t,h) \rightsquigarrow (t',h') \land \mathsf{valid}_{n+1}(t,h) \Rightarrow \mathsf{valid}_n(t',h')$$

Proof. By case analysis of the small step relation $\rightsquigarrow$. ◀

▶ **Theorem 32** (A Valid Configuration Is Not Deadlocked)**.** *If a valid configuration has some threads then there exists a thread in this configuration neither waiting for a condition variable nor a lock.*

$$\mathsf{valid}_n(t,h) \land \exists id.\ t(id) \neq \varnothing \Rightarrow \exists id'.\ \mathsf{waiting\_for}(\mathsf{fst}(t(id'))) = \varnothing$$

Proof. We assume that all threads in t are waiting for a condition variable or a lock. Since (t,h) is a valid configuration there exists a valid list of augmented threads A with a corresponding valid bag $G = \mathsf{valid\_bag}(A)$, where $\mathsf{valid\_bag}$ maps any element (id, c, ξ, p, O, g) to an element $(\mathsf{waiting\_for}(c), O)$. By Lemma 33, we have $G{=}\{\!\!\{\}\!\!\}$, implying $A{=}\{\}$, implying $t{=}\mathbf{0}$ which contradicts the hypothesis of the theorem.

Note that in the definition of validity of a configuration we also keep track of all locations whose addresses are allocated, which makes it possible to provide the function R, mapping lock and condition variable addresses to their levels, for Lemma 33. The first hypothesis in this lemma is met by the constraint 13-a in the definition of validity of a configuration. Additionally, the second hypothesis in this lemma is met by the constraints 12 or 13-b, where the related thread is waiting for a lock or a condition variable, respectively. ◀

Lemma 33 ensures that in any state of the execution if all the desired invariants are respected then it is impossible that all threads are waiting for an object. In this lemma G is a bag of object-obligations pairs such that each element t of G is associated with a thread in a state of the execution, where the first element of t is associated with the object for which t is waiting and the second element is associated with the obligations of t.

▶ **Lemma 33** (A Valid Bag of Augmented Threads Is Not Deadlocked)**.**

$$\begin{array}{l} \forall\ G : \mathit{Bags}(\mathit{Addresses} \times \mathit{Bags}(\mathit{Addresses})), R : \mathit{Addresses} {\to} \mathit{Levels}. \\ (\forall (o,O){\in}G.\ o \prec O \land (\exists o', O'.\ (o', \{\!\!\{o\}\!\!\} {\uplus} O') \in G)) \Rightarrow G = \{\!\!\{\}\!\!\} \end{array}$$

Proof. By contradiction; assume that $\exists (o_m, O_1) \in G$ where $\neg\exists (o,O) \in G.\ R(o) < R(o_m)$. By H_2 we have $\exists o', O'.\ (o', \{\!\!\{o_m\}\!\!\} {\uplus} O') \in G$ and by H_1 we have $o' \prec \{\!\!\{o_m\}\!\!\} {\uplus} O'$, which contradicts minimality of the level of o_m. ◀

▶ **Theorem 34** (The Initial Configuration Is Valid)**.** *The initial configuration, consisting of an empty heap and a single thread whose program is verified by the proposed proof rules, is a valid configuration.*

$$\mathsf{correct}_{sp}(\mathsf{obs}(\{\!\!\{\}\!\!\}), c, \lambda\_.\mathsf{obs}(\{\!\!\{\}\!\!\})) \Rightarrow \forall n, id.\ \mathsf{valid}_n(\mathbf{0}[id{:=}c; \mathsf{done}], \mathbf{0})$$

$$\begin{array}{l}
\mathsf{ol}(m, r) ::= (m.l, r, m.l) \\
\mathsf{ov}(m, r) ::= (m.v, r, m.l) \\
\mathsf{l}(m, r) ::= (\mathsf{ol}(m, r), (\mathsf{linv}, [m]), (0, []), \{\!\!\{\}\!\!\}) \\
\mathsf{v}(m, r) ::= (\mathsf{ov}(m, r), (0, []), (\mathsf{M}, []), \{\!\!\{\mathsf{ov}(m, r)\}\!\!\})
\end{array}$$

$$\mathsf{mutex}(\mathsf{mutex}\ m, \mathsf{waitobj}\ o) = \mathsf{lock}(\mathsf{l}(m, \mathsf{R}(o){-}1)) * \mathsf{cond}(o) \land o = \mathsf{v}(m, \mathsf{R}(o))$$

$$\begin{array}{l}
\mathsf{pt}(\mathsf{linv}, [m]{\cdot}args) = \lambda Wt.\ \lambda Ot.\ \exists b, w.\ m.b \mapsto b * m.w \mapsto w \land Wt(m.v){=}w \land \\
\ (0 < b\ ?\ 0 < Ot(v) : Wt(v) = 0) \\
\mathsf{pt}(\mathsf{M}, args) = \lambda Wt.\ \lambda Ot.\ \mathsf{true}
\end{array}$$

Figure 44 Verification of the fair mutexes implementation shown in Figure 16 using the proof rules in Figure 43 (part one of two).

Proof. The goal is achieved because there are an augmented thread list $T{=}[(id, c, \mathsf{done}, \mathbf{0}, \{\!\!\{\}\!\!\}, \mathbf{0})]$, a list of lock-invariant pairs $\mathsf{Linv}{=}[]$, two permission heaps $p_i{=}\mathbf{0}$ and $p_l{=}\mathbf{0}$, and two ghost resource heaps $g_i{=}\mathbf{0}$ and $g_l{=}\mathbf{0}$, such that all the conditions in the definition of validity of configurations are met. ◀

D.6 An Example Proof

In this section we show how the program in Figure 16 can be verified using the proof rules in Figure 43, as shown in Figures 44 and 45[16].

[16] Note that for this program we assume a straightforward extension of the programming language with immutable structures, i.e. tuples with named components.

```
routine new_mutex(){
req : {true}
 $l$ := new_lock;
 $\{\mathsf{ulock}(((l, r{-}1, l), (0, []), (0, []), \{\!\![]\!\!\}), \{\!\![]\!\!\}, \{\!\![]\!\!\})\}$
 $v$ := new_cvar; nop; //Rule InitCV
 $\{\mathsf{ulock}(((l, r{-}1, l), (0, []), (0, []), \{\!\![]\!\!\}), \{\!\![]\!\!\}, \{\!\![]\!\!\}) * \mathsf{cond}((v, r, l), (0, []), (\mathsf{M}, []), \{\!\![(v, r, l)]\!\!\})\}$
 $m$ := mutex($l{:=}l, v{:=}v, b{:=}$new_int(1)$, w{:=}$new_int(1)); nop; //Rule InitLock
 $m$ ens : $\{\lambda m.\ \exists o.\ \mathsf{mutex}(m, o) \land \mathsf{R}(o){=}r\}\}$

routine enter_cs(mutex $m$){
req : $\{\mathsf{obs}(O) * \mathsf{mutex}(m, o) \land \mathsf{O}(o) \prec O\}$
 acquire($m.l$);
 // Let $l = \mathsf{l}(m, \mathsf{R}(o){-}1)$ and $ol = \mathsf{ol}(m, \mathsf{R}(o){-}1)$
 $\{\mathsf{obs}(O \uplus \{\!\![ol]\!\!\}) * \mathsf{mutex}(m, o) * \exists Wt, Ot.\ \mathsf{locked}(l, Wt, Ot) * \mathsf{pt}(\mathsf{linv}, [m])(Wt, Ot)\}$
 if($m.b$){
   $m.w := m.w{+}1$;
   $\{\mathsf{obs}(O \uplus \{\!\![ol]\!\!\}) * \mathsf{mutex}(m, o) * \exists Wt, Ot.\ \mathsf{locked}(l, Wt, Ot) * \mathsf{pt}(\mathsf{linv}, [m])(Wt \uplus \{\!\![m.v]\!\!\}, Ot)\}$
   wait($m.v, m.l$)
   $\{\mathsf{obs}(O \uplus \{\!\![ol, \mathsf{O}(o)]\!\!\}) * \mathsf{mutex}(m, o) * \exists Wt, Ot.\ \mathsf{locked}(l, Wt, Ot) * \mathsf{pt}(\mathsf{linv}, [m])(Wt, Ot)\}\}$
 else{ $m.b := 1$; nop; //Rule ChargeObligation
   $\{\mathsf{obs}(O \uplus \{\!\![ol, \mathsf{O}(o)]\!\!\}) * \mathsf{mutex}(m, o) * \exists Wt, Ot.\ \mathsf{locked}(l, Wt, Ot) * \mathsf{pt}(\mathsf{linv}, [m])(Wt, Ot)\}\}$;
 $\{\mathsf{obs}(O \uplus \{\!\![ol, \mathsf{O}(o)]\!\!\}) * \mathsf{mutex}(m, o) * \exists Wt, Ot.\ \mathsf{locked}(l, Wt, Ot) * \mathsf{pt}(\mathsf{linv}, [m])(Wt, Ot)\}$
 release($m.l$)
ens : $\{\mathsf{obs}(O \uplus \{\!\![\mathsf{O}(o)]\!\!\}) * \mathsf{mutex}(m, o)\}\}$

routine exit_cs(mutex $m$){
req : $\{\mathsf{obs}(O \uplus \{\!\![\mathsf{O}(o)]\!\!\}) * \mathsf{mutex}(m, o) \land \mathsf{O}(o) \prec O\}$
 acquire($m.l$);
 // Let $l = \mathsf{l}(m, \mathsf{R}(o){-}1)$ and $ol = \mathsf{ol}(m, \mathsf{R}(o){-}1)$
 $\{\mathsf{obs}(O \uplus \{\!\![ol, \mathsf{O}(o)]\!\!\}) * \mathsf{mutex}(m, o) * \exists Wt, Ot.\ \mathsf{locked}(l, Wt, Ot) * \mathsf{pt}(\mathsf{linv}, [m])(Wt, Ot)\}$
 $m.b := 0$;
 if($0{<}m.w$){ $m.w := m.w{-}1$; $m.b := 1$; notify($m.v$)
   $\{\mathsf{obs}(O \uplus \{\!\![ol]\!\!\}) * \mathsf{mutex}(m, o) * \exists Wt, Ot.\ \mathsf{locked}(l, Wt, Ot) * \mathsf{pt}(\mathsf{linv}, [m])(Wt, Ot)\}\}$
 else{ nop //Rule DischargeObligation
   $\{\mathsf{obs}(O \uplus \{\!\![ol]\!\!\}) * \mathsf{mutex}(m, o) * \exists Wt, Ot.\ \mathsf{locked}(l, Wt, Ot) * \mathsf{pt}(\mathsf{linv}, [m])(Wt, Ot)\}\}$;
 release($m.l$)
ens : $\{\mathsf{obs}(O) * \mathsf{mutex}(m, o)\}\}$
```

Figure 45 Verification of the fair mutexes implementation shown in Figure 16 using the proof rules in Figure 43 (part two of two).

Automated Large-Scale Multi-Language Dynamic Program Analysis in the Wild

Alex Villazón
Universidad Privada Boliviana, Bolivia
avillazon@upb.edu

Haiyang Sun
Università della Svizzera italiana, Switzerland
haiyang.sun@usi.ch

Andrea Rosà
Università della Svizzera italiana, Switzerland
andrea.rosa@usi.ch

Eduardo Rosales
Università della Svizzera italiana, Switzerland
rosale@usi.ch

Daniele Bonetta
Oracle Labs, United States
daniele.bonetta@oracle.com

Isabella Defilippis
Universidad Privada Boliviana, Bolivia
isabelladefilippis@upb.edu

Sergio Oporto
Universidad Privada Boliviana, Bolivia
sergiooporto@upb.edu

Walter Binder
Università della Svizzera italiana, Switzerland
walter.binder@usi.ch

Abstract

Today's availability of open-source software is overwhelming, and the number of free, ready-to-use software components in package repositories such as NPM, Maven, or SBT is growing exponentially. In this paper we address two straightforward yet important research questions: would it be possible to develop a tool to automate dynamic program analysis on public open-source software at a large scale? Moreover, and perhaps more importantly, would such a tool be useful? We answer the first question by introducing NAB, a tool to execute large-scale dynamic program analysis of open-source software in the wild. NAB is fully-automatic, language-agnostic, and can scale dynamic program analyses on open-source software up to thousands of projects hosted in code repositories. Using NAB, we analyzed more than 56K Node.js, Java, and Scala projects. Using the data collected by NAB we were able to (1) study the adoption of new language constructs such as JavaScript Promises, (2) collect statistics about bad coding practices in JavaScript, and (3) identify Java and Scala task-parallel workloads suitable for inclusion in a domain-specific benchmark suite. We consider such findings and the collected data an affirmative answer to the second question.

2012 ACM Subject Classification Software and its engineering → Dynamic analysis

Keywords and phrases Dynamic program analysis, code repositories, GitHub, Node.js, Java, Scala, promises, JIT-unfriendly code, task granularity

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2019.20

Category Tool Insights Paper

33rd European Conference on Object-Oriented Programming (ECOOP 2019).
Editor: Alastair F. Donaldson; Article No. 20; pp. 20:1–20:27
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Supplement Material ECOOP 2019 Artifact Evaluation approved artifact available at https://dx.doi.org/10.4230/DARTS.5.2.11

Acknowledgements This work has been supported by Oracle (ERO project 1332), Swiss National Science Foundation (scientific exchange project IZSEZ0_177215), Hasler Foundation (project 18012), and by a Bridging Grant with Japan (BG 04-122017).

1 Introduction

Analyzing today's large code repositories[1] has become an important research area for understanding and improving different aspects of modern software systems. Static and dynamic program analyses are complementary approaches to this end. Static program analysis is the art of reasoning about the behavior of computer programs without actually running them, which is useful not only in optimizing compilers for producing efficient code, but also for automatic error detection and other tools that help programmers [45]. Complementary to static analysis is *dynamic program analysis* (DPA), which employs runtime techniques (such as instrumentation and profiling) to explore the runtime behavior of applications under specific workload conditions and input.

In contrast to the large body of work on mining code repositories through static program analysis [39, 52, 37, 53, 38, 7, 11, 51], studies applying DPA to large public code repositories are scarce [35, 42]. Moreover, all such studies are limited at narrow, specific aspects of a particular programming language or framework, and none of them scales to the overwhelming amount of available projects that could potentially be analyzed.

In this paper, we tackle two basic yet important research questions: can we create a tool for automated DPA that can scale to the vast amount of available public open-source projects, and would such a tool be of practical interest?

Given the constant growth of the number of projects in public code repositories and the increasing popularity of different programming languages and runtimes – e.g., JavaScript/Node.js and the many languages targeting the Java Virtual Machine (JVM) – such a tool not only should be scalable, but should also be language-agnostic and resilient to malicious or buggy code. Such aspects already correspond to non-trivial technical challenges. Beyond these technical aspects, developing such a tool would require answering one more fundamental question: *how* should the tool execute code from repositories that are not designed to enable DPA?

Our answer to the question is pragmatic: the tool should *automatically* look for the available executable code in a repository, and *try* to execute anything that could potentially be executed. Such executable code could correspond to existing benchmarks (e.g., workloads defined by the developers via the Java Microbenchmark Harness (JMH) [48]) or software tests (e.g., defined in the default test entry of a Node.js project managed by Node Package Manager (NPM), or based on popular testing frameworks such as JUnit [63]). By replicating the process on the massive size of public repositories, such a tool should be able to identify a very high amount of (automatically) executable code.

With the tool, would running massive DPA on publicly available repositories be of any scientific interest? Our insight is that such a tool can be useful to collect statistics and evidence about code patterns and application characteristics that may benefit language designers, software system designers, and programming-language researchers at large.

[1] In this paper, the term *repository* denotes a code hosting site (such as GitHub, GitLab, or BitBucket), containing multiple *projects* (i.e., open-source code subjected to version control).

One concrete application for such a tool could be the study of the adoption (by a wide open-source community) of new programming-language constructs involving dynamic behavior (i.e., where pure static analysis cannot give any concrete insight). For example, it is currently unclear how the recently-introduced JavaScript Promise API [29] is being used by Node.js developers: while some recent research seems to suggest that developers are frequently mis-using the API [40, 1], its growing adoption by the Node.js built-in modules (e.g., Node.js' file-system module) could result in a more disciplined usage. In this context, a massive analysis of Node.js projects would be of great help to assess the adoption of the API, and to drive its future evolution. Measuring aspects such as the size of a so-called *promise chain* [1] (i.e., the number of promises that are linked together) requires DPA.

A second practical application of a tool for massive DPA is the study of problematic code patterns in dynamic languages. For example, several studies [12, 46, 17] focus on *JIT-unfriendly* code patterns [17], i.e., code that may obstacle dynamic optimizations performed by a Just-In-Time (JIT) compiler. While these studies have shown that such bad code patterns can impair the performance of modern language execution runtimes, none of them has investigated how common problematic coding practices are. Identifying such bad code patterns and assessing their use on the high number of open-source Node.js projects as well as the NPM modules they depend on could be very useful in practice, as it would provide a "bird's-eye view" over the quality of the NPM ecosystem. Similarly to the previous case, DPA is needed to identify such patterns in several runtime-dependent scenarios.

A final useful application for a massive DPA tool is the search for workloads suitable to conduct experimental evaluations. For many domain-specific evaluation needs (e.g., concurrency on the JVM), there is a lack of suitable benchmarks, and creating new benchmark suites requires non-trivial effort for finding proper workload candidates [69]. For example, existing general-purpose benchmark suites including Java and Scala benchmarks (e.g., DaCapo [64] and ScalaBench [56]) have only few task-parallel workloads [5, 58, 61]. Ideally, a fully automated system could discover relevant workloads by massively analyzing the open-source projects in public code repositories. Such a system could find real-world concurrent applications that spawn numerous tasks of diverse granularities, suitable for inclusion in a benchmark suite targeting concurrency on the JVM. Similarly to our previous examples, profiling all parallel tasks spawned by an application and measuring each task's granularity requires DPA.

To support the diversity of DPA scenarios that we have described at the scale of public code repositories, in this paper we present NAB,[2] a novel, distributed, container-based infrastructure for massive DPA on code repositories hosting open-source projects, which may be implemented in different programming languages. NAB resorts to containerization for efficient sandboxing, for the parallelization of DPA execution, and for simplifying the deployment on clusters or in the Cloud. Sandboxing is important to isolate the underlying execution environment and operating system, since NAB executes unverified projects that may contain buggy or even harmful code. Also, parallelizing DPA execution is an important feature for massive analysis, as sequential analysis of massive code repository would take prohibitive time. NAB features both crawler and analyzer components, which are deployed in lightweight containers and can be replicated. They are governed by NAB's coordination component, which ensures scalability and elasticity, facilitating the provisioning of new container images through simple configuration settings. NAB includes a plugin mechanism for the integration of existing DPA tools and the selection of different build systems, testing frameworks, and runtimes for multi-language support.

[2] NAB's recursive name stands for "NAB is an Analysis Box".

Our work makes the following contributions:

- We present NAB, a novel, distributed infrastructure for custom DPA in the wild. To the best of our knowledge, NAB is the first scalable, container-based infrastructure for automated, massive DPA on open-source projects, supporting multiple programming languages.
- We present a novel analysis to collect runtime statistics about the usage of promises in Node.js projects, which focuses on the size of promise chains. The analysis sheds light on the usage of the Promise API in open-source Node.js projects. We present an implementation of the analysis called Deep-Promise, which relies on NodeProf [62], an open-source dynamic instrumentation framework for Node.js based on GraalVM [68].
- We conduct a large-scale study on Node.js projects, searching for JIT-unfriendly code that may impair the effectiveness of optimizations performed by the JavaScript engine's JIT compiler. To this end, we apply JITProf [17], an open-source DPA.
- We perform a large-scale analysis on Java and Scala projects searching for task-parallel workloads suitable for inclusion in a benchmark suite. To this end, we apply tgp [54], an open-source DPA for task-granularity profiling on the JVM.

Our work confirms (1) that NAB can be used for applying DPA massively on public code repositories, and (2) that the large-scale analyses enabled by NAB provide insights that are of practical interest, thus affirmatively answering to our research questions.

This paper is structured as follows. Section 2 describes NAB's architecture and implementation. Section 3 details the experimental setup for our studies. Sections 4, 5, and 6 present the results of our three case studies. Section 7 discusses important aspects such as safety, extensibility, scalability, as well as the limitations of NAB. We discuss related work in Section 8 and conclude in Section 9.

2 NAB

This section presents our tool for massive DPA, NAB. First, we introduce NAB's architecture (Section 2.1); then, we describe how NAB's main components interact (Section 2.2). We continue by detailing the crawling (Section 2.3) and analysis (Section 2.4) process. Finally, we describe NAB's plugin mechanism supporting different DPA tools (Section 2.5), as well as the implementation technologies used to support containerization (Section 2.6).

2.1 Architecture

At its core, NAB features a *microservice* architecture based on a *master-worker* pattern relying on a *publish-subscribe* communication layer, allowing asynchronous events to be exchanged between its internal components. Figure 1 depicts the overall NAB architecture based on Docker containers [22]. NAB uses existing containerized services (marked in gray) and introduces four new components, three of them running in containers: NAB-Crawler, NAB-Analyzer, and NAB-Master; as well as one external service, NAB-Dashboard.

The NAB-Crawler instances are responsible for mining and crawling code repositories, collecting metadata that allows making a decision on which projects to analyze. The NAB-Analyzer instances are responsible for downloading the code, applying some filtering and eventually running the DPA tool. The results generated by the DPA (such as profiles containing various dynamic metrics) are stored in a NoSQL MongoDB [26] database. NAB provides a *plugin* mechanism to integrate different DPA tools in NAB-Analyzer instances.

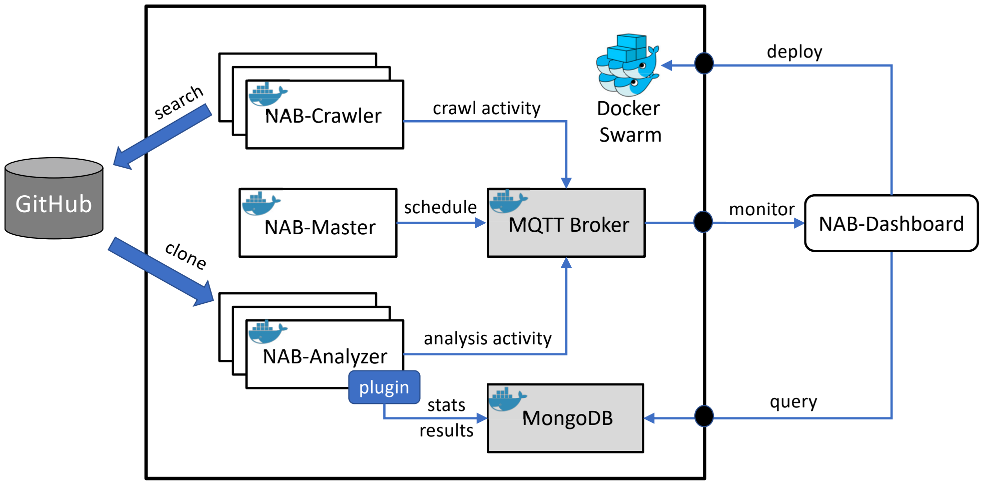


Figure 1 Overview of NAB. The three core NAB containerized services (NAB-Master, NAB-Crawler, and NAB-Analyzer) are shown as white boxes inside the Docker Swarm, whereas existing containerized services are marked in gray. A third-party DPA tool (not shown) is invoked by NAB-Analyzer through a plugin and generates profiles.

NAB-Master orchestrates the distribution of crawling and DPA activities with NAB-Crawler and NAB-Analyzer instances. Finally, NAB-Dashboard is responsible for the deployment of the NAB components through the Docker Swarm [23] orchestration service and monitors the progress of an ongoing DPA.

NAB communication service is based on MQTT (Message Queuing Telemetry Transport), an ISO-standard, lightweight publish-subscribe protocol [14]. The MQTT Broker is the core communication service that receives subscription requests from NAB components to different *topics* and redistributes messages that are published on such topics. This ensures that the communication between NAB components is implemented in a loosely-coupled manner, as they only need to agree on specific topics without knowing each other beforehand.

Some NAB services expose ports (represented as black circles in Figure 1) to allow interaction with them from outside the Docker Swarm. For example, NAB-Dashboard uses the exposed ports to access the MQTT communication service and subscribes to all topics used by the other NAB components. This information allows monitoring the distributed execution of a DPA. Similarly, the NAB-Dashboard can query the MongoDB database to collect execution results of previous runs, e.g., for post-mortem performance analysis, for visualizing the detailed distributed execution, for finding load imbalances, or for debugging purposes.

To improve scalability, NAB can be configured to use several MQTT Brokers through HAProxy [19] (a high-performance TCP load balancer that distributes the exchanged MQTT messages) as well as multiple distributed and replicated MongoDB instances through a *mongos* router [25] coordinating a MongoDB Shard [27] (not shown in Figure 1).

2.2 Interactions between NAB Components

When NAB is started, NAB-Dashboard initializes (through Docker Swarm) all the containerized services shown in Figure 1, passing user-defined specifications about the analysis to be executed to NAB-Master. Such specifications depend on the DPA to use, as well as on the code repository where crawling should be performed. While NAB provides supports for crawling different repositories (e.g., GitHub [24], GitLab [16], Bitbucket [2]), in this paper we focus on GitHub, which is the one providing the most advanced search API.

When performing a DPA on GitHub, the specifications sent to NAB-Master include crawling and analysis settings, the DPA plugin configuration, and a set of user-defined time intervals (which restricts the crawling to specific ranges of dates). The time intervals refer to the dates where the projects had the most recent activity (i.e., commits made to any branch). Since Docker Swarm does not provide a mechanism to enforce a given order for starting containers, we implement a synchronization mechanism such that all NAB components wait for the MQTT Broker container to be ready, since it is the core communication-related component, responsible for receiving and dispatching messages.

NAB-Master handles four lists: `timeIntervals` (storing the time intervals to crawl and initialized according to the specifications set by the user), `projects` (storing the name of the projects to analyze, initially empty), `crawlers` and `analyzers` (storing the IDs of available and ready NAB-Crawlers and NAB-Analyzer instances, respectively, both initially empty). During the initialization, NAB-Master subscribes to three topics: `availableCrawler` (to receive messages from NAB-Crawler instances that are ready), `availableAnalyzer` (same purpose of `availableCrawler`, but reserved for ready NAB-Analyzer instances, and `projectFound` (to receive the name of the projects to analyze from NAB-Crawler instances). Then, NAB-Master simply waits for messages on these topics to arrive. Thanks to the loosely coupled and asynchronous architecture, NAB-Crawler and NAB-Analyzer instances can start at any moment or be spawned on demand.

Figure 2 depicts the interaction between NAB components for crawling and analysis (note that topic subscription is now shown in the figure). Whenever a NAB-Crawler instance starts, it subscribes to the `crawlerAssignment` topic to receive crawling tasks from NAB-Master. To announce that it is ready for crawling, the NAB-Crawler publishes a message to the `availableCrawler` topic together with its ID. NAB-Master stores the received ID in the `crawlers` list. Then, it verifies if there are available time intervals to crawl in the `timeIntervals` list, and, if so, publishes a message to the `crawlerAssignment` topic to start the crawling process, including the ID of the NAB-Crawler that should perform the crawling. Each NAB-Crawler instance receives the message, and verifies if its own ID corresponds to the ID included in the message. In this case, it processes the request and starts crawling.

The NAB-Crawler performs the query on GitHub's search API, and filters out the projects matching the analysis criteria (more details are given in Section 2.3). For each matching project (see the loop frame in Figure 2), a message is published to the `projectFound` topic such that NAB-Master can collect all projects amenable to analysis in the `projects` list. Once the crawling is terminated, a message is sent to the `availableCrawler` topic to signal the availability to crawl projects in a new time interval.

Following a procedure similar to the one employed for NAB-Crawlers, when a NAB-Analyzer instance starts, it subscribes to the `analyzerAssignment` topic and publishes a message to the `availableAnalyzer` topic specifying its ID. Such ID is stored in the `analyzers` list handled by NAB-Master. If there are entries in the `projects` list, NAB-Master publishes a message to the `analyzerAssignment` topic, specifying the ID of the NAB-Analyzer that should execute the DPA and the project to analyze. The corresponding NAB-Analyzer, upon verifying that its ID matches the one contained in the message, starts the DPA activity, by first cloning the project and then running the DPA tool though the selected plugin (more details on the DPA execution are given in Section 2.4). Upon completion, the NAB-Analyzer collects the generated results as well as statistics on the DPA execution (shown as `results` and `stats` in the figure, respectively), and stores them in the MongoDB database. Finally, the NAB-Analyzer discards the temporary data needed for running the

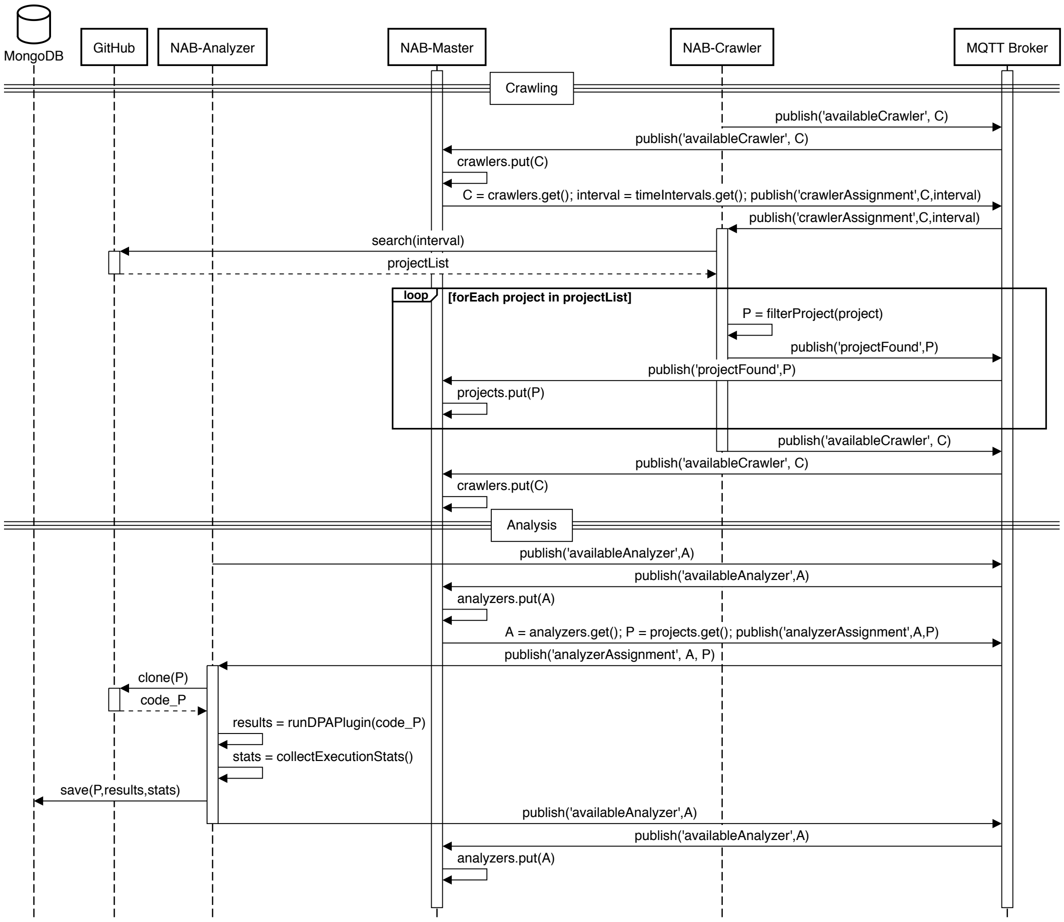

MongoDB
GitHub
NAB-Analyzer
NAB-Master
NAB-Crawler
MQTT Broker
Crawling
publish('availableCrawler', C)
publish('availableCrawler', C)
crawlers.put(C)
C = crawlers.get(); interval = timeIntervals.get(); publish('crawlerAssignment',C,interval)
publish('crawlerAssignment',C,interval)
search(interval)
projectList
loop
[forEach project in projectList]
P = filterProject(project)
publish('projectFound',P)
publish('projectFound',P)
projects.put(P)
publish('availableCrawler', C)
publish('availableCrawler', C)
crawlers.put(C)
Analysis
publish('availableAnalyzer',A)
publish('availableAnalyzer',A)
analyzers.put(A)
A = analyzers.get(); P = projects.get(); publish('analyzerAssignment',A,P)
publish('analyzerAssignment', A, P)
clone(P)
code_P
results = runDPAPlugin(code_P)
stats = collectExecutionStats()
save(P,results,stats)
publish('availableAnalyzer',A)
publish('availableAnalyzer',A)
analyzers.put(A)

Figure 2 Interactions between NAB's components during crawling and analysis (after component initialization).

DPA, initializes a fresh execution environment and announces that is ready for a new DPA activity by publishing a message to the `availableAnalyzer` topic. NAB-Master stops when no further time intervals to crawl and projects to analyze are in the corresponding lists.

2.3 Crawling

Each NAB-Crawler instance receives (from NAB-Master) a set of specifications describing the characteristics of projects to be crawled, including the time intervals to consider during crawling (which can express either the date of project creation or the last update), the programming language(s) or the build system(s) to match, and the maximum number of results per request. This information is used to query the GitHub's search API to collect metadata of the matching projects. Additionally, NAB-Crawler can filter the metadata resulting from the query according to various criteria, such as selecting only projects with a specific entry in the build file (useful for discarding projects that are unrelated to a given DPA), or those with a minimum number of forks, watchers, stars, or contributors. Such criteria are set by the user, and are sent to the NAB-Crawler instance by NAB-Master. Finally, the NAB-Crawler sends the matching projects' names to NAB-Master.

Since the GitHub API has a limit[3] of 1,000 results per search query (independently from the selected time intervals), NAB-Master automatically subdivides the user-defined search intervals such that each NAB-Crawler instance is assigned fewer projects to crawl than the aforementioned limit, thus eventually crawling all the projects in the specified time intervals.

2.4 Executing DPA

The analysis starts with cloning the source code of the project from GitHub. NAB provides support for different build systems, e.g., NPM for Node.js, SBT for Scala, or Maven (MVN) for Java and Scala. NAB-Analyzer spawns a process to run the automatic build system, which typically downloads dependencies and compiles the required code. Finally, the project's testing code is executed, applying a DPA through a plugin (see Section 2.5). Upon DPA completion, the NAB-Analyzer instance stores the analysis results and the execution statistics in the database. By default, the DPA is run on the most recent revision of the default branch (as set up by the manager of the project on GitHub).

Apart from the results of the DPA, NAB collects different statistics on the analysis execution, such as start and end timestamps for every performed activity, and the exit code of the spawned process (indicating success or failure). Such statistics are stored in the database along with the analysis results. NAB-Analyzer sets a timeout (called *analysis timeout*) for the spawned process running the DPA to prevent buggy, malicious, or non-terminating application code (or DPA code) from excessively consuming resources. DPAs exceeding the analysis timeout are forcibly terminated.

NAB also reports projects that fail to build. Build failures are usually caused by developers assuming the presence of pre-installed software or specific settings (e.g., variables or paths) in the environment, which instead are not present. For example, in a Node.js project, some module dependencies may be missing in its configuration. Similar issues can occur in Java projects, where some Maven-managed projects may fail to build due to missing libraries or tools (e.g., parsers or pre-processors).

Since NAB uses a two-level orchestration (i.e., an external orchestration handled by Docker Swarm and an internal one handled by NAB-Master), it can happen that NAB-Analyzer containers are restarted by Docker Swarm without NAB-Master being notified. To handle this case, NAB implements a fault-tolerance mechanism using a timer that is started for each scheduled DPA. If a result arrives before the analysis timeout, the timer is stopped; otherwise, the DPA is re-scheduled for execution for a configurable number of times. Users can configure how NAB should handle multiple results in the case of a restarted DPA (e.g., keep only the first result, keep all results, combine the set of results.)

2.5 NAB Plugins and DPA Tools

To run massive analyses, NAB provides a *plugin* mechanism for different analysis frameworks to run DPA tools.[4] To integrate a third-party DPA tool into NAB, the user needs to provide three shell scripts: (1) to set up the execution environment for the analysis framework, (2) to execute the DPA tool, and (3) to post-process the DPA results. NAB allows one to select a build system and a runtime, and takes care of executing the provided scripts and applying the DPA tools to the projects under analysis. Integrating existing DPA tools into NAB does not require much effort. Typically, creating a new plugin requires about 100 lines of code, divided into the three aforementioned shell scripts. The post-processing script is usually the longest, as it needs to format the DPA results in JSON for storing them in the NAB database.

[3] `https://developer.github.com/v3/search/`

[4] NAB plugins can also directly integrate DPA tools that do not depend on any analysis framework.

Table 1 NAB supported languages, build systems, analysis frameworks, DPA tools, and runtimes. The sign † indicates those used in the following case studies.

Language	Build System	Analysis Framework	DPA Tool	Runtime
JavaScript†	NPM†	NodeProf† [62]	Deep-Promise† JITProf† [17]	GraalVM† [68]
Java†	MVN†	DiSL† [41] AspectJ [34]	tgp† [54] JavaMOP [31]	HotSpot VM† [9] GraalVM
Scala†	SBT† MVN	DiSL†	tgp†	HotSpot VM† GraalVM

For the supported build systems, NAB captures the exact entry point where the runtime is invoked, and plugs in the corresponding DPA tool. For example, to run a DPA tool with a plugin for a weaver or instrumentation framework working at the Java bytecode level (e.g., AspectJ's load-time weaver [34] or the DiSL [41] dynamic instrumentation framework) the user can select the MVN build system and Oracle's HotSpot JVM [9] as runtime; NAB takes care of passing all the required parameters (e.g., instrumentation agents, libraries, or generated harness bytecode) to the JVM, to start the analysis. Existing DPA tools that can run in a containerized environment are suitable for NAB. On the contrary, DPA tools that require access to special OS- or hardware-layer features that are not supported in containerized environments (e.g., hardware performance counters or NUMA-specific CPU/memory policy control) are not candidates for a NAB plugin. Table 1 shows the currently supported programming languages, build systems, analysis frameworks, DPA tools, and runtimes.

Extending NAB to support other languages, build systems and testing frameworks is mostly a straightforward engineering effort. To extend NAB, one needs to modify the NAB-Analyzer component (generating new scripts wrapping the build system to set NAB-specific variables and to invoke the selected plugin) and its building configuration (updating the `Dockerfile`, to install the new build system and to deploy the runtime within the NAB-Analyzer image). All other core NAB components remain unchanged.

2.6 Implementation Technologies for Containerization

NAB-Master, NAB-Crawler, and NAB-Analyzer are implemented in Node.js and are deployed in containers in a Docker Swarm. NAB uses an overlay network inside the Swarm to optimize the communication between the NAB components and to leverage the internal DNS service (to reach NAB services by name, avoiding complex settings). NAB relies on Docker Compose [21], which simplifies the deployment of the whole analysis infrastructure on a cluster or in the Cloud. Since all NAB services are based on Docker container technology, NAB can be easily migrated to other Docker-friendly orchestration engines, such as Kubernetes [4] or Mesos [20], which provide similar functionalities as Docker Swarm. NAB has been tested on Azure Cloud [44] and on a shared cluster with hundreds of nodes.

3 Experimental Setup

Here, we present the experimental setup used for running NAB to analyze massive collections of open-source projects in our case studies. Even though crawling and analysis can be done at the same time in NAB, we separate both processes to first generate a list of projects that is then used for different DPAs.

Table 2 Number of selected GitHub projects. For each programming language (JavaScript/Node.js, Java, Scala), the first row lists the amount of all GitHub projects. Subsequent rows show the number of surviving projects after applying filters (i.e., the predicates in the leftmost column). The number of projects considered in our case studies is shown in the row "$\geq$ 2 Contrib.".

	2013	**2014**	**2015**	**2016**	**2017**	**Total**
All Node.js	226,879	380,607	620,211	1,055,799	1,708,261	3,991,757
NPM Build	226,858	332,399	363,181	364,871	363,898	1,651,207
Test Entry	22,956	65,018	100,581	134,917	158,547	482,019
$\geq$ 2 Contrib.	4,311	11,415	20,593	30,889	42,078	**109,286**
All Java	164,208	321,290	636,316	1,001,370	1,477,473	3,600,657
MVN Build	164,146	301,632	362,061	364,861	363,898	1,556,598
Test Entry	17,569	28,411	32,005	32,948	35,773	146,706
$\geq$ 2 Contrib.	2,748	4,361	5,583	6,112	7,114	**25,918**
All Scala	7,875	11,670	18,692	25,501	33,463	97,201
SBT Build	7,853	11,640	18,618	25,402	33,188	96,701
Test Entry	1,802	3,222	5,991	9,217	11,971	32,203
$\geq$ 2 Contrib.	222	404	706	1,021	1,723	**4,076**

3.1 NAB Configuration and Deployment

We run NAB in a shared cluster, composed of 176 nodes (each requiring reservation), where each node is equipped with a 16-core AMD Opteron CPU (2.6 GHz) and 128 GB RAM. We run 1,024 NAB-Analyzer instances using 64 nodes, i.e., 16 NAB-Analyzer instances per node. The nodes are connected to a 10 Gb/s internal network and to a 1 Gb/s external network.

We configure the NAB core services with V8 Node.js 10.9 deployed with Docker-CE 1.18. The NAB containers are built using a Linux Ubuntu 16.04 Docker image. We deploy Eclipse Mosquitto MQTT 1.4 broker and MongoDB 4.0.

For all the analyses performed in the three case studies, the NAB-Analyzer instances are configured with an analysis timeout of one hour.

3.2 Crawling and Project Selection

We crawl 5 years of Node.js,[5] Java, and Scala projects from GitHub (from 2013-01-01 to 2017-12-31). To minimize the number of (typically small) personal projects crawled, we only collect projects with a minimum of 2 contributors, as suggested by [33]. For each crawled project, we check whether an automatic build configuration file is present (i.e., NPM's `package.json` file for Node.js, SBT's `build.sbt` file for Scala, or MVN's `pom.xml` file for Java), such that the project can be automatically built. The analyses presented in this paper are conducted on the unit tests in the projects that can be run by automatic means (e.g., in case of NPM, by executing "`npm test`"). For this reason, we also check whether a test entry exists.

Table 2 shows the total number of crawled and selected projects per year (i.e., projects that existed and whose most recent commit made on any branch occurred that year). For

[5] Node.js projects are identified as JavaScript projects in GitHub, as JavaScript is used as programming language in the project description, but can be recognized thanks to the presence of Node.js-specific configuration files.

each programming language considered, the table reports the number of projects that can be built, that contain a test entry, and those with at least 2 contributors (such filters are applied in cascade). Only projects passing all filters (i.e., 109,286 Node.js projects, 25,918 Java projects, and 4,076 Scala projects, as reported in the last row) are considered in the following case studies.

4 Case Study I: Analyzing the Use of Promises in Node.js

In our first study, we present a novel dynamic analysis (called Deep-Promise) to collect insights on how developers use JavaScript's Promise API, allowing one to understand its actual adoption to handle asynchronous executions. The Promise API is a key novel language feature introduced in the ECMA specification to enable better handling of asynchronous interactions in JavaScript applications. Promises greatly simplify the way asynchronous code can be expressed, by introducing the notion of *promise chain*, i.e., a sequence of asynchronous events with logical dependencies.

The goal of this study is to provide a better understanding of the popularity and usage trend of the API in real-world projects and in the modules they depend on, which can be useful for the Node.js community. To the best of our knowledge, this is the first large-scale study on the use of promises in Node.js projects and the NPM modules they depend on, enabled by NAB.

4.1 Monitoring Promises in JavaScript

The introduction of promises since ECMAScript 6 [29] greatly simplifies the development of asynchronous applications in JavaScript. With promises, asynchronous executions can be implemented elegantly, avoiding the so-called "callback hell" problem, by enabling chaining of (asynchronous) functions [1]. The value that resolves or rejects a promise can be used as the input of the reacting promise(s). The latter promise(s) depend on the previous one, forming a promise chain.

In [40], the authors introduce the notion of *promise graph*, a formal graph-based model for understanding and debugging code developed using the Promise API. A promise graph is composed of several promise chains. Each promise chain is an acyclic data structure showing the dependencies among promises and the values that resolve or reject each promise. The *size* of a promise chain, i.e., the number of promises inside the chain, gives us insights about the use of promise constructs. As promise chains of size one have no subsequent reactions to be executed asynchronously, such chains are not used to handle asynchronous executions. Hence, we denote as *trivial* a promise chain of size one, as opposed to a *non-trivial* promise chain, which size is greater than one.

Building a DPA to accurately capture all promise chains requires an instrumentation framework capable of intercepting every use of the Promise API. Before explaining Deep-Promise, we first introduce the underlying instrumentation framework NodeProf, integrated in NAB through a dedicated analysis plugin.

4.1.1 NodeProf Framework

NodeProf [62] is an open-source[6] dynamic instrumentation framework for Node.js based on GraalVM [68]. NodeProf relies on the dynamic instrumentation of the Abstract Syntax Tree (AST) interpreter of the GraalVM JavaScript engine. In contrast to other dynamic analysis

[6] https://github.com/Haiyang-Sun/nodeprof.js

frameworks for Node.js (e.g., Jalangi [57]), NodeProf is compatible with ECMAScript 8, and supports JavaScript constructs such as promises and `async/await`, which are not supported in other frameworks. An additional advantage of NodeProf is that it instruments only loaded code, while other frameworks typically rely on source-code instrumentation, which usually needs to instrument all source code files in advance before execution, even if they are not used. NodeProf's approach can significantly reduce the time needed for the instrumentation, as the NPM modules a project depends on can contain thousands of source code files (while only a small portion of them may be used by the project).

4.1.2 Deep-Promise DPA

Deep-Promise is a DPA implemented on top of NodeProf that constructs the full promise graph of any Node.js application at runtime. Deep-Promise tracks the creation of all promises and the dependent relations between them. There can be three different dependencies between two promises: (1) *fork*, where a promise has one or more reactions registered, e.g., via `Promise.then` or `Promise.catch`; (2) *join*, where multiple promises join into one promise via `Promise.all` or `Promise.race`; (3) *delegate*, where a promise is used as a value to resolve or reject another promise. Additionally, Deep-Promise also tracks usages of `async/await`, which deal with implicit promises.

Deep-Promise directly instruments the built-in promise implementation, capturing the creation of every promise and the dependency between promises. Upon program termination, the promise graph is dumped and stored in the database by the NAB-Analyzer instance executing the analysis. The resulting promise chains are later analyzed with an offline tool to compute statistics.

4.2 Executing Deep-Promise with NAB

For every project to analyze, we run the automatic build system through NPM, by executing "`npm install`", which downloads, compiles, and installs dependencies, including third-party testing frameworks such as *mocha*, *unitest*, or *grunt*. If the installation succeeds, we run the default test program in the project by executing "`npm test`" with Deep-Promise enabled.

From the 109,286 executed Node.js projects (see Table 2 on page 10), NAB reports 23,297 successfully analyzed projects with Deep-Promise. Unsuccessful projects include those with broken tests (e.g., wrong test settings or assuming non-standard pre-installed software), projects with failing tests, and those exceeding the 1-hour analysis timeout of our cluster-based experimental setup.

The total execution time reported by NAB is 7.1 hours. Running the analysis sequentially in a single NAB-Analyzer would take up to about 2.7 months.[7]

4.3 Promise API Adoption for Asynchronous Executions

First, we measure how widely promises are used in project tests by identifying the usage of the Promise API in either application code (i.e., code exercised by tests) or the dependent NPM modules. From the 23,297 successfully analyzed projects, we find that 5,971 projects (i.e., 25.6%) make use of the Promises API. In our analysis, we differentiate trivial promise chains (that are not used to handle asynchronous executions) from non-trivial ones.

[7] This estimation is calculated as the sum of the execution times reported for the analysis of each project by a NAB-Analyzer instance.

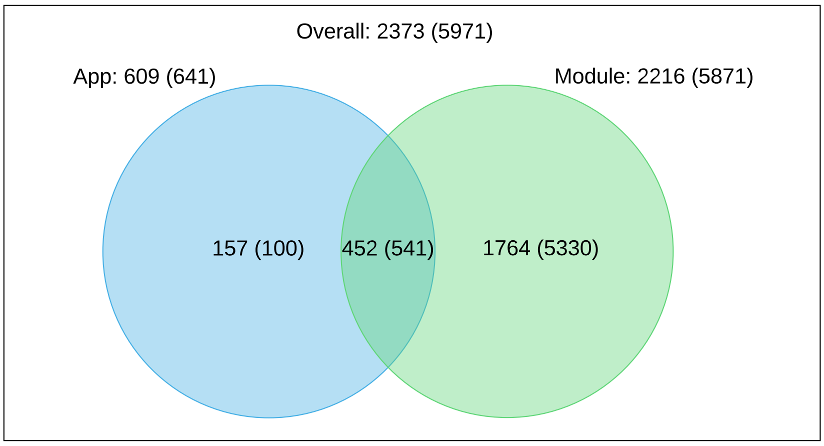


Figure 3 Distribution of projects that use promises only in application code (blue circle), only in modules (green circle), or in both (intersection). The left-side values indicate the number of projects excluding trivial promises, while the right-side values in parentheses include trivial promises.

Figure 3 shows the overall distribution of projects that use promises in the application code (blue circle), in the dependent NPM modules (green circle), or in both (their intersection). The right-side values in parentheses show the total number of projects making use of promises (before excluding trivial promise chains), while the left-side values indicate the number of projects after the exclusion. Overall, 5,971 projects use promises, while only 2,373 projects use non-trivial promise chains. The left-side numbers (after excluding trivial promises) are smaller than the right-side numbers in parentheses, except for the application-only part, because some projects use promises in both application code and in modules, but only trivial promises in modules. After excluding trivial promises, such projects are found using promises only in the application code.

From the figure, we can observe that out of 23,297 projects, only 157 (0.6%) of them use promises only in application code, while 2,216 (9.5%) use promises also indirectly, i.e., in the modules they depend on. Our findings suggest that many projects do not directly depend on promises, but rather rely on the promise support introduced by other modules that they depend on.

Our analysis also reveals that from the 5,971 projects that use the Promise API, a total of 440 different dependent NPM modules use promises, out of which 41 modules create only trivial ones. Figure 4 shows the most frequently used NPM modules that make use of promises on the x-axis (used by at least 100 projects), and the number of projects using such NPM modules on the y-axis. Each bar represents the number of projects and is further divided into two parts according to the maximum size of the promise chain (i.e., showing trivial and non-trivial promise chain usages). Modules *lodash* and *prettier* contain only trivial promise chains, as they use promises only for version detection.[8] The other modules use promises for different purposes. For example, *jest* and *mocha* are test harnesses widely used by NPM modules, which use promises to run tests, while *pify* is a library used to “promisify” a callback by returning a promise-wrapped version of it.

[8] Such modules execute `Promise.resolve` and check whether the returned object is a promise object or not, to detect whether the current JavaScript version is ECMAScript 6 or higher. Such a promise usage is unrelated to asynchronous executions.

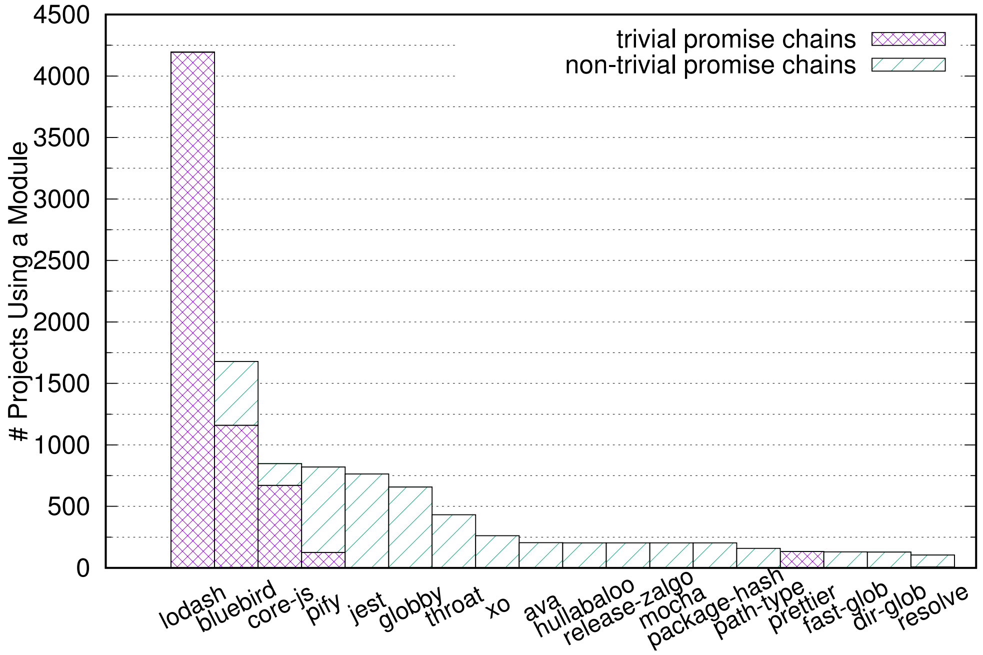


Figure 4 Most frequently used NPM modules that make use of promises, and the number of projects using such modules.

Our results suggest that such modules (i.e., with non-trivial promises) should be preferred over the others (i.e., without promises or with only trivial ones) by researchers as well as language implementers who might want to evaluate and optimize the usage of promises in real-world Node.js applications. Specifically, identifying and optimizing a popular promise usage pattern in one of such modules might have a positive performance impact on several existing applications.

4.4 Frequency of Promise API usage

We further study the frequency of Promise API usages among the 2,373 projects with non-trivial promise chains. The `Promise` constructor and the `Promise.then` method are the most used ones (in 2,287 and 2,363 projects, respectively), as they are the most common way to create and use promises. `Promise.catch` is used less frequently (in 1,732 projects), which reveals that not all programmers add a `catch` statement when programming with promises (which is considered a best practice when using promises in JavaScript [55]). Other APIs, such as `Promise.all` and `Promise.race`, are used even less frequently (in 1,488 and 233 projects, respectively). Finally, only 47 projects use `await` for asynchronous functions, as the `async/await` feature was introduced in ECMAScript 8 [30] (mid 2017).

The size of a promise chain is an important characteristic for understanding how applications use promises. Figure 5 shows the distribution of different maximum promise-chain sizes among the 2,373 projects that use non-trivial promise chains. 50.5% of the projects create only promise chains sized within 10; in 38.3% of the projects the longest promise-chain has a size between 11 and 100; and 11.2% of the projects have at least one promise chain with size greater than 100. We observe the longest promise chain (5,002 promises) in the project *lahmatiy/postcss-csso*.

Our findings suggest that the number of projects with long promise chains is relatively high. As the length of the promise chain is a potential indicator of a long-living application, such projects could be considered as potentially interesting for the development of microbenchmarks stressing the Promise API.

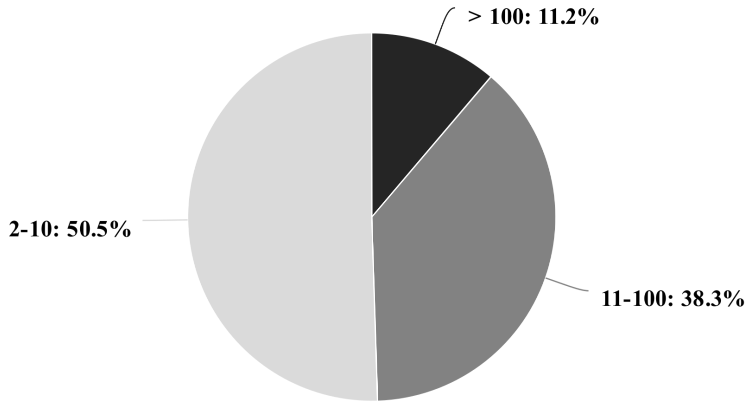


Figure 5 Statistics for the maximum promise-chain size observed in the 2,373 projects that make use of non-trivial promise chains. Percentages indicate projects with maximum chain size between 2 and 10, between 11 and 100, and more than 100.

In conclusion, this study demonstrates how NAB can be used for analyzing the usage of a particular programming-language construct (here, asynchronous executions via promises in Node.js projects) in the wild. The results of our analysis could be useful for Node.js developers to find projects and popular modules that use promises for asynchronous executions. In particular, evaluating and optimizing such modules could be beneficial to several existing applications. Moreover, we found many projects with long promise chains; such projects might be considered as potentially interesting for benchmarking promises on Node.js.

In future work, we plan to re-execute our analysis on new versions of the considered modules, and to track the adoption of the Promise API over time. In this way, the analysis could also indicate which specific parts of the API are gaining more adoption (if any).

5 Case Study II: Finding JIT-unfriendly Code Patterns in Node.js

Our second massive study of public repositories using NAB deals with the quality of the JavaScript code available on the NPM package repository. Specifically, our goal is to execute a comprehensive DPA to identify bad coding practices that are known to affect the performance of Node.js applications. Such coding patterns – also called *JIT-unfriendly* code patterns – may prevent typical JIT compiler optimizations, such as function inlining, on-stack replacement, and polymorphic inline caching.

To this end, we resort to an existing DPA for JavaScript, called JITProf [17], which can identify and collect a variety of JIT-unfriendly code patterns otherwise impossible to identify using static analysis. JITProf is open-source,[9] and relies on the Jalangi [57] instrumentation framework. Since Jalangi does not support the latest ECMAScript standard, we adapt the analysis to run on the NodeProf framework (see Section 4.1.1) for the GraalVM.

This yields the benefits of supporting up-to-date language features (as NodeProf is compatible with ECMAScript 8), and also reduces the time needed for the instrumentation.

JITProf can identify seven different categories of JIT-unfriendly code patterns, namely: `AccessUndefArrayElem`, tracking accesses to undefined array elements; `BinaryOpOnUndef`, to track when `undefined` is used in binary operations; `InconsistentObjectLayout`, to

[9] `https://github.com/Berkeley-Correctness-Group/JITProf`

Table 3 Amount of projects where at least a JIT-unfriendly code pattern in found (out of 26,938 analyzed Node.js projects). Dependent NPM modules are excluded from the analysis.

JIT-unfriendly Pattern	**# Projects**	**%**
`AccessUndefArrayElem`	1,253	4.7%
`BinaryOpOnUndef`	757	2.8%
`InconsistentObjectLayout`	9,509	35.3%
`NonContiguousArray`	194	0.7%
`PolymorphicOperation`	3,073	11.4%
`SwitchArrayType`	81	0.3%
`TypedArray`	546	2.0%
At least one	**9,969**	**37.0%**

find object access patterns that lead to inline-cache misses; `NonContiguousArray`, to locate non-contiguous array accesses; `PolymorphicOperation`, to identify polymorphic (including megamorphic) binary and unary operations; `SwitchArrayType`, to find unexpected transitions in arrays internal backing storage (i.e., array strategies [6]); `TypedArray`, to account for unnecessary usages of generic arrays where typed arrays could be used (e.g., to replace contiguous numeric arrays). A detailed description of the analyses required to identify such patterns can be found in [17].

5.1 Executing JITProf with NAB

We run JITProf with NodeProf in two settings, i.e., (1) profiling only application code, and (2) profiling also dependent NPM modules. Out of the 109,286 Node.js projects executed (see Table 2 on page 10), NAB reports 26,938 successfully analyzed projects with application-only profiling, and 3,940 projects when profiling also dependent NPM modules. The analyses in JITProf that identify JIT-unfriendly patterns require heavy instrumentation (as they track object creation, accesses and operations), which may significantly slow down application execution (leading to failures due to the presence of timeouts in the code, or increasing the execution time past the 1-hour analysis timeout) or cause out-of-memory errors, particularly when profiling NPM modules, due to the large amount of code that is instrumented and analyzed in all dependent NPM modules in each project. For this reason, JITProf does not complete on several projects, which are ignored in our analysis. Similarly to the previous use case, we exclude projects with broken or failing tests.

The total execution time reported by NAB is 25.3 hours, whereas a sequential execution in a single NAB-Analyzer would take up to about 8.4 months.

5.2 JIT-unfriendly Patterns in Application Code

We first focus on application code disregarding the dependent NPM modules. The results of the analyses are shown in Table 3. As shown in the second column ("# Projects"), a total of 9,969 projects result in *at least one* JITProf warning, i.e., 37.0% of the successfully analyzed projects suffer from at least one JIT-unfriendly code pattern. The most common pattern found is `InconsistentObjectLayout`, occurring in 9,509 projects. This result implies that these projects perform read or write accesses to objects in a sub-optimal way that may prevent compiler optimizations, and may therefore pay a performance penalty when performing such accesses in frequently executed code.

Table 4 Top 3 dependent NPM modules suffering from JIT-unfriendly code. Column "# Modules" indicates the number of unique modules where a given pattern occurs. For each module, the values in parentheses indicate the number of projects using that module.

JIT-unfriendly Pattern	# Modules	Top 3 NPM Modules		
`AccessUndefArrayElem`	252	*commander* (637)	*glob* (569)	*abbrev* (178)
`BinaryOpOnUndef`	83	*strip-json-comments* (110)	*jsbn* (110)	*sinon* (83)
`InconsistentObjectLayout`	523	*commander* (687)	*chai* (369)	*tape* (337)
`NonContiguousArray`	49	*semver* (167)	*jsbn* (110)	*eslint* (51)
`PolymorphicOperation`	453	*lodash* (311)	*glob* (178)	*mime-types* (174)
`SwitchArrayType`	16	*babylon* (4)	*lodash* (3)	*eslint* (3)
`TypedArray`	144	*lodash* (51)	*jshint* (48)	*regenerate* (38)
At least one	**900**	***commander*** (**963**)	***glob*** (**569**)	***lodash*** (**432**)

5.3 JIT-unfriendly Patterns in NPM Modules (top 3)

We also collect statistics on JIT-unfriendly code in the NPM modules used by the exercised tests, which is reported in Table 4. From the set of 3,940 Node.js projects successfully analyzed, 900 dependent NPM modules execute at least one JIT-unfriendly code pattern, as shown in the second column ("# Modules"). The most common JIT-unfriendly code pattern is `InconsistentObjectLayout` (in 523 modules), while only 16 modules show occurrences of `SwitchArrayType`. For each code pattern, we also identify the top 3 modules (i.e., the 3 NPM modules used most frequently among projects where we found the pattern). The values in parentheses below a module name in Table 4 show the number of projects using that module. For example, as shown in the first row of the table, there are 637 projects using the NPM module *commander*, 569 projects using *glob*, and 178 projects using *abbrev*.

Several modules suffer from more than one JIT-unfriendly code pattern. For example, the popular NPM module *lodash* frequently executes 3 kinds of JIT-unfriendly code patterns (i.e., `PolymorphicOperation`, `SwitchArrayType` and `TypedArray`). The 3 top modules with at least one JIT-unfriendly code pattern are *commander*, *glob* and *lodash* (as reported in the last row of the table). These modules are very popular, being imported by almost 130K other distinct NPM modules overall.[10]

In summary, our study reveals that Node.js developers frequently use code patterns that could prevent or jeopardize dynamic optimizations and have a potential negative impact on applications performance. Such patterns occur both in application code and in dependent NPM modules used by a project.

[10] The estimation of the number of dependent modules is taken from the global NPM registry (`https://www.npmjs.com/`) and dated November 2018.

6 Case Study III: Discovering Task-parallel Workloads

For many specific evaluation needs, there is a lack of suitable domain-specific benchmarks, forcing researchers to resort to less appropriate general-purpose benchmarks. In our third case study, we focus on discovering workloads that can fulfill the needs of a domain-specific evaluation. We consider a researcher requiring task-parallel applications on the JVM exhibiting diverse task granularities, to analyze concurrency-related aspects. Currently, only few task-parallel workloads can be found in well-known benchmark suites [5, 58, 61, 60] targeting the JVM. Moreover, except for DaCapo,[11] such suites were last updated several years ago, thus they may be not representative of state-of-the-art applications using task parallelism.

6.1 Executing tgp with NAB

`tgp` [54] is an open-source[12] DPA for detecting the granularity of tasks spawned by multithreaded, task-parallel applications running on the JVM. `tgp` profiles all tasks spawned by an application (defined as subtypes of the Java interfaces/classes `java.lang.Runnable`, `java.util.concurrent.Callable`, and `java.util.concurrent.ForkJoinTask`), collecting their *granularity*, i.e., the amount of work carried out by each parallel task, in terms of the number of executed bytecode instructions. The DPA runs on top of DiSL [41], a dynamic analysis framework for the JVM that ensures complete instrumentation coverage (i.e., it can instrument every method with a bytecode representation), thus enabling the detection of tasks used inside the Java class library.

For this case study, we run `tgp` on top of DiSL (which can be attached to NAB via a dedicated plugin) to measure the granularity of all tasks spawned during the execution of testing code in Java and Scala projects from GitHub. Our goal is to discover projects with a high diversity of task granularities, which could be good workload candidates for benchmarking task parallelism on the JVM.

The total execution times reported by NAB for this DPA are 9.5 hours (Java projects) and 1.8 hours (Scala projects), whereas a sequential execution in a single NAB-Analyzer instance would take up to about 1.9 months (Java) and 12.5 days (Scala).

6.2 Results for Java Projects

Out of the 25,918 Java projects analyzed (see Table 2 on page 10), 1,769 projects make use tasks and successfully complete all tests within the 1-hour analysis timeout, thus they are considered for the following analysis.

The total number of tasks spawned by the analyzed projects is 1,406,802. The minimum granularity found is 1 and the maximum granularity is 187,673,879,636. Table 5 (Java section) shows the distribution of tasks wrt. their granularities. The first column shows all ranges of task granularities found. The second column reports the number of tasks with granularity in the corresponding range. The third column indicates the number of projects having at least one task with a granularity in the considered range.

The analysis results obtained with NAB reveal that test methods in project `https://github.com/rolfl/MicroBench` (a Java harness for building and running microbenchmarks written in Java 8) spawn a total of 55 tasks with granularities spanning all ranges

[11] Dacapo 9.12-MR1-bach was released on January 2018. However, the workloads were not significantly modified since the previous release (dated 2009) and no new workload has been added.

[12] `https://github.com/Fithos/tgp`

Table 5 Distribution of all tasks spawned in the considered Java and Scala projects wrt. their granularities.

Granularity	Java		Scala	
Range	Tasks	Projects	Tasks	Projects
$[10^0$ - $10^1)$	137,468	686	301,066	771
$[10^1$ - $10^2)$	278,765	466	280,244	710
$[10^2$ - $10^3)$	215,211	673	2,795,702	860
$[10^3$ - $10^4)$	285,196	1,092	1,278,974	769
$[10^4$ - $10^5)$	247,284	1,367	124,473	771
$[10^5$ - $10^6)$	128,992	1,492	74,989	769
$[10^6$ - $10^7)$	89,710	1,327	13,002	806
$[10^7$ - $10^8)$	17,178	1,046	4,555	677
$[10^8$ - $10^9)$	5,696	581	1,789	619
$[10^9$ - $10^{10})$	1,164	177	430	276
$[10^{10}$ - $10^{11})$	120	53	22	20
$[10^{11}$ - $10^{12})$	18	8	1	1

except $[10^4$ - $10^5)$. In addition, the project `https://github.com/47Billion/netty-http` (a library to develop HTTP services with Netty [65]) makes use of a total of 123 tasks in its tests, with granularities in all ranges except $[10^{11}$ - $10^{12})$. Both Java projects can be good candidate workloads for task benchmarking, as they exhibit a high diversity of task granularities.

6.3 Results for Scala Projects

Out of the 4,076 Scala projects analyzed (see Table 2 on page 10), 860 projects contain task-parallel workloads and successfully complete all tests within the analysis timeout. Such projects spawn a total of 4,875,247 tasks. The minimum granularity found is 2 while the maximum is 204,418,653,894. Table 5 (Scala section) shows the distribution of tasks wrt. their granularities.

The analysis results pinpoint three candidate workloads spawning tasks with granularities spanning all ranges except $[10^{11}$ - $10^{12})$. First, the project `https://github.com/iheartradio/asobu` (a library for building distributed REST APIs for microservices based on Akka cluster [36]) spawns a total of 19,880 tasks in its tests. Second, project `https://github.com/TiarkRompf/virtualization-lms-core` (a library for building high performance code generators and embedded compilers in Scala) executes a total of 5,759 tasks. Finally, tests inside project `https://github.com/ryanlsg/gbf-raidfinder` (a library for tracking gaming-related tweets from Twitter) run a total of 20,934 tasks. This set of projects provides candidate Scala workloads for benchmarking task execution on the JVM, due to their high diversity of task granularities.

Overall, our analysis results show that NAB can help discover good candidate workloads satisfying domain-specific benchmarking needs. Moreover, this study demonstrates NAB's support for multiple programming languages.

7 Discussion

In this section, we discuss the strengths and limitations of our approach, focusing on different aspects of massive DPA, including safety, extensibility, scalability, and code evolution.

7.1 Safety

As NAB is executing unknown projects, sandboxing is crucial to protect the execution platform from malicious or erroneous code. NAB relies on Docker containers to isolate the execution of DPAs from the host environment. Thus, a project may only crash a NAB-Analyzer instance running in a Docker container, without harming the underlying platform. Such crashes are handled by NAB's fault-tolerance mechanism. A project that repeatedly crashes or takes excessive time to execute will be excluded from any further DPA. The fault-tolerance mechanism also mitigates problems caused by unstable third-party DPA tools or by experimental runtimes.

7.2 Extensibility

NAB has been designed for extensibility, as code repositories have different search APIs, and projects may use different version-control systems, programming languages, build systems, and testing frameworks. Moreover, third-party DPA tools may require specific execution environments, such as a modified JVM. NAB uses a plugin mechanism to handle the large variety of systems it needs to interact with. The currently supported analysis settings are listed in Table 1 on page 9. It is straightforward to implement additional plugins; each existing NAB plugin has only about 100 lines of code.

We plan to add support for other version-control systems such as Mercurial [43], and for other programming languages and their ecosystems. Moreover, supporting the Java Microbenchmark Harness (JMH) [48] is straightforward, allowing leveraging existing benchmarks in open-source projects that use MVN. Furthermore, we are extending NodeProf to support additional dynamic languages offered by GraalVM, such as Python, Ruby, and R. These extensions will enable studies on an even larger code base, targeting many popular programming languages.

Finally, in addition to the direct interaction with the standard GitHub search API, NAB can be extended to interact with offline mirrors and metadata archive dataset of GitHub, such as GHTorrent [18] and GHArchive [15], which may improve the crawling time. However, the suitability of using such offline services strongly depends on the type of DPA and required metadata. For example, for the case studies presented in this paper, we found that GHTorrent and GHArchve lack part of the required metadata (e.g., information about build system and the number of contributors).

7.3 Scalability

NAB's analysis infrastructure has been designed with scalability as a driving principle, leveraging a container-based microservice architecture. We executed NAB in clusters varying the number of nodes (16, 32, 48, and 64) with a constant number of containers per node (16), observing close to linear scalability when testing the core analysis infrastructure, i.e., running a test project with fixed execution time and without any analysis plugin, thus confirming the low overhead of NAB's distributed infrastructure.

Although NAB features load-balancing mechanisms for its publish-subscribe communication infrastructure and for database accesses, during massive DPA (as in our case studies) we started observing some performance degradation with more than 1K NAB-Analyzer containers, when the number of messages exchanged for analysis coordination, result notification and result storing increases significantly. This is due to the limits reached by Docker Swarm's internal overlay network, which will be improved as Docker evolves. This issue can be mitigated by running several NAB deployments (i.e., running in separate Docker Swarms) that coordinate themselves using external MQTT brokers.

The optimal number of NAB-Analyzer instances to run on each host depends on the resource demands of the DPA tool to be executed. Even though Docker Swarm's scheduler tries to distribute the load fairly, it cannot distinguish the containers' roles and may stop and restart them without making any difference. While the restart of NAB services is handled by the fault-tolerance mechanism, it is important to avoid deploying critical NAB services (notably NAB-Master and the MQTT Brokers) on the same host where high-demanding NAB-Analyzer instances are deployed, so as to avoid performance penalties in running the analysis. NAB provides configurable deployment settings to properly place the core services.

7.4 DPA Reproducibility and Code Evolution

NAB can differentiate between different versions of a DPA (e.g., when an existing DPA is updated, it results in a new version). Moreover, NAB can apply an arbitrary DPA version to an arbitrary project revision. When a DPA completes, NAB stores provenance metadata, which identifies the DPA version and project revision. This metadata makes DPAs reproducible, as NAB allows one to re-execute a specific DPA version on a specific project revision even if they have been updated.

For version dependency management supporting wildcards, as in the case of Node.js, since many versions may match a given wildcard for a single dependency, it would be extremely costly (and thus impractical) to test each valid version for every dependency. NAB follows the default behavior of "`npm install`", which installs only the latest version of a dependent module matching the wildcard. Thus, provenance metadata includes the exact dependency version installed.

NAB enables the analysis of multiple revisions of the same project, which helps gain insight into the changing runtime behavior during project evolution. NAB supports this through incremental analysis: the user may request the analysis of only those project revisions that have not been analyzed yet, avoiding to re-analyze projects that have not changed since the last run of the same DPA. While such studies would significantly increase the number of analyses to run, NAB enables such computationally expensive analyses thanks to its distributed and scalable architecture, allowing the deployment of an analysis on a large cluster or in the Cloud.

Preserving provenance metadata is also useful to understand the evolution of a DPA. The user may apply the new version of a DPA to exactly the same project revisions analyzed previously (with an older version of the DPA). Exploring the differences between the analysis results can help identify bugs in the DPA implementation.

7.5 Limitations

In the following text, we outline the main limitations of our approach.

7.5.1 Low-level Metrics

One inherent limitation when running in a virtualized environment (which container technology is based on) is that low-level performance counters such as hardware performance counters [28] are restricted or not accessible. Thus, low-level dynamic metrics related to the CPU or memory subsystem cannot be collected when running the NAB components in Docker containers. Moreover, if multiple NAB components are deployed on the same machine, performance interference will prevent the collection of accurate time-based metrics.

This limitation is due to the trade-off between the safety offered by Docker containers and the flexibility of collecting arbitrary low-level metrics. While NAB also supports a deployment setting without containerization, such a setting would sacrifice the safety properties needed when automatically executing the code of unknown (and hence untrusted) projects. Thus, for the measurements presented in this paper, we always use NAB with containerization.

7.5.2 Security Vulnerabilities and OS/Kernel Dependencies

NAB relies on Docker's default sandboxing (i.e., it does not use any privileged setting). However, DPA tools that require host's kernel system calls (e.g., the linux "`perf`" tool) will need to run NAB with privileged access (thus, potentially insecure). Furthermore, since the real host kernel is used by such tools, this setting will not work in virtualized environments. This is a well-known limitation of hardware-specific and OS-dependent profilers running on Docker.[13]

7.5.3 Representativeness of Testing Code

A general limitation of the presented use cases is that DPAs are applied only to the existing testing code of open-source projects. Such workloads may not be representative for a real usage scenario of an application in production. However, since testing code in projects also exercises library code extensively (as is the case of Node.js projects executing dependent NPM modules), significant information on the dynamic behavior of library code can be collected and analyzed. Thus, our approach ensures that the analyzed code stems from real applications and libraries, and the analyses presented in this paper yield relevant results. Moreover, in [69] the authors point out that many projects have relatively long-running testing code, different from simple and short unit tests.

As mentioned before, we will provide a plugin for JMH to analyze existing benchmarks in open-source projects, in addition to testing code. Furthermore, we plan to apply techniques for automated test-case generation [50, 59] to yield executable (and hence dynamically analyzable) code that maximizes various coverage metrics [42].

7.5.4 Analyzed Codebase and Analysis Timeout

The cluster used for obtaining our evaluation results requires a reservation and is heavily booked. For this reason, the projects considered in our use cases cover only the period 2013–2017. An extension of the use cases to cover also year 2018 is already scheduled. The choice of 1 hour as analysis timeout for DPAs also stems from the need of limiting the computational effort of the DPAs, to complete them within the limited timeframe of the cluster reservation.

8 Related Work

In this section, we provide an overview of the most significant related work. First, we review massive analysis of code repositories. Next, we discuss DPAs targeting JavaScript. Finally, we focus on previous approaches for generating benchmarks.

[13] `https://docs.docker.com/engine/security/seccomp/`

8.1 Massive Analysis of Code Repositories

Analyzing publicly available code repositories has become an important research area for understanding and improving different characteristics of software. Most related work relies on static analysis, notably for evaluating code quality [39, 52], predicting program properties [53], detecting code duplication [37], checking contracts in Java [11], identifying effects of software scale [38], automatically documenting code modifications [7], and summarizing bug reports [51]. Although static analyses can shed light on several aspects of software, there is a large body of properties that can be observed only when applications are executed [8], as DPA exposes the system's actual behavior. Our work facilitates the application of third-party DPAs to the projects in large code repositories.

Studies massively applying DPA are scarce. Legunsen et al. [35] use the JavaMOP [31] runtime-verification tool to check the correct usage of Java APIs. Even though the authors target 200 projects, their evaluation does not rely on any automated system enabling massive DPA. Marinescu et al. [42] present a framework to analyze how open-source projects evolve in terms of code, tests, and coverage by collecting both static and dynamic metrics. Although the authors apply DPA using containers to easily deploy several versions of the studied applications, their evaluation is limited to only 6 open-source projects. Overall, the aforementioned work dynamically analyzes a small and fixed set of projects, lacking an automatic and scalable system supporting massive custom DPA in the wild, as offered by NAB.

8.2 DPA for JavaScript

Dynamic analysis of JavaScript and Node.js applications is an active area of research. In [40] the authors introduce the notion of *promise graph*, a graph-based model to reason about the usage of JavaScript Promise objects through graphical visualization. Promise graphs are built using DPA, and can be used to identify bugs and API misuses in Node.js. A follow-up paper [1] from the same authors expands the work on promise graphs to perform automatic bug detection on real-world Node.js applications. The main focus of promise graphs is bug detection, while our DPA Deep-Promise focuses on the characterization of the Promise API usage and on asynchronous application behaviors (non-trivial promise chains) in the wild. To the best of our knowledge, no other large-scale study on the usage of the Promise API on Node.js projects and the NPM modules they depend on has been conducted.

Beyond JavaScript promises, DPA has been applied to JavaScript and Node.js in a variety of forms. As an example, JITProf [17] is a DPA tool that can identify JIT-unfriendly code patterns in JavaScript programs. JITProf lacks the ability to perform analyses on large code bases, and the JITProf paper evaluates only 50 client-side JavaScript applications. With NAB, we are able to scale analyses similar to those of JITProf up to a significantly higher number of JavaScript applications, enabling more representative results.

8.3 Benchmark Generation

Several studies focus on the creation of hand-coded synthetic benchmarks [10, 67], synthetic workload traces [49, 47, 13], and automatically synthesized benchmarks [3, 66, 32]. Overall, these techniques *generate* short workloads exhibiting a set of desired behaviors (e.g., intensive use of CPU, memory, I/O) to enable estimating and comparing the performance of hardware and applications. In contrast, our approach massively applies DPAs to *existing* testing code at the scale of public code repositories, as a technique for automatically discovering potential workload candidates satisfying domain-specific benchmarking needs.

To the best of our knowledge, NAB is the first system that can automatically run third-party DPAs in the wild. Similar in spirit, the AutoBench [69] toolchain can be used to look for potential benchmarks in Java workloads. In comparison to NAB, AutoBench lacks scalability, multi-language support, failure handling, sandboxing, and parallel code-repository crawling and analysis on clusters or in the Cloud. Moreover, AutoBench only supports MVN projects, relies exclusively on JUnit, and lacks any plugin mechanism for integrating third-party DPA tools that are fundamental for conducting analyses in the wild.

9 Conclusions

Motivated by the vast amount of today's public open-source code and available ready-to-use software components, this paper tackles two important research questions: whether it would be possible to develop a tool to automate large-scale DPA on public open-source software at a large scale, and whether such a tool would be useful for the community.

To positively answer the first question, we develop NAB, a novel, distributed infrastructure for executing massive custom DPA on open-source code repositories. NAB resorts to containerization for efficient DPA parallelization (fundamental to obtain analysis results in reasonable timeframes), sandboxing (to isolate buggy or malicious code) and for simplifying the deployment on clusters or in the Cloud. NAB features both crawler and analyzer components, which are deployed in lightweight containers that can be efficiently replicated. Moreover, NAB supports different build systems, testing frameworks, runtimes for multi-language support, and can easily integrate existing DPA tools. To the best of our knowledge, NAB is the first scalable, container-based infrastructure for automated, massive DPA on open-source projects, supporting multiple programming languages.

To positively answer the second question, we present three case studies where NAB enables massive DPA on more than 56K open-source projects hosted on GitHub, leveraging unit tests that can be automatically executed and analyzed. We present a novel analysis that sheds light on the usage of the Promise API in open-source Node.js projects. We find many projects with long promise chains, which can potentially be considered for benchmarking promises on Node.js. Moreover, the results of our analysis could be useful for Node.js developers to find projects and popular modules that use promises for asynchronous executions, which optimization could be beneficial to several existing applications. We conduct a large-scale study on the presence of JIT-unfriendly code on Node.js projects. Our study reveals that Node.js developers frequently use code patterns that could prevent or jeopardize dynamic optimizations and have a potential negative impact on applications performance. Finally, we perform a large-scale analysis on Java and Scala projects, searching for task-parallel workloads suitable for inclusion in a benchmark suite. We identify five candidate workloads (two in Java and three in Scala) that may be used for benchmarking task parallelism on the JVM.

Regarding ongoing research, we are exploring to which extent the testing code executed by NAB is representative for real-world usage scenarios of applications. We are applying automated test-case-generation techniques to increase the amount of dynamically analyzable code. Finally, we are also extending NAB to different repositories (including offline mirrors and datasets) and programming languages.

References

1 S. Alimadadi, D. Zhong, M. Madsen, and F. Tip. Finding Broken Promises in Asynchronous JavaScript Programs. *OOPSLA*, pages 162:1–162:26, 2018.

2 Atlassian. Bitbucket API. https://developer.atlassian.com/bitbucket/api/2/reference/, 2018.

3 R. Bell Jr and L. K. John. The Case for Automatic Synthesis of Miniature Benchmarks. In *MoBS*, pages 4–8, 2005.

4 D. Bernstein. Containers and Cloud: From LXC to Docker to Kubernetes. *IEEE Cloud Computing*, 1(3):81–84, 2014.

5 S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, et al. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *OOPSLA*, pages 169–190, 2006.

6 C. F. Bolz, L. Diekmann, and L. Tratt. Storage Strategies for Collections in Dynamically Typed Languages. In *OOPSLA*, pages 167–182, 2013.

7 R. P.L. Buse and W. R. Weimer. Automatically Documenting Program Changes. In *ASE*, pages 33–42, 2010.

8 B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke. A Systematic Survey of Program Comprehension through Dynamic Analysis. *IEEE Transactions on Software Engineering*, 35(5):684–702, 2009.

9 Oracle Corporation. Java SE HotSpot at a Glance. https://www.oracle.com/technetwork/java/javase/tech/index-jsp-136373.html, 2018.

10 H. J. Curnow and Brian A. Wichmann. A Synthetic Benchmark. *Comput. J.*, 19:43–49, 1976.

11 J. Dietrich, D.J. Pearce, K. Jezek, and P. Brada. Contracts in the Wild: A Study of Java Programs. In *ECOOP*, pages 9:1–9:29, 2017.

12 Y. Ding, M. Zhou, Z. Zhao, S. Eisenstat, and X. Shen. Finding the Limit: Examining the Potential and Complexity of Compilation Scheduling for JIT-based Runtime Systems. In *ASPLOS*, pages 607–622, 2014.

13 L. Eeckhout, K. de Bosschere, and H. Neefs. Performance Analysis Through Synthetic Trace Generation. In *ISPASS*, pages 1–6, 2000.

14 International Organization for Standardization. ISO/IEC 20922:2016. https://www.iso.org/standard/69466.html, 2018.

15 GHArchive. Public GitHub Timeline and Archive. https://www.gharchive.org/, 2018.

16 GitLab. GitLab API. https://docs.gitlab.com/ee/api/, 2018.

17 L. Gong, M. Pradel, and K. Sen. JITProf: Pinpointing JIT-unfriendly JavaScript Code. In *ESEC/FSE*, pages 357–368, 2015.

18 G. Gousios. The GHTorrent Dataset and Tool Suite. In *MSR*, pages 233–236, 2013.

19 HAProxy. HAProxy Community Edition. http://www.haproxy.org/, 2018.

20 B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A Platform for Fine-grained Resource Sharing in the Data Center. In *NSDI*, pages 295–308, 2011.

21 Docker Inc. Docker Compose. https://docs.docker.com/compose/, 2018.

22 Docker Inc. Docker Technology. https://docs.docker.com/, 2018.

23 Docker Inc. Swarm Mode Overview. https://docs.docker.com/engine/swarm/, 2018.

24 GitHub Inc. REST API v3. https://developer.github.com/v3/search/, 2018.

25 MongoDB Inc. mongos. https://docs.mongodb.com/manual/reference/program/mongos/, 2018.

26 MongoDB Inc. Scalable and Flexible document database. https://www.mongodb.com/, 2018.

27 MongoDB Inc. Sharding. https://docs.mongodb.com/manual/sharding/, 2018.

28 Innovative Computing Laboratory (ICL) - University of Tennessee. PAPI. http://icl.utk.edu/papi/, 2017.

29 ECMA International. ECMAScript 2015 Language Specification (ECMA-262 6th Edition). https://www.ecma-international.org/ecma-262/6.0/, 2015.

30 ECMA International. ECMAScript 2017 Language Specification (ECMA-262 8th Edition). `https://www.ecma-international.org/ecma-262/8.0/`, 2017.

31 D. Jin, P. O. N. Meredith, C. Lee, and G. Roşu. JavaMOP: Efficient Parametric Runtime Monitoring Framework. In *ICSE*, pages 1427–1430, 2012.

32 A. Joshi, L. Eeckhout, and L. K. John. The Return of Synthetic Benchmarks. In *SPEC Benchmark Workshop*, pages 1–11, 2008.

33 E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian. The Promises and Perils of Mining GitHub. In *MSR*, pages 92–101, 2014.

34 G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In *ECOOP*, pages 327–353, 2001.

35 O. Legunsen, W. U. Hassan, X. Xu, G. Roşu, and D. Marinov. How Good Are the Specs? A Study of the Bug-Finding Effectiveness of Existing Java API Specifications. In *ASE*, pages 602–613, 2016.

36 Lightbend, Inc. Cluster Specification. `https://doc.akka.io/docs/akka/2.5/common/cluster.html`, 2018.

37 C. V. Lopes, P. Maj, P. Martins, V. Saini, D. Yang, J. Zitny, H. Sajnani, and J. Vitek. DéjàVu: A Map of Code Duplicates on GitHub. *OOPSLA*, pages 84:1–84:28, 2017.

38 C. V. Lopes and J. Ossher. How Scale Affects Structure in Java Programs. In *OOPSLA*, pages 675–694, 2015.

39 Y. Lu, X. Mao, Z. Li, Y. Zhang, T. Wang, and G. Yin. Does the Role Matter? An Investigation of the Code Quality of Casual Contributors in GitHub. In *APSEC*, pages 49–56, 2016.

40 M. Madsen, O. Lhoták, and F. Tip. A Model for Reasoning About JavaScript Promises. *OOPSLA*, pages 86:1–86:24, 2017.

41 L. Marek, A. Villazón, Y. Zheng, D. Ansaloni, W. Binder, and Z. Qi. DiSL: A Domain-specific Language for Bytecode Instrumentation. In *AOSD*, pages 239–250, 2012.

42 P. D. Marinescu, P. Hosek, and C. Cadar. COVRIG: A Framework for the Analysis of Code, Test, and Coverage Evolution in Real Software. In *ISSTA*, pages 93–104, 2014.

43 Mercurial. Mercurial Source Control Management. `https://www.mercurial-scm.org/`, 2018.

44 Microsoft. Microsoft Azure. `https://azure.microsoft.com/en-us/`, 2018.

45 A. Møller and M. I. Schwartzbach. Static Program Analysis, October 2018. Department of Computer Science, Aarhus University, `http://cs.au.dk/~amoeller/spa/`.

46 R. Mudduluru and M. K. Ramanathan. Efficient Flow Profiling for Detecting Performance Bugs. In *ISSTA*, pages 413–424, 2016.

47 S. Nussbaum and J. E. Smith. Modeling Superscalar Processors via Statistical Simulation. In *PACT*, pages 15–24, 2001.

48 OpenJDK. JMH - Java Microbenchmark Harness. `http://openjdk.java.net/projects/code-tools/jmh/`, 2018.

49 M. Oskin, F. T. Chong, and M. Farrens. HLS: Combining Statistical and Symbolic Simulation to Guide Microprocessor Designs. In *ISCA*, pages 71–82, 2000.

50 C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-Directed Random Test Generation. In *ICSE*, pages 75–84, May 2007.

51 S. Rastkar, G. C. Murphy, and G. Murray. Summarizing Software Artifacts: A Case Study of Bug Reports. In *ICSE*, pages 505–514, 2010.

52 B. Ray, D. Posnett, P. Devanbu, and V. Filkov. A Large-scale Study of Programming Languages and Code Quality in GitHub. *CACM*, pages 91–100, 2017.

53 Veselin Raychev, Martin Vechev, and Andreas Krause. Predicting Program Properties from "Big Code". In *POPL*, pages 111–124, 2015.

54 A. Rosà, E. Rosales, and W. Binder. Analyzing and Optimizing Task Granularity on the JVM. In *CGO*, pages 27–37, 2018.

55 O. Rudenko. Best Practices for Using Promises in JS. `https://60devs.com/best-practices-for-using-promises-in-js.html`, 2015.

56 Scala Benchmarking Project. ScalaBench. `http://www.scalabench.org/`, 2018.

57 K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs. Jalangi: A Selective Record-replay and Dynamic Analysis Framework for JavaScript. In *ESEC/FSE*, pages 488–498, 2013.

58 A. Sewe, M. Mezini, A. Sarimbekov, and W. Binder. DaCapo con Scala: Design and Analysis of a Scala Benchmark Suite for the Java Virtual Machine. In *OOPSLA*, pages 657–676, 2011.

59 S. Shamshiri, R. Just, J. M. Rojas, G. Fraser, P. McMinn, and A. Arcuri. Do Automatically Generated Unit Tests Find Real Faults? An Empirical Study of Effectiveness and Challenges (T). In *ASE*, 2015, pages 201–211, 2015.

60 K. Shiv, K. Chow, Y. Wang, and D. Petrochenko. SPECjvm2008 Performance Characterization. In *SPEC Benchmark Workshop on Computer Performance Evaluation and Benchmarking*, pages 17–35, 2009.

61 L. A. Smith, J. M. Bull, and J. Obdrzálek. A Parallel Java Grande Benchmark Suite. In *SC*, pages 6–16, 2001.

62 H. Sun, D. Bonetta, C. Humer, and W. Binder. Efficient Dynamic Analysis for Node.Js. In *CC*, pages 196–206, 2018.

63 The JUnit Team. JUnit. `https://junit.org`, 2018.

64 The DaCapo Benchmark Suite. DaCapo. `http://http://www.dacapobench.org/`, 2018.

65 The Netty project. Netty project. `https://netty.io/`, 2018.

66 L. Van Ertvelde and L. Eeckhout. Benchmark Synthesis for Architecture and Compiler Exploration. In *IISWC*, pages 1–11, 2010.

67 R. P. Weicker. Dhrystone: A Synthetic Systems Programming Benchmark. *Commun. ACM*, 27(10):1013–1030, 1984.

68 T. Würthinger, C. Wimmer, C. Humer, A. Wöß, L. Stadler, C. Seaton, G. Duboscq, D. Simon, and M. Grimmer. Practical Partial Evaluation for High-performance Dynamic Language Runtimes. In *PLDI*, pages 662–676, 2017.

69 Y. Zheng, A. Rosà, L. Salucci, Y. Li, H. Sun, O. Javed, L. Bulej, L. Y. Chen, Z. Qi, and W. Binder. AutoBench: Finding Workloads That You Need Using Pluggable Hybrid Analyses. In *SANER*, pages 639–643, 2016.

MagpieBridge: A General Approach to Integrating Static Analyses into IDEs and Editors

Linghui Luo
Heinz Nixdorf Institute, Paderborn University, Paderborn, Germany
linghui.luo@upb.de

Julian Dolby
IBM Research, New York, USA
dolby@us.ibm.com

Eric Bodden
Heinz Nixdorf Institute, Paderborn University, Paderborn, Germany
Fraunhofer IEM, Paderborn, Germany
eric.bodden@upb.de

Abstract

In the past, many static analyses have been created in academia, but only a few of them have found widespread use in industry. Those analyses which are adopted by developers usually have IDE support in the form of plugins, without which developers have no convenient mechanism to use the analysis. Hence, the key to making static analyses more accessible to developers is to integrate the analyses into IDEs and editors. However, integrating static analyses into IDEs is non-trivial: different IDEs have different UI workflows and APIs, expertise in those matters is required to write such plugins, and analysis experts are not typically familiar with doing this. As a result, especially in academia, most analysis tools are headless and only have command-line interfaces. To make static analyses more usable, we propose MagpieBridge– a general approach to integrating static analyses into IDEs and editors. MagpieBridge reduces the $m \times n$ complexity problem of integrating m analyses into n IDEs to $m+n$ complexity because each analysis and type of plugin need be done just once for MagpieBridge itself. We demonstrate our approach by integrating two existing analyses, Ariadne and CogniCrypt, into IDEs; these two analyses illustrate the generality of MagpieBridge, as they are based on different program analysis frameworks – WALA and Soot respectively – for different application areas – machine learning and security – and different programming languages – Python and Java. We show further generality of MagpieBridge by using multiple popular IDEs and editors, such as Eclipse, IntelliJ, PyCharm, Jupyter, Sublime Text and even Emacs and Vim.

2012 ACM Subject Classification Software and its engineering → Software notations and tools

Keywords and phrases IDE, Tool Support, Static Analysis, Language Server Protocol

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2019.21

Category Tool Insights Paper

Funding This research was supported by the research training group "Human Centered Systems Security" (NERD.NRW) sponsored by the state of North-Rhine Westphalia in Germany and by the DFG's collaborative research center 1119 CROSSING.

1 Introduction

Many static analyses have been created to find a wide range of issues in code. Given the prominence of security exploits in practice, many analyses focus on security, such as TAJ [59], Andromeda [58], HybriDroid [34], FlowDroid [31], CogniCrypt [48] and DroidSafe [44]. There are also many analyses that address other code quality issues, such as FindBugs [46], SpotBugs [23], PMD [17] for common programming flaws (e.g. unused variables, dead code, empty catch blocks, unnecessary creation of objects, etc.) and TRACKER [57] for resource

33rd European Conference on Object-Oriented Programming (ECOOP 2019).
Editor: Alastair F. Donaldson; Article No. 21; pp. 21:1–21:25
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

leaks. Other analyses target code performance, such as J2EE transaction tuning [41]. There are also specialized analyses for specific domains, such as Ariadne [38] for machine learning. These analyses collectively represent a large amount of work, as they embody a variety of advanced analyses for a range of popular programming languages. To make this effort more tractable, many analyses are built on existing program analysis frameworks that provide state-of-the-art implementations of commonly-needed building blocks such as call-graph construction, pointer analysis, data-flow analysis and slicing, which in turn all rest on an underlying abstract internal representation (IR) of the program. Doop [7,33], Soot [21,49], Safe [19], Soufflé [22] and WALA [29] are well-known.

While development of these analyses has been a broad success of programming language research, there has been less adoption of such analyses in tools commonly used by developers, i.e., in interactive development environments (IDEs) such as Eclipse [8], IntelliJ [13], PyCharm [18], Android Studio [1], Spyder [24] and editors such as Visual Studio Code [28], Emacs [10], Atom [3], Sublime Text [26], Monaco [16] and Vim [27]. There have been some positive examples: the J2EE transaction analysis shipped in IBM WebSphere [12], Andromeda was included in IBM Security AppScan [2], both ultimately based on Eclipse technology. Similarly, CogniCrypt comprises an Eclipse plugin that exposes the results of its crypto-misuse analysis directly to the developer within the IDE. Each of these tools involved a substantial engineering effort to integrate a specific analysis for a specific language into a specific tool. Table 1 shows the amount of code in plugins for analyses is a significant fraction of code in the analysis itself. Given that degree of needed effort, the sheer variety of popular tools and potentially-useful analyses makes it impractical to build every combination.

Table 1 Comparison between the LOC (lines of Java code) for analysis and the LOC for plugin.

Tool	Analysis (LOC)	Plugin (LOC)	Plugin/Analysis
FindBugs	132,343	16,670	0.13
SpotBugs	121,841	16,266	0.13
PMD	117,551	33,435	0.28
CogniCrypt	11,753	18,766	1.60
DroidSafe	41,313	8,839	0.21
Cheetah	4,747	864	0.18
SPLlift	1,317	3,317	2.52

While the difficulty of integrating such tools into different development environments has lead to poor adoption of these tools and research results in practice, it also makes empirical evaluations of them challenging. Evaluations of static analyses have been mostly restricted to automated experiments where the analyses are run in "headless" mode as command-line tools [31, 50, 53, 62], paying little to no attention to usability aspects on the side of the developer. As many recent studies show [35, 36, 47], however, those aspects are absolutely crucial: if program analysis tools do not yield actionable results, or if they do not report them in a way that developers can understand, then the tools will not be adopted. So to develop and evaluate such tools, researchers need ways to bring tools into IDEs more easily and quickly.

The ideal solution is the magic box shown in Figure 1, which adapts any analysis to any editor,[1] and presents the results computed by the analysis, e.g., security vulnerabilities or other bugs, using common idioms of the specific tool, e.g., problem lists or hovers.

[1] Note: In the following, when we write *editor*, we mean any code editor, which comprises IDEs.

In this work, we present MagpieBridge,[2] a system which uses two mechanisms to realize a large fraction of this ultimate goal:

1. Since many analyses are written using program analysis frameworks, MagpieBridge can focus on supporting the core data structures of these frameworks. For instance, analyses based on data-flow frameworks can be supported if the magic box can render their data-flow results naturally. Furthermore, while there are multiple frameworks, they share many common abstractions such as data flow and call graphs, which allows one to support multiple frameworks with relative ease.
2. More and more editors support the Language Server Protocol (LSP) [15], a protocol by which editors can obtain information from arbitrary "servers". LSP is designed in terms of idioms common to IDEs, such as problem lists, hovers and the like. Thus, the magic box can take information from a range of analyses and render it in a few common tooling idioms. LSP support in each editor then displays these in the natural idiom of the editor.

Our system MagpieBridge exploits these two mechanisms to implement the magic box for analyses built using WALA or Soot, with more frameworks under development, and for any editor that supports the LSP. In this paper, we present the MagpieBridge workflow, explaining the common APIs we defined for enabling integration. We demonstrate two existing analyses – CogniCrypt and Ariadne, which are based on different frameworks (Soot and WALA), for different application areas (cryptography misuses and machine learning) and for different programming languages (Java and Python) into multiple popular IDEs and editors (Eclipse, Visual Studio Code, PyCharm, IntelliJ, JupyterLab, Monaco, Vim, Atom and Sublime Text) supporting different features (diagnostics, hovers and code lenses) using MagpieBridge. We make MagpieBridge publicly available as `https://github.com/MagpieBridge/MagpieBridge`.

[2] In a Chinese legend, a human and a fairy fall in love, but this love angers the gods, who separate them on opposite sides of the Milky Way. However, on the seventh day of the seventh lunar month each year, thousands of magpies form a bridge, called 鹊桥 in Chinese and Queqiao in pinyin, allowing the lovers to meet.

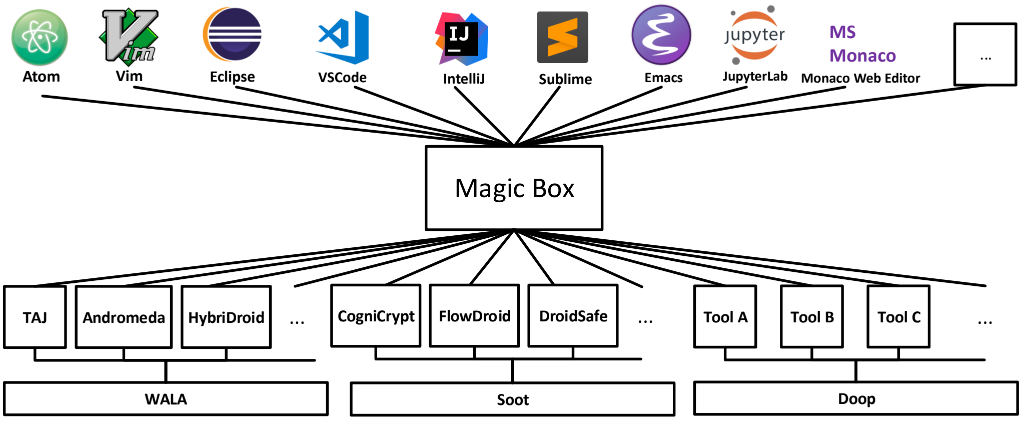


Figure 1 The desired solution: a magic box that connects arbitrary static analyses to arbitrary IDEs and editors.

2 Background and Related Work

Existing tools and frameworks

Given the importance of programming tools for IDEs, there have been a variety of efforts to provide them, both commercial and open source. We here survey some significant ones, focusing on those that use WALA [40] or Soot [49,60] and hence are most directly comparable to our work.

There have been a few commercial tools, notably IBM AppScan [2] and RIGS IT Xanitizer [30]. Both products make use of WALA and target JavaScript among other languages. They comprise views to display analysis results as annotations to the source code, and allow for some triaging of the often longish lists of potential vulnerabilities within the IDEs. Among other issues, AppScan finds tainted flows and allows the user to focus on a specific flow through the program, although the user needs to decide what flow is of interest.

There has been a wider variety of open-source tools. WALA has been used in e.g. JOANA [43,45]. Soot is used in the widely adopted open-source crypto-misuse analyzer Eclipse CogniCrypt [48], and is also part of the research tools Cheetah [36], SPLlift [32] and DroidSafe [44]. All tools named so far integrate with the Eclipse IDE.

JOANA focuses on Java, including Android, and provides a range of advanced analyses based on information flow control.

CogniCrypt is a tool to detect misuses of cryptographic APIs in Java and Android applications. Its current UI integration is relatively basic, offering simple error annotations in the program code and the problems view. CogniCrypt further comprises an XText-based [39] Eclipse plugin that allows developers to edit API-specification files using syntax highlighting and code completion. Those specification files directly determine the definition of the static analysis.

SPLlift is a research tool to analyze Java-based software product lines. Its UI is an extension to FeatureIDE [56], which allows it to show variations in the product line's code base through color coding. Detected programming errors are shown as code annotations and in the problems view. FeatureIDE itself is also an extension to Eclipse.

Cheetah is a research prototype for the just-in-time static taint analysis within IDEs. In Cheetah, the analysis is triggered upon saving a source-code file, but in its case the analysis is automatically prioritized to provide rapid updates to the error messages in those code regions that are in the developer's current scope. From there the analysis works its way outwards, potentially reporting errors in farther parts of the program only after several seconds or even minutes. Due to this mechanism, Cheetah requires the IDE to provide information about which file edit caused the analysis to be triggered, and what the project layout looks like. Cheetah also provides a somewhat richer UI integration than the previously named tools. For instance, when users select an individual taint-flow message in the problems view, it highlights in the code all statements involved in that particular taint, and also shows a list of those statements in a separate view – useful in case those are scattered across multiple source code files.

Analysis based on Doop [7,33] has been experimentally integrated into the ProGuard optimizer for Android applications [61]. This is a once-off integration rather than a framework for Doop analyses, and it is focused on the build processs rather than the IDE itself. Still, it reflects the special-purpose integrations that show how analysis tends to be used.

Until now, program-analysis frameworks have focused on making it easier to develop analyses, with supportive infrastructure for basics such as scalable call graph, pointer analysis, and data-flow analysis. There have been presentations[3] and tutorials[4] at conferences which have provided both introductions and detailed tutorials for analysis construction; however, until now, there has been little focus on assisting with integrating such analyses into usable tools.

Language Server Protocol (LSP)

The Language Server Protocol (LSP) [15] is a JSON-based RPC protocol originally developed by Microsoft for its Visual Studio Code to support different programming languages. LSP follows a client/server architecture, in which "clients" are typically meant to be code editors, i.e., IDEs such as IntelliJ, Eclipse, etc., or traditional editors such as Visual Studio Code, Vim, Emacs or Sublime Text. Those clients can trigger certain actions in "servers", e.g. by opening a source-code file. Those servers can be of different flavours, but LSP allows them to contribute certain contents to the editor's user interface, such as code annotations, list items or hovers. We will give concrete examples, including screenshots, in Section 4. As we show in this work, the LSP's design lends itself to implement static code analysis tools as servers. In such a design, clients trigger analysis servers through LSP, and those servers communicate back their results through LSP as well, causing analysis results to automatically be shown in the client through the respective editor's native interfaces.

SASP and SARIF

The Static Analysis Server Protocol (SASP) [25], although similar in name to LSP, is a distinctly different protocol. Started in 2017 by the static code analysis vendor GrammaTech, it describes a standardized communication protocol to facilitate communication between static analysis tools and consumers of their results. Compared to LSP, it supports a richer data-exchange format that is explicitly fine-tuned to static analysis. This is realized through the Static Analysis Results Interchange Format (SARIF) [20,25] that SASP uses to communicate static-analysis results from servers to clients. Generally, SASP therefore promises a more tight coupled integration compared to LSP static analyses into editors, potentially needing more work on the server. Also, as of now, SASP and SARIF have seen little adoption by tool vendors. Currently, the standard is mostly put forward by GrammaTech, which through SASP offers third-party static analysis tools to allow a triaging of those tools' results in GrammaTech's CodeSonar [5]. SARIF exporters currently exist for some few static analysis tools, including CogniCrypt [48], the Clang Static Analyzer [4], Cppcheck [6], and Facebook Infer [11], which makes them amenable for an integration through SASP. However, right now, CodeSonar appears to be the only client ready to consume SARIF results, and it is unclear whether this will change in the near future. It is for this reason that MagpieBridge builds, for now, on top of LSP instead of SASP and SARIF. Furthermore, SASP is currently still in the early stage of its development and there exists no formal specification of the protocol [25], which makes it hard to compare it to LSP in detail and hard to use for our work.

[3] e.g. `https://souffle-lang.github.io/pdf/SoufflePLDITutorial.pdf`

[4] e.g. `http://wala.sourceforge.net/wiki/index.php/Tutorial`

3 Approach

3.1 The MagpieBridge Workflow

MagpieBridge uses the Language Server Protocol to integrate program analyses into editor and IDE clients. MagpieBridge is implemented using the Eclipse LSP4J [9] LSP implementation based on JSON-RPC [14], but MagpieBridge hides LSP4J details and presents an interface in terms of high-level analysis abstractions. The overall workflow is shown in Figure 2.

There are multiple mechanisms by which LSP-based tools can be used, but the most common mechanism is that an IDE or editor is configured to launch any desired tools. Each tool is built as a jar file based on the MagpieServer, with a main method that creates a `MagpieServer` (Listing 1), then adds the desired program analyses (`ServerAnalysis` in Listing 2) with `addAnalysis`, and then launches `MagpieServer` with `launch` so that it receives messages. This is shown with the `addAnalysis` and `launch` edges in Figure 2. With such a jar, MagpieBridge can be used simply by configuring an editor to launch it. Figure 3 shows our Sublime Text setup to launch both Ariadne and CogniCrypt analyses. The user merely obtains jar files of the analyses and sets up Sublime Text to launch each of them for the appropriate languages. That is all the setup that is needed.

Based on LSP4J, there are several mechanisms for sending and receiving messages. Most clients/editors simply launch the server and then expect it to handle messages using standard I/O (e.g. Eclipse, IntelliJ, Emacs and Vim); however some clients expect to talk using a well-known socket (e.g. Spyder), Web-based tools communicate using WebSockets (e.g.

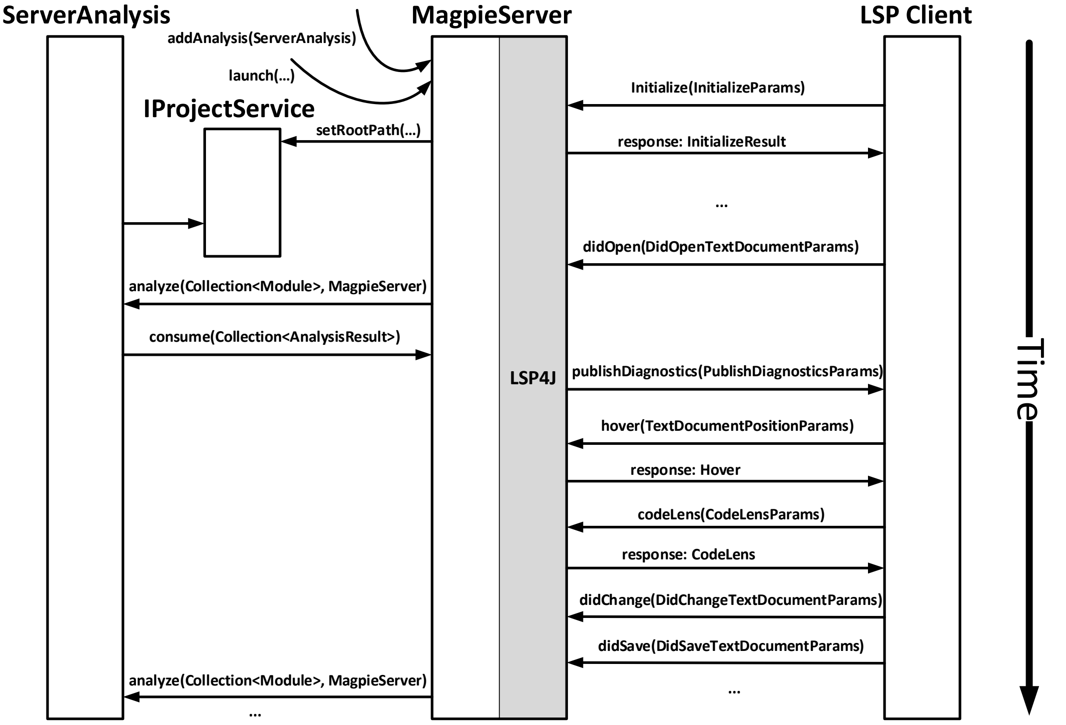
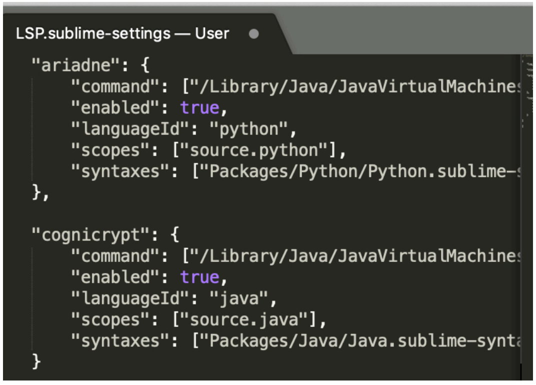

ServerAnalysis
addAnalysis(ServerAnalysis)
launch(...)
IProjectService
setRootPath(...)
analyze(Collection<Module>, MagpieServer)
consume(Collection<AnalysisResult>)
analyze(Collection<Module>, MagpieServer)
...
MagpieServer
LSP4J
LSP Client
Initialize(InitializeParams)
response: InitializeResult
...
didOpen(DidOpenTextDocumentParams)
publishDiagnostics(PublishDiagnosticsParams)
hover(TextDocumentPositionParams)
response: Hover
codeLens(CodeLensParams)
response: CodeLens
didChange(DidChangeTextDocumentParams)
didSave(DidSaveTextDocumentParams)
...
Time

Figure 2 Overall MagpieBridge workflow.



LSP.sublime-settings — User

```
"ariadne": {
    "command": ["/Library/Java/JavaVirtualMachines
    "enabled": true,
    "languageId": "python",
    "scopes": ["source.python"],
    "syntaxes": ["Packages/Python/Python.sublime-s
},

"cognicrypt": {
    "command": ["/Library/Java/JavaVirtualMachines
    "enabled": true,
    "languageId": "java",
    "scopes": ["source.java"],
    "syntaxes": ["Packages/Java/Java.sublime-synta
}
```

Figure 3 Configuration for Sublime Text to launch MagpieServer.

Jupyter and Monaco) and only few tools support both standard I/O and socket (e.g. Visual Studio Code). Our `MagpieServer` supports all these channels out of the box and can be configured to communicate with a client using any of the channels.

Once `MagpieServer` is launched, it interacts with the client tool using standard LSP mechanisms:

- The first step is initialization. The client sends an `initialize` message to the server, which includes information about the project being analyzed, such as its project root path. `MagpieServer` calls `setRootPath` on each `IProjectService` (service that resolves project scope such as source code path and library code path) instance to initialize project path information. MagpieBridge currently understands Eclipse, Maven and Gradle projects. `MagpieServer` also sends the response `InitializeResult` which declares its capabilities back to the client. This is shown in the upper portion of Figure 2
- Subsequently, the client informs `MagpieServer` whenever it works with a file: the `didOpen`, `didChange` and `didSave` messages are sent to the server whenever files are opened, edited and saved respectively. These messages allow MagpieBridge to call the analysis via the `analyze` method whenever anything changes. Each analysis server decides the granularity of when it actually runs analysis and how much analysis it does. This is shown with the `didOpen` and `analyze` edges in Figure 2
- As shown in the rest of Figure 2, analysis uses the `consume` method to report analysis results of type `AnalysisResult` (Listing 4) to `MagpieServer`, which handles them via the appropriate LSP mechanism, specified by the `kind` method (Listing 4), which returns a `Kind` (Listing 5):

  Diagnostic denotes issues found in the code, corresponding to lists of errors and warnings that might be reported by a compiler. Tools typically report them either in a list of results or highlight the results directly in the code. When the program analysis provides such results via `consume`, `MagpieServer` reports them to the client tool with the LSP `publishDiagnostics` API.

  Hover denotes annotations to be displayed for a specific program variable or location. It could be used to report e.g. the type of a variable or the targets of a function call. Tools often show them when the cursor highlights a specific location. When the program analysis provides such results via `consume`, `MagpieServer` keeps them and reports them to the client tool as responses to LSP `hover` API calls by the client tool.

CodeLens denotes information to be added inline in the source code, analogous to generated comments. Tools typically report them as distinguished lines of text inserted between lines of source code. When the program analysis provides such results via `consume`, `MagpieServer` keeps them and reports them to the client tool as responses to LSP `codeLens` API calls by the client tool.

These analysis results have a `position` method that returns a `Position` (Listing 6) denoting the source location to which the result pertains. The result requires a precise location based on starting and ending line and column numbers, which is required by the LSP protocol. Note that the `Position` of MagpieBridge implements the Java `Comparable` interface; MagpieBridge exploits this to store analysis results in `NavigableMap` structures so that it can find the nearest result if a user hovers in a location near result, e.g. some whitespace immediately after a variable or expression.

```java
public class MagpieServer implements LanguageServer, LanguageClientAware{
    protected LanguageClient lspClient;
    protected Map<String, IProjectService> languageProjectServices;
    protected Map<String, Set<ServerAnalysis>> languageAnalyses;

    public void addProjectService(String language, IProjectService projectService){...}
    public void addAnalysis(String language, ServerAnalysis analysis){...}
    public void doAnalyses(String language){...}
    public void consume(Collection<AnalysisResult>){...}

    protected Consumer<AnalysisResult> createDiagnosticConsumer(){...}
    protected Consumer<AnalysisResult> createHoverConsumer(){...}
    protected Consumer<AnalysisResult> createCodeLensConsumer(){...}
    ...
}
```

Listing 1 The core of the server.

```java
public interface ServerAnalysis{
    public String source();
    public void analyze(Collection<Module> files, MagpieServer server);
}
```

Listing 2 Interface for defining analysis on the server.

```java
public interface IProjectService {
    public void setRootPath(Path rootPath);
}
```

Listing 3 Interface for defining service which resolves project scope.

3.2 The MagpieBridge System

We explain our MagpieBridge system with an overview in Figure 4. MagpieBridge needs to support various analysis tools that were built on top of different frameworks, e.g., TAJ, Andromeda and HybriDroid use WALA, while CogniCrypt, FlowDroid and DroidSafe rely on Soot and many other analyses are based on Doop. These analysis frameworks have different IRs, which MagpieBridge needs to use to generate analysis results. One key requirement for all the frameworks supported by MagpieBridge is very precise source-code

```java
public interface AnalysisResult {
    public Kind kind();
    public String toString(boolean useMarkdown);
    public Position position();
    public Iterable<Pair<Position,String>> related();
    public DiagnosticSeverity severity();
    public Pair<Position, String> repair();
}
```

Listing 4 Interface for defining analysis result.

```java
public enum Kind {
    Diagnostic, Hover, CodeLens
}
```

Listing 5 Enum for defining kinds of analysis results.

```java
public interface Position extends Comparable {
    public int getFirstLine();
    public int getLastLine();
    public int getFirstCol();
    public int getLastCol();
    public int getFirstOffset();
    public int getLastOffset();
    public URL getURL();
}
```

Listing 6 Interface for defining position.

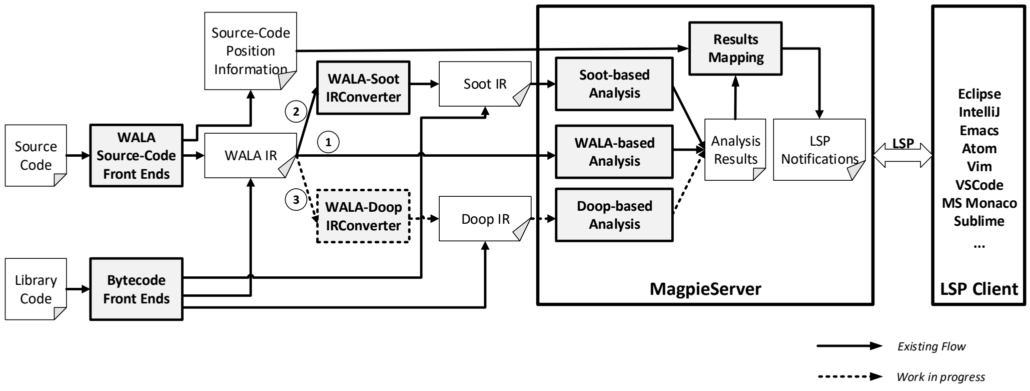


Figure 4 Overview of our MagpieBridge system.

mappings, since in LSP all the messages communicate using starting and ending line and column numbers. In the following we explain how MagpieBridge achieves this requirement for WALA-based analyses, Soot-based analyses and Doop-based analyses respectively.

3.2.1 WALA-based Analysis

The simplest code path in MagpieBridge (flow ① in Figure 4) uses WALA source language front ends for creating IR on which to perform analysis. WALA comprises both bytecode and source-code front ends for different languages (Java, Python and JavaScript), and the source-code front end preserves source-code positions very well. This information can be consumed later in the LSP notifications, since it is kept in WALA's IR. WALA's IR is a traditional three-address code in Static Single Assignment (SSA) form, which is translated from WALA's Common Abstract Syntax Tree (CAst).

The approach to source-code front ends for WALA is using existing infrastructure for each supported language: Eclipse JDT for Java, Mozilla Rhino for JavaScript and Jython for Python. Each of these front ends is maintained with respect to its respective language standards, and all the front ends provide precise mappings of source locations for constructs. To provide detailed source mapping for the generated IR, each WALA function body has an instance of `DebuggingInformation` (Listing 7) which allows MagpieBridge to map locations from requests to IR elements at a very fine level.

```
public interface DebuggingInformation {
    Position getCodeBodyPosition();
    Position getCodeNamePosition();
    Position getInstructionPosition(int instructionOffset);
    String[][] getSourceNamesForValues();
    Position getOperandPosition(int instructionOffset, int operand);
    Position getParameterPosition(int param);
}
```

Listing 7 Debugging information interface.

Listing 7 details how much source mapping information is available. `getCodeBodyPosition` is the source range of the entire function, and `getCodeNamePosition` is the position of just the name in the body. `getInstructionPosition` is the source position of a given IR instruction. `getOperandPosition` is the source position of a given operand in an IR instruction. `getParameterPosition` is the position of a given parameter declaration in the source.

3.2.2 Soot-based Analysis

Soot comprises a solid Java bytecode front end. The bytecode only has the line number of each statement. This is not sufficient to support features such as hover, fix and codeLens in an editor. For those features, position information about variable, expressions, calls and parameters are necessary. However, they are lost in the bytecode. Soot further comprises source-code front ends. Such front ends, however, require frequent updates due to the frequently changing specification of the Java source language, which has caused Soot's source-code front ends to become outdated. Besides, Soot IR was not designed to keep precise source-code position information, e.g., there is no API for getting the parameter position in a method. Our approach is to take WALA's source-code front end to generate WALA IR and convert it to Soot IR. Soot has multiple IRs, the most commonly used IR is called Jimple [60]. Jimple is also a three-address code and has Java-like syntax, but is simpler, e.g., no nested statements. Opposed to WALA IR, Jimple is not in SSA-form. Both WALA and Soot are implemented in Java and manipulate the IR through Java objects. This makes the conversion between the IRs feasible. In particular, we have implemented the WALA-Soot IRConverter and defined the common APIs (Listing 4) to encode analysis results, as well as the `MagpieServer` (Listing 1) that hosts the analysis. Currently the WALA-Soot IRConverter only converts WALA IR generated by WALA's Java source-code front end. In fact, WALA uses a pre-IR before generating the actual WALA IR in SSA-form, and this non-SSA pre-IR is actually the IR that we convert to Jimple. Since also Jimple is not in SSA, this conversion is more direct. This pre-IR contains 24 different instructions as shown in Figure 5. After studying both IRs, we found out that 15 instructions in WALA IR can be converted to JAssignStmt in Jimple. Most of the times the conversion is one-to-one, only a few cases are one-to-many. The precise source-code position information from WALA IR is encapsulated in the tags (annotations) of the converted Soot IR. In the future, we plan to convert WALA IR from front ends of other languages such as Python and JavaScript to a potentially extended version of the Soot IR.

The flow ② in Figure 4 for integrating Soot-based analysis starts by dividing the analyzed program code into application source code and library code (which can be in binary form). The source code is parsed by one of WALA's source-code front end and it outputs WALA IR, as well as precise source code position information associated in the IR. For a Soot-based analysis, the WALA IR is translated by a WALA-Soot IRConverter into Soot IR

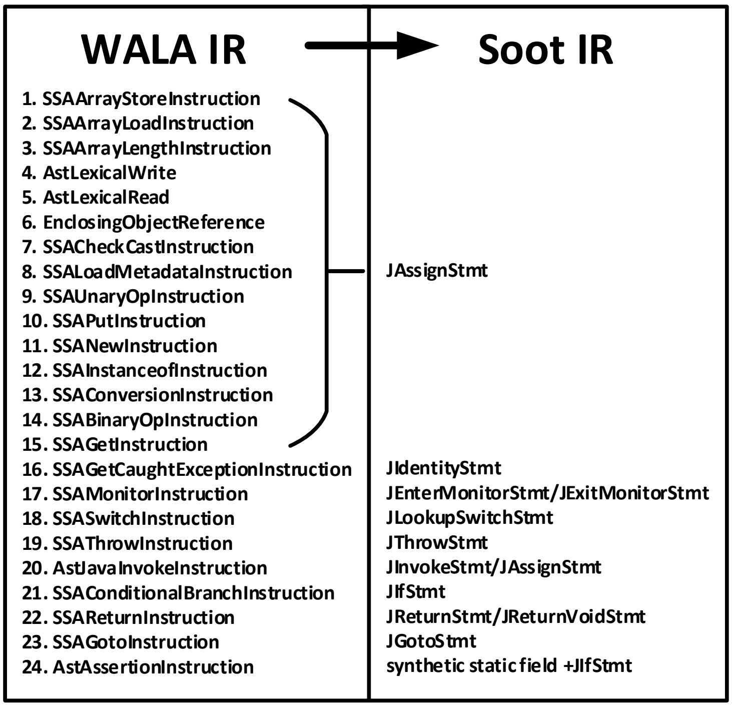

WALA IR
Soot IR
1. SSAArrayStoreInstruction
2. SSAArrayLoadInstruction
3. SSAArrayLengthInstruction
4. AstLexicalWrite
5. AstLexicalRead
6. EnclosingObjectReference
7. SSACheckCastInstruction
8. SSALoadMetadataInstruction
9. SSAUnaryOpInstruction
10. SSAPutInstruction
11. SSANewInstruction
12. SSAInstanceofInstruction
13. SSAConversionInstruction
14. SSABinaryOpInstruction
15. SSAGetInstruction
JAssignStmt
16. SSAGetCaughtExceptionInstruction
JIdentityStmt
17. SSAMonitorInstruction
JEnterMonitorStmt/JExitMonitorStmt
18. SSASwitchInstruction
JLookupSwitchStmt
19. SSAThrowInstruction
JThrowStmt
20. AstJavaInvokeInstruction
JInvokeStmt/JAssignStmt
21. SSAConditionalBranchInstruction
JIfStmt
22. SSAReturnInstruction
JReturnStmt/JReturnVoidStmt
23. SSAGotoInstruction
JGotoStmt
24. AstAssertionInstruction
synthetic static field +JIfStmt

Figure 5 Conversion from WALA IR to Soot IR.

```
public class ExampleAnalysis implements ServerAnalysis{

    @Overide
    public String source(){
        return "Example Analysis"
    }

    @Overide
    public void analyze(Collection<Module> sources, MagpieServer server){
        ExampleTransformer t = getExampleTransformer();
        loadSourceCodeWithWALA(sources);
        JavaProjectService service = (JavaProjectService)
             server.getProjectService("java");
        loadLibraryCodeWithSoot(service.getLibraryPath());
        runSootPacks(t);
        List<AnalysisResult> results = t.getAnalysisResults();
        server.consume(results);
    }
    ...
}

public class Example{

    public static void main(String... args){
        MagpieServer server = new MagpieServer();
        IProjectService service = new JavaProjectService();
        ExampleAnalysis analysis = new ExampleAnalysis();
        String language = "java";
        server.addProjectService(language, service);
        server.addAnalysis(language, analysis);
        server.launch(...);
    }
}
```

Listing 8 The MagpieServer runs a Soot-based analysis.

(Jimple). The library code is parsed by Soot's bytecode front end and then complements the program's IR obtained from the source code. The Soot IR in Figure 4 thus consists of two parts: Jimple converted by the WALA-Soot IRConverter, which represents the source-code portion/application code of the program, and Jimple generated by Soot's bytecode front end which represents the library code. Based on the composite Soot IR, Soot further conducts a call graph and optionally also pointer analysis, which can then be followed by arbitrary data-flow analyses.

Listing 8 shows an example of running a Soot-based analysis `ExampleTransformer` (analyses are called transformers in Soot) on the `MagpieServer`. The `ExampleTransformer` accesses the program through the singleton object `Scene` in order to analyze the program. Once the `MagpieServer` receives the source code, the method `loadSourceCodeWithWALA` parses the source code, converts it to Soot IR with the WALA-Soot IRConverter and stores the IR in the `Scene`. The class `JavaProjectService` resolves library path for the current project. `loadLibraryCodeWithSoot` loads the necessary library code from the path and adds the IR into `Scene`. The method `runSootPacks` invokes Soot to build call-graph and run the actual analysis. The analysis results will be then consumed by the server. In this example, only the source files sent to the server are analyzed together with the library code. However, it can be configured to perform a whole-program analysis, since the source code path can also be resolved by `JavaProjectService`.

We explain how the class `JavaProjectService` which implements `IProjectService` resolves the full Java project scope, i.e., source code path and library code path. As specified in LSP, the editors send the project root path (rootURI) to the server in the first request `initialize`. Library and source code path can be resolved by using the build-tool dependency plugins (e.g. caching results of mvn dependency:list) or parsing the configuration (e.g. pom.xml, build.gradle) and source code files located in the root path. Project structure conventions for different kinds of projects are also considered in MagpieBridge. For more customized projects, MagpieBridge also allows the user to specify the library and source code path manually as program arguments.

3.2.3 Doop-based Analysis

Doop uses Datalog to allow for declarative analysis specifications, encoding instructions as Datalog relations as well as instruction source positions. There is code to convert from the WALA Python IR to Datalog, and that captures both the semantics of statements as well as source mapping, and these declarations capture the information needed for analysis tool support. For instance, there is a Datalog relation that captures instruction positions and is generated directly from WALA IR:

```
.decl Instruction_SourcePosition(?insn:Instruction,
  ?startLine:number, ?endLine:number, ?startColumn:number, ?endColumn:number)
```

This code has been used experimentally for analysis using Doop of machine code written in Python. This code path could be used to express analyses in editors using MagpieBridge, and such work is under development.

4 Demonstration

To make MagpieBridge more concrete, we use two illustrative analyses, based on different frameworks – Soot and WALA, respectively – for different languages – Java and Python – in different domains – security and bug finding – both in a range of editors:

CogniCrypt analyzes how cryptographic APIs are used in a program, and reports a variety of vulnerabilities such as encryption protocols being misused or when protocols are used in situations where they should not. The tool then also gives suggestions on how to fix the problem. CogniCrypt comprises a highly efficient demand-driven, inter-procedural data-flow analysis [55] based on Soot, and has its own Eclipse-based plugin. As Table 1 shows, its plugin actually required substantially more code than the analysis itself. The plugin also is limited to Eclipse. We illustrate what it looks like to use CogniCrypt in multiple tools using MagpieBridge. To keep exposition simple, we focus on a case in which a weak encryption mode is used (Electronic Codebook Mode, ECB). In the general case the analysis can also report complex flows through the program. Screenshots in Figure 6, Figure 7, Figure 8 and Figure 9 show the crypto warning reported by CogniCrypt in different editors. As we can see, only the call `Cipher.getInstance` with the insecure parameter is marked in each editor.

Ariadne analyzes how tensor (multi-dimensional array) data structures are used in machine-learning code written in Python, and reports a range of information. It presents basic tensor-shape information for program variables, and finds and fixes certain kinds of program bugs. A key operation is reshaping a tensor: the `reshape` operation takes a tensor and a new shape, and returns a new tensor with the desired shape when that is possible. To simplify complex tensor semantics, a tensor can be reshaped only when its total size is equal to size of the desired new shape. Another operation is performing a convolution, e.g. `conv2d`, which requires the input tensor to have a specific number of dimensions. We illustrate cases of these bugs, and how they are shown in multiple editors (Figure 10, Figure 11, Figure 12, Figure 13, and Figure 14).

We illustrate how the aspects of LSP used by MagpieBridge are rendered in a variety of editors; while there are common notions such as a list of diagnostics, different tools make different choices in how those elements are displayed. We describe in turn several LSP aspects and how analysis information is displayed using them.

4.1 Diagnostics

The most straightforward interface is for an analysis to report a set of issues, but even this simple concept is handled differently in different editors.

- Some editors have a problem view, i.e., a list summarizing all outstanding issues. An example of this interface is Sublime Text, illustrated in Figure 8 where a warning about weak encryption is shown in a list.
- Some editors do not have such a list, but choose to highlight issues directly in the code. An example of this interface is Monaco, illustrated in Figure 7; the same warning about weak encryption is shown inline. To minimize clutter, editors typically make such warnings as hovers, and we show it displayed in Monaco. A somewhat different visualization of the same idea is in Figure 13, in which Atom shows an invalid use of `reshape` in Tensorflow.
- Some editors do both. An example of this interface is Eclipse, illustrated in Figure 6 where a warning about weak encryption is shown both inline and in a list. Again to minimize clutter, the inline message is realized via a hover.

Note that all issues displayed here are computed by the very same analysis in all editors and rendered as the same LSP objects; however, they appear natural in each editor, due to the editor-specific LSP client implementations.

4.2 Code Lenses

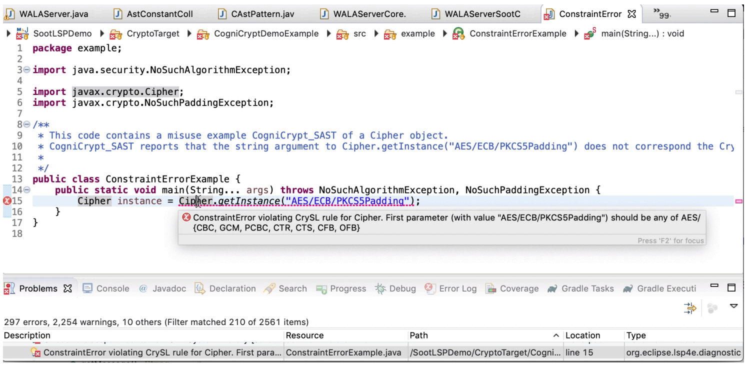
Code lenses look like comments, but are inserted into the code by analyses and are used to reflect generally-useful information about the program. An example is shown in Figure 10, in which the shapes of tensors are listed explicitly for various program variables and function arguments.

4.3 Hovers

Hovers are used to reflect generally-useful information about the program, but, unlike code lenses, they are visible only on demand. As such, an analysis can sprinkle them liberally in the program and they will not be distracting since they are only visible when needed. Different tools have different ways of user interaction. In Figure 11, the user hovers over the variable `x_dict` in PyCharm to reveal the shape of tensors that it holds. In Figure 12, the user enters a Vim command with the cursor over the variable `x_dict`.

4.3.1 Repairs

LSP provides the ability to specify fixes for diagnostics; a diagnostic can specify replacement text for the text to which the given diagnostic applies. The method `repair()` in the interface `AnalysisResult` is designed exactly for this purpose (see Listing 4). Figure 14 shows an example of this: the top half shows an error report in Visual Studio Code that a call to `conv2d` is invalid, since such calls require a tensor with four dimensions whereas the provided argument has only 2. However, the analysis determines that a plausible fix is to `reshape` the provided argument to have more dimensions, and the lower part of the figure shows a prompt, in Emacs, suggesting a `reshape` call to insert.



```java
package example;

import java.security.NoSuchAlgorithmException;

import javax.crypto.Cipher;
import javax.crypto.NoSuchPaddingException;

/**
 * This code contains a misuse example CogniCrypt_SAST of a Cipher object.
 * CogniCrypt_SAST reports that the string argument to Cipher.getInstance("AES/ECB/PKCS5Padding") does not correspond the Cry
 *
 */
public class ConstraintErrorExample {
    public static void main(String... args) throws NoSuchAlgorithmException, NoSuchPaddingException {
        Cipher instance = Cipher.getInstance("AES/ECB/PKCS5Padding");
    }
}
```

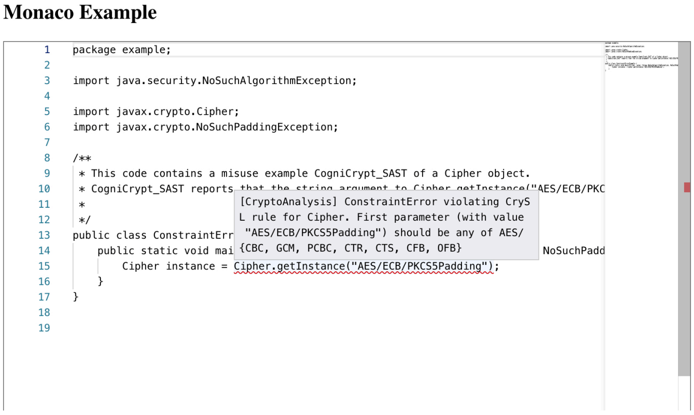
ConstraintError violating CrySL rule for Cipher. First parameter (with value "AES/ECB/PKCS5Padding") should be any of AES/{CBC, GCM, PCBC, CTR, CTS, CFB, OFB}

Press 'F2' for focus

297 errors, 2,254 warnings, 10 others (Filter matched 210 of 2561 items)

Description	Resource	Path	Location	Type
ConstraintError violating CrySL rule for Cipher. First para...	ConstraintErrorExample.java	/SootLSPDemo/CryptoTarget/Cogni...	line 15	org.eclipse.lsp4e.diagnostic

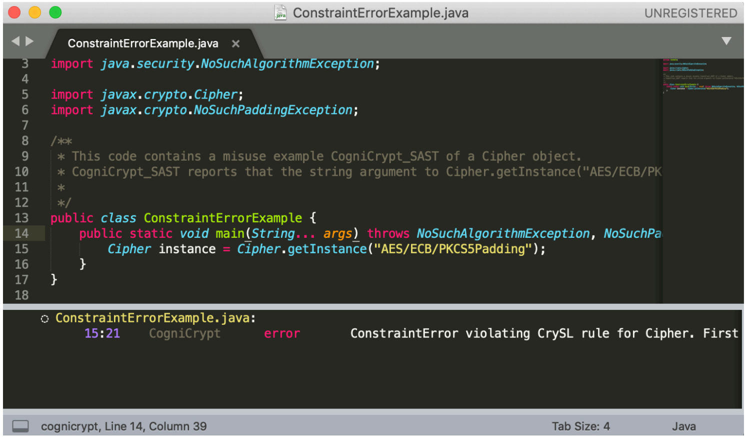
Figure 6 Insecure crypto warning in Eclipse.


Monaco Example

```java
package example;

import java.security.NoSuchAlgorithmException;

import javax.crypto.Cipher;
import javax.crypto.NoSuchPaddingException;

/**
 * This code contains a misuse example CogniCrypt_SAST of a Cipher object.
 * CogniCrypt_SAST reports that the string argument to Cipher.getInstance("AES/ECB/PKC
 *
 */
public class ConstraintErr
    public static void mai                                             NoSuchPadd
        Cipher instance = Cipher.getInstance("AES/ECB/PKCS5Padding");
    }
}
```

[CryptoAnalysis] ConstraintError violating CrySL rule for Cipher. First parameter (with value "AES/ECB/PKCS5Padding") should be any of AES/{CBC, GCM, PCBC, CTR, CTS, CFB, OFB}

Figure 7 Insecure crypto warning in Monaco.

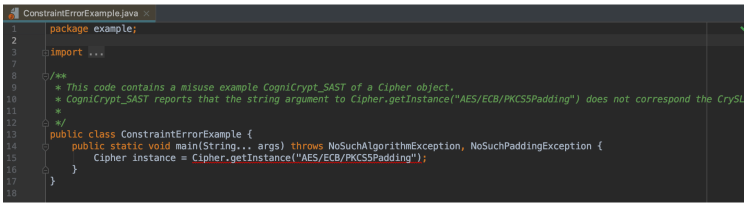


```java
import java.security.NoSuchAlgorithmException;

import javax.crypto.Cipher;
import javax.crypto.NoSuchPaddingException;

/**
 * This code contains a misuse example CogniCrypt_SAST of a Cipher object.
 * CogniCrypt_SAST reports that the string argument to Cipher.getInstance("AES/ECB/PK
 *
 */
public class ConstraintErrorExample {
    public static void main(String... args) throws NoSuchAlgorithmException, NoSuchPa
        Cipher instance = Cipher.getInstance("AES/ECB/PKCS5Padding");
    }
}
```

```
○ ConstraintErrorExample.java:
     15:21     CogniCrypt       error       ConstraintError violating CrySL rule for Cipher. First
```

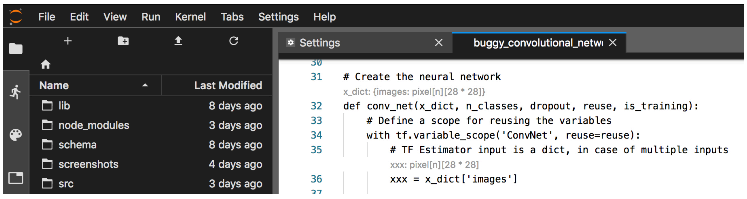
cognicrypt, Line 14, Column 39   Tab Size: 4   Java

Figure 8 Insecure crypto warning in Sublime Text.



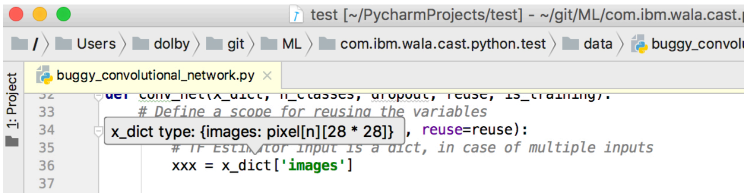
```java
package example;

import ...

/**
 * This code contains a misuse example CogniCrypt_SAST of a Cipher object.
 * CogniCrypt_SAST reports that the string argument to Cipher.getInstance("AES/ECB/PKCS5Padding") does not correspond the CrySL
 *
 */
public class ConstraintErrorExample {
    public static void main(String... args) throws NoSuchAlgorithmException, NoSuchPaddingException {
        Cipher instance = Cipher.getInstance("AES/ECB/PKCS5Padding");
    }
}
```

Figure 9 Insecure crypto warning in IntelliJ.



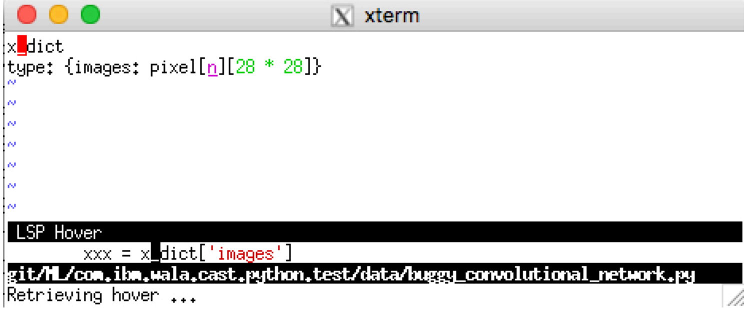
```
# Create the neural network
x_dict: {images: pixel[n][28 * 28]}
def conv_net(x_dict, n_classes, dropout, reuse, is_training):
    # Define a scope for reusing the variables
    with tf.variable_scope('ConvNet', reuse=reuse):
        # TF Estimator input is a dict, in case of multiple inputs
        xxx: pixel[n][28 * 28]
        xxx = x_dict['images']
```

Figure 10 Code lenes showing tensor types in JupyterLab.


test [~/PycharmProjects/test] - ~/git/ML/com.ibm.wala.cast.
/ Users dolby git ML com.ibm.wala.cast.python.test data buggy_convol
buggy_convolutional_network.py
1: Project

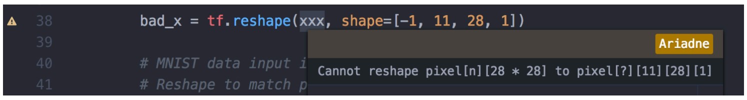
```
    # Define a scope for reusing the variables
  x_dict type: {images: pixel[n][28 * 28]}  , reuse=reuse):
        # TF Estimator input is a dict, in case of multiple inputs
        xxx = x_dict['images']
```

Figure 11 Hover tip showing tensor types in PyCharm.



```
x_dict
type: {images: pixel[n][28 * 28]}
~
~
~
~
~
~
~
 LSP Hover
          xxx = x_dict['images']
git/ML/com.ibm.wala.cast.python.test/data/buggy_convolutional_network.py
Retrieving hover ...
```

Figure 12 Hover tip showing tensor types in Vim

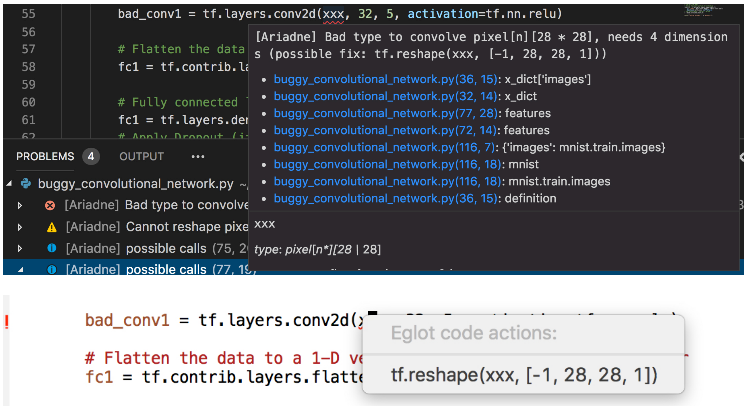


```
        bad_x = tf.reshape(xxx, shape=[-1, 11, 28, 1])

        # MNIST data input i
        # Reshape to match p
```

Ariadne

Cannot reshape pixel[n][28 ∗ 28] to pixel[?][11][28][1]

Figure 13 Diagnostic warning showing an incompatible `reshape` in Atom.



```
        bad_conv1 = tf.layers.conv2d(xxx, 32, 5, activation=tf.nn.relu)

        # Flatten the data
        fc1 = tf.contrib.la

        # Fully connected
        fc1 = tf.layers.de
```

buggy_convolutional_network.py ~/
[Ariadne] Bad type to convolve
[Ariadne] Cannot reshape pixe
[Ariadne] possible calls (75, 2
[Ariadne] possible calls (77, 19

[Ariadne] Bad type to convolve pixel[n][28 ∗ 28], needs 4 dimensions (possible fix: tf.reshape(xxx, [-1, 28, 28, 1]))

- buggy_convolutional_network.py(36, 15): x_dict['images']
- buggy_convolutional_network.py(32, 14): x_dict
- buggy_convolutional_network.py(77, 28): features
- buggy_convolutional_network.py(72, 14): features
- buggy_convolutional_network.py(116, 7): {'images': mnist.train.images}
- buggy_convolutional_network.py(116, 18): mnist
- buggy_convolutional_network.py(116, 18): mnist.train.images
- buggy_convolutional_network.py(36, 15): definition

xxx

type: pixel[n][28 | 28]*

```
!       bad_conv1 = tf.layers.conv2d(x

        # Flatten the data to a 1-D ve
        fc1 = tf.contrib.layers.flatte
```

Eglot code actions:

tf.reshape(xxx, [-1, 28, 28, 1])

Figure 14 Diagnostic error showing fixable incorrect dimensions for `conv2d`. Error shown in Visual Studio Code and quick fix in Emacs.

5 Comparison Between MagpieBridge-Based Approach and Plugin-Based Approach

While MagpieBridge enables analyses to run in a larger set of IDEs, the question remains of how the support in any specific IDE using MagpieBridge compares to a custom-built plugin for that same IDE. Because most analysis tools do not have integration with most IDEs, we are going to focus our comparison on one existing combination: the CogniCrypt plugin for Eclipse. Afterwards, we discuss in more general terms the range of functionality exploited by custom plugins that is supported by LSP.

5.1 Comparison Between MagpieBridge-Based CogniCrypt and CogniCrypt Eclipse Plugin

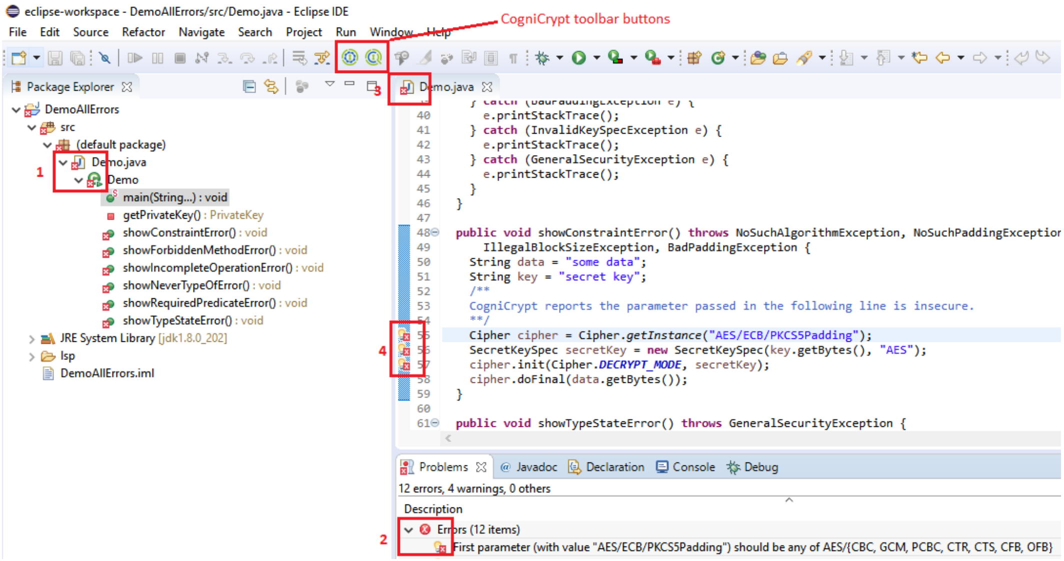
The CogniCrypt Eclipse Plugin [48] consists of two components: code generation, which generates secure implementations for user-defined cryptographic programming tasks, and cryptographic misuse detection, which runs static code analysis in the background and reports insecure usage of cryptographic APIs. MagpieBridge focuses on analysis, and so we do not consider the code-generation component here. For comparison, we integrated the static crypto analysis of CogniCrypt with MagpieBridge into Eclipse IDE.

Figure 15 and Figure 16 are screenshots in which the original CogniCrypt Eclipse Plugin reports insecure crypto warnings. In comparison, Figure 17 shows our CogniCrypt-integration with MagpieBridge. Figure 15 shows two buttons that CogniCrypt adds to the toolbar: “Generate Code For Cryptographic Task” and “Apply CogniCrypt Misuse to Selected Project”. By clicking the latter, one triggers the misuse detection using the plugin in its default configuration. The plugin can also be configured to trigger the analysis whenever a Java file is saved. On the other hand, MagpieBridge-based CogniCrypt starts the analysis automatically whenever a Java file is opened or saved. In either case, after the analysis has been run, any detected misuses are indicated in Eclipse in several ways, which the corresponding numbers show in Figure 15 and Figure 17:

1. In the Package Explorer view, the error ticks appear on the affected Java element and their parent elements.
2. In the Problems view, the detected misuses are listed as errors.
3. The editor tab is annotated with an error marker.
4. In the editor’s vertical ruler / gutter, an error marker is displayed near the affected line.

As shown in Figure 16, one can hover over an error marker next to the affected line to view the description of the misuse. The appearance of the MagpieBridge-based and plugin-based CogniCrypt is rather similar, with just a few differences:

- MagpieBridge-based CogniCrypt does not change the appearance of the IDE. To work with the MagpieServer which runs the crypto analysis, end-users do not have to do anything different. The analysis runs automatically whenever a Java file is opened or saved by an end-user. In contrast, in the Eclipse Plugin, one can trigger the analysis manually, or (optionally) have it started automatically whenever a file is saved.
- Results are indicated similarly in the CogniCrypt Eclipse Plugin MagpieBridge-based CogniCrypt; however, in MagpieBridge-based CogniCrypt in addition to the error markers, squiggly lines appear under the affected lines.
- In MagpieBridge-based CogniCrypt, the hover message also includes a quick fix that can replace the insecure parameter `AES/ECB/PKCS5Padding` with a secure parameter `ASE/CBC/PKCS5Padding` automatically. Since MagpieBridge preserves the precise source

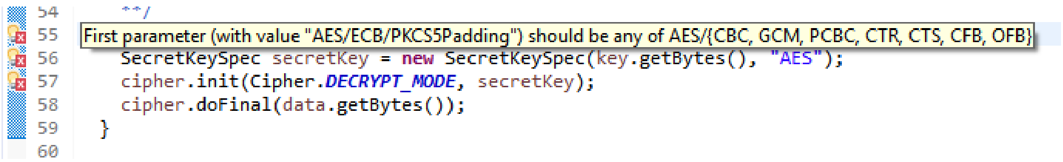


```
DemoAllErrors
  src
    (default package)
      Demo.java
        Demo
          main(String...) : void
          getPrivateKey() : PrivateKey
          showConstraintError() : void
          showForbiddenMethodError() : void
          showIncompleteOperationError() : void
          showNeverTypeOfError() : void
          showRequiredPredicateError() : void
          showTypeStateError() : void
  JRE System Library [jdk1.8.0_202]
  lsp
  DemoAllErrors.iml
```

```java
    } catch (BadPaddingException e) {
      e.printStackTrace();
    } catch (InvalidKeySpecException e) {
      e.printStackTrace();
    } catch (GeneralSecurityException e) {
      e.printStackTrace();
    }
  }

  public void showConstraintError() throws NoSuchAlgorithmException, NoSuchPaddingExceptior
      IllegalBlockSizeException, BadPaddingException {
    String data = "some data";
    String key = "secret key";
    /**
    CogniCrypt reports the parameter passed in the following line is insecure.
    **/
    Cipher cipher = Cipher.getInstance("AES/ECB/PKCS5Padding");
    SecretKeySpec secretKey = new SecretKeySpec(key.getBytes(), "AES");
    cipher.init(Cipher.DECRYPT_MODE, secretKey);
    cipher.doFinal(data.getBytes());
  }

  public void showTypeStateError() throws GeneralSecurityException {
```

```
12 errors, 4 warnings, 0 others
Description
Errors (12 items)
    First parameter (with value "AES/ECB/PKCS5Padding") should be any of AES/{CBC, GCM, PCBC, CTR, CTS, CFB, OFB}
```

Figure 15 The appearance of CogniCrypt Eclipse Plugin.



```java
    **/
First parameter (with value "AES/ECB/PKCS5Padding") should be any of AES/{CBC, GCM, PCBC, CTR, CTS, CFB, OFB}
    SecretKeySpec secretKey = new SecretKeySpec(key.getBytes(), "AES");
    cipher.init(Cipher.DECRYPT_MODE, secretKey);
    cipher.doFinal(data.getBytes());
  }
```

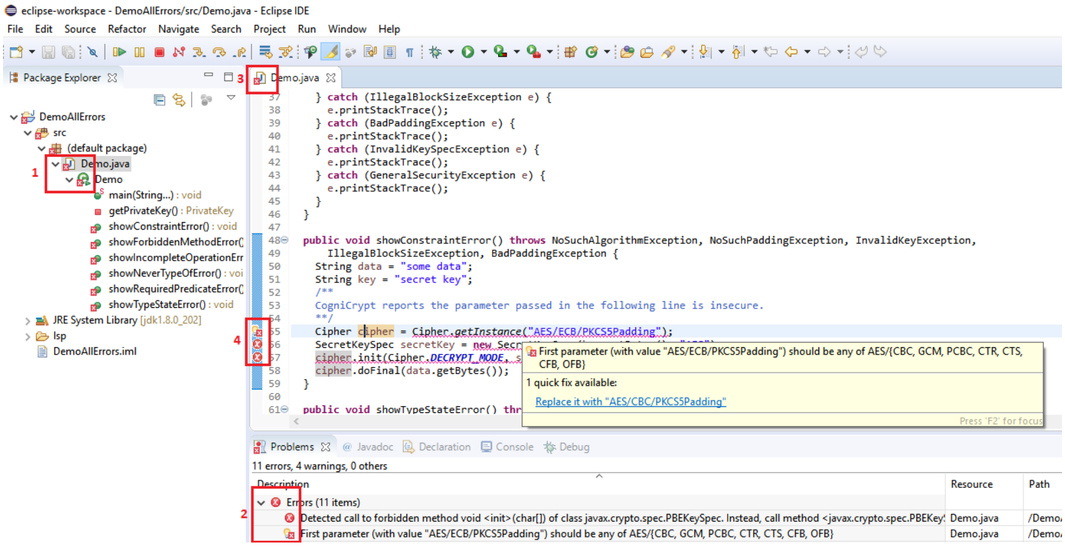
Figure 16 CogniCrypt Eclipse Plugin: insecure crypto warning message shown by hovering.

code position from the WALA source-code front end, e.g., the exact code range (starting/ending line/column numbers) of each parameter of a method call, we were able to build such quick fix easily with the `codeAction` feature supported by LSP. Such quick fix is not available in the CogniCrypt Eclipse Plugin, although the warning message already indicates what a secure parameter should look like.

Another difference is that, since MagpieBridge does not add buttons to the IDE, it needs to invoke the analysis automatically. When the end-user changes the opened file, the MagpieServer clears the warnings when it receives the `didChange` notification from the IDE. The analysis is then restarted whenever the end-user saves the file, i.e., the MagpieServer receives a `didSave` notification. Once the MagpieServer receives the notification from the Eclipse IDE, it resolves the source code and library code path required for the inter-procedural crypto analysis. This analysis is all asynchronous, so that the analysis always runs in the background and updated error messages are shown once they are available. If they want to, end-users have the ability to connect and disconnect the MagpieServer at runtime, e.g., via "Preferences" in Eclipse IDE.

5.2 Comparison to Other Plugin-Based Approaches

As shown in Figure 18, LSP offers a set of UI features to present the analysis results to end-users that are sufficient to capture the majority of UI features used in a range of existing plugins for a single analysis tool in a specific IDE. Most of the plugin approaches we identified were implemented as Eclipse plugins (Cheetah [37], SpotBugs [23] and ASIDE [63]), but


eclipse-workspace - DemoAllErrors/src/Demo.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help
Package Explorer
Demo.java
DemoAllErrors
src
(default package)
Demo.java
Demo
main(String...) : void
getPrivateKey() : PrivateKey
showConstraintError() : void
showForbiddenMethodError(
showIncompleteOperationErr
showNeverTypeOfError() : voi
showRequiredPredicateError(
showTypeStateError() : void
JRE System Library [jdk1.8.0_202]
lsp
DemoAllErrors.iml
1
2
3
4
Problems Javadoc Declaration Console Debug
Press 'F2' for focus

```
    } catch (IllegalBlockSizeException e) {
      e.printStackTrace();
    } catch (BadPaddingException e) {
      e.printStackTrace();
    } catch (InvalidKeySpecException e) {
      e.printStackTrace();
    } catch (GeneralSecurityException e) {
      e.printStackTrace();
    }
  }

  public void showConstraintError() throws NoSuchAlgorithmException, NoSuchPaddingException, InvalidKeyException,
      IllegalBlockSizeException, BadPaddingException {
    String data = "some data";
    String key = "secret key";
    /**
    CogniCrypt reports the parameter passed in the following line is insecure.
    **/
    Cipher cipher = Cipher.getInstance("AES/ECB/PKCS5Padding");
    SecretKeySpec secretKey = new Secr
    cipher.init(Cipher.DECRYPT_MODE, s
    cipher.doFinal(data.getBytes());
  }

  public void showTypeStateError() thr
```

First parameter (with value "AES/ECB/PKCS5Padding") should be any of AES/{CBC, GCM, PCBC, CTR, CTS, CFB, OFB}

1 quick fix available:

Replace it with "AES/CBC/PKCS5Padding"

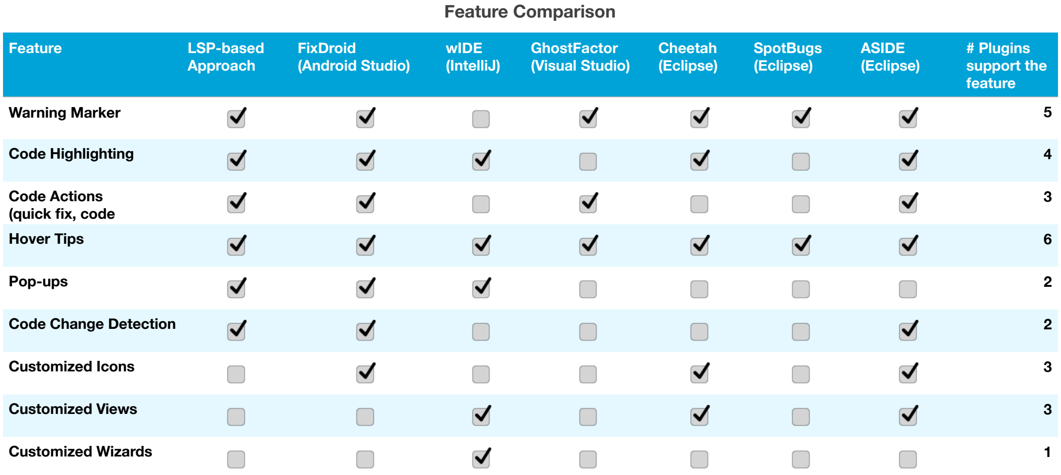
11 errors, 4 warnings, 0 others

Description	Resource	Path
Errors (11 items)		
Detected call to forbidden method void <init>(char[]) of class javax.crypto.spec.PBEKeySpec. Instead, call method <javax.crypto.spec.PBEKey:	Demo.java	/Demo.
First parameter (with value "AES/ECB/PKCS5Padding") should be any of AES/{CBC, GCM, PCBC, CTR, CTS, CFB, OFB}	Demo.java	/Demo.

Figure 17 The appearance of MagpieBridge-based CogniCrypt: insecure crypto warning message and quick fix shown by hovering.

some of them were created for other popular IDEs such as Android Studio (FixDroid [52]), IntelliJ (wIDE [51]) and Visual Studio (GhostFactor [42]). Figure 18 shows the comparison between features that can be supported with LSP to features supported by these existing plugin approaches.

Some plugins do use IDE features that are not explicitly supported by LSP; however, there are often analogs in LSP that could be used instead. For instance, Cheetah uses a custom view, essentially a separate window panel in the IDE, to show an example data-flow trace for a bug; in LSP, related information capturing a trace can be attached to problems as illustrated in Figure 14. Other uses of custom views and wizards are mainly for analysis configuration. Simple forms of such analysis configuration could be supported by the message protocol in LSP.


Feature Comparison

Feature	LSP-based Approach	FixDroid (Android Studio)	wIDE (IntelliJ)	GhostFactor (Visual Studio)	Cheetah (Eclipse)	SpotBugs (Eclipse)	ASIDE (Eclipse)	# Plugins support the feature
Warning Marker	☑	☑	□	☑	☑	☑	☑	5
Code Highlighting	☑	☑	☑	□	☑	□	☑	4
Code Actions (quick fix, code	☑	☑	□	☑	□	□	☑	3
Hover Tips	☑	☑	☑	☑	☑	☑	☑	6
Pop-ups	☑	☑	☑	□	□	□	□	2
Code Change Detection	☑	☑	□	□	□	□	☑	2
Customized Icons	□	☑	□	□	☑	□	☑	3
Customized Views	□	□	☑	□	☑	□	☑	3
Customized Wizards	□	□	☑	□	□	□	□	1

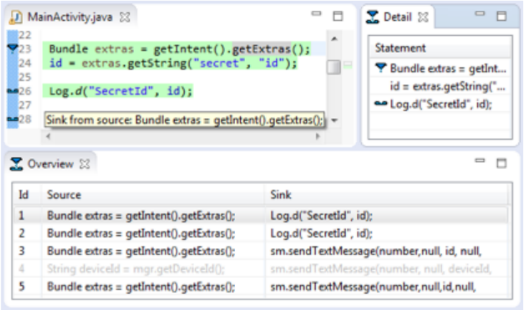
Figure 18 Feature comparison between LSP-based approach and other plugin-based approaches.

One minor feature unsupported by LSP appeared in the plugins: customized icons (see Figure 19, Figure 20 and Figure 21) are not supported by the LSP-based approach, since that requires changes to the appearance of the IDEs, which LSP intends not to. Although studies have shown customized icons are useful to catch end-users' attention [52, 54, 63], it is not clear if it is more effective than the default error icon supported by each editor.

As we can see in Figure 18, the major features such as hover tips, warning marker and code highlighting, which are supported by a majority of the plugins, can be supported by an LSP-based approach. However, LSP support varies across IDEs, both in what features are handled and how they are shown. In LSP, hover tips are specified as the `hover` request sent from the client to the server, warning marker can be realized by the `publishDiagnostics` notification and `documentHighlight` is the corresponding request for code highlighting. However, the implementation of `documentHighlight` varies from editor to editor, since the specification for this feature in LSP is unclear. Most plugins listed in Figure 18 support code highlighting. This features means changing the background color of affected lines of code as shown in Figure 19, Figure 20 and Figure 21. While Visual Studio Code limits this feature to only highlights all references to a symbol scoped in a file, sublime Text choses an underline for highlighting (see Figure 23). In addition, there is no possibility with LSP to specify the background color used in this feature, all editors have their pre-defined colors.

Some advanced features such as code actions (we have shown quick fix with MagpieBridge-based CogniCrypt), pop-ups and code change detections can also be supported by LSP. There are two interfaces (`showMessage` and `showMessageRequest`) defined in LSP which are implemented as pop-up windows in editors. Figure 24 shows a message sent from a server to the Eclipse IDE that is displayed in a pop-up window. Where more interactions are required, the interface `showMessageRequest` allows to pass actions and wait for an answer from the client. Figure 25 shows a pop-up windows with a message and available actions in Visual Studio Code.

Features that are not supported by LSP for now can be extended to LSP in the future, since LSP is a moving target with ever-growing functionality and support. One just has to keep in mind that, as the LSP is extended, the IDEs/editors that support it, might require extensions as well.



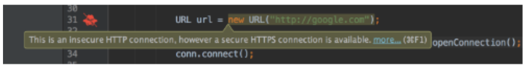
```
Bundle extras = getIntent().getExtras();
id = extras.getString("secret", "id");

Log.d("SecretId", id);
```

Sink from source: Bundle extras = getIntent().getExtras();

Statement
Bundle extras = getInt...
id = extras.getString("...
Log.d("SecretId", id);

Id	Source	Sink
1	Bundle extras = getIntent().getExtras();	Log.d("SecretId", id);
2	Bundle extras = getIntent().getExtras();	Log.d("SecretId", id);
3	Bundle extras = getIntent().getExtras();	sm.sendTextMessage(number,null, id, null,
4	String deviceId = mgr.getDeviceId();	sm.sendTextMessage(number, null, deviceId,
5	Bundle extras = getIntent().getExtras();	sm.sendTextMessage(number,null,id,null,

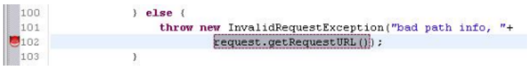
Figure 19 Cheetah: code highlighting, hover tips, customized icon and views.



```
URL url = new URL("http://google.com");
openConnection();
conn.connect();
```

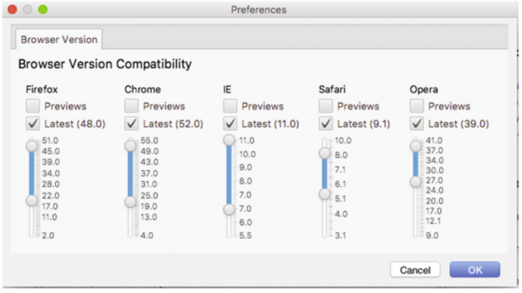
This is an insecure HTTP connection, however a secure HTTPS connection is available. more... (⌘F1)

Figure 20 FixDroid: code highlighting, hover tips and customized icon.



```
} else {
    throw new InvalidRequestException("bad path info, "+
            request.getRequestURL());
}
```

Figure 21 ASIDE: code highlighting and customized icon.



Preferences

Browser Version

Browser Version Compatibility

Firefox
☐ Previews
☑ Latest (48.0)

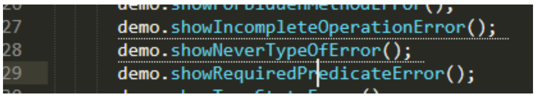
Chrome
☐ Previews
☑ Latest (52.0)

IE
☐ Previews
☑ Latest (11.0)

Safari
☐ Previews
☑ Latest (9.1)

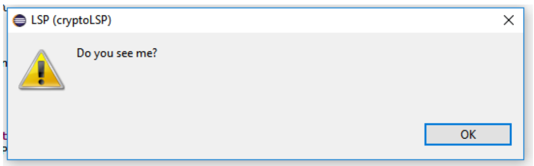
Opera
☐ Previews
☑ Latest (39.0)

Figure 22 wIDE: customized wizard.



```
demo.showIncompleteOperationError();
demo.showNeverTypeOfError();
demo.showRequiredPredicateError();
```

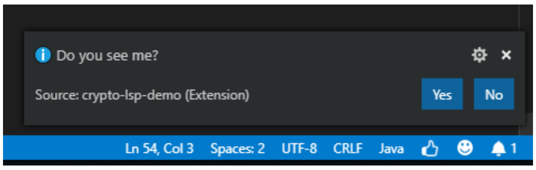
Figure 23 Highlighting in Sublime Text.



LSP (cryptoLSP)

Do you see me?

Figure 24 Pop-up in Eclipse.



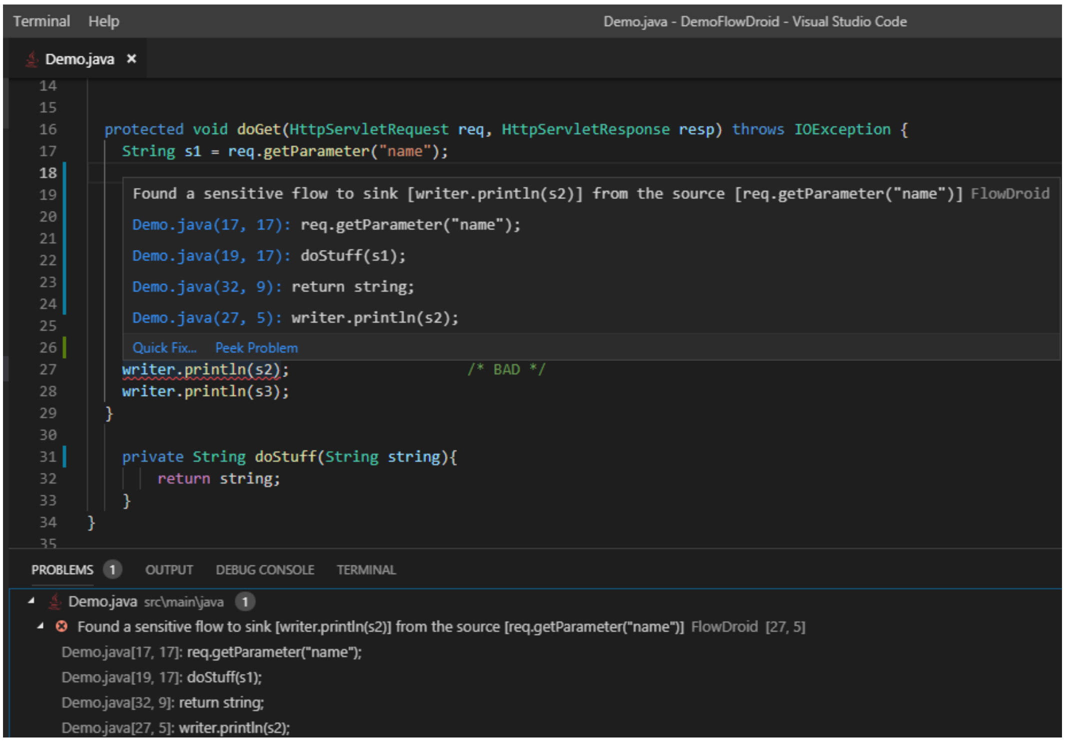
Do you see me?

Source: crypto-lsp-demo (Extension)

Figure 25 Pop-up with actions in Visual Studio Code.

6 Conclusion and Future Work

The difficulty of integrating static tools into different IDEs and editors has caused little adoption of the tools by developers and researchers, and MagpieBridge addresses this problem by providing a general approach to integrating static analyses into IDEs and editors. MagpieBridge uses the increasingly popular Language Server Protocol and supports from rich analysis frameworks, WALA and Soot. We have shown MagpieBridge supporting CogniCrypt, but this is just the beginning; we conclude and presage future work by showing what is, to the best of our knowledge, the first ever IDE integration of the well-known FlowDroid security analysis. Figure 26 shows FlowDroid analyzing the data flow starting from a parameter of the HTTP request, finding a cross-site scripting vulnerability which can be exploited by attackers, and showing a witness trace of it. The expressions in the witness are shown precisely, which is possible since the IRConverter of MagpieBridge is able to run FlowDroid unchanged on the converted IR and recover precise source mappings. As far as we know, this has never been done before with FlowDroid. MagpieBridge then renders this precise trace from FlowDroid in the IDE, also the first time this has been done. While FlowDroid is one of the best-known security analyses, this is just one example of what more can be done with MagpieBridge, and our future work includes handling many more analyses.



```
14
15
16    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws IOException {
17      String s1 = req.getParameter("name");
18
19
20
21
22
23
24
25
26
27      writer.println(s2);                    /* BAD */
28      writer.println(s3);
29    }
30
31      private String doStuff(String string){
32          return string;
33      }
34  }
35
```

```
Found a sensitive flow to sink [writer.println(s2)] from the source [req.getParameter("name")] FlowDroid
Demo.java(17, 17): req.getParameter("name");
Demo.java(19, 17): doStuff(s1);
Demo.java(32, 9): return string;
Demo.java(27, 5): writer.println(s2);
```

```
Found a sensitive flow to sink [writer.println(s2)] from the source [req.getParameter("name")] FlowDroid [27, 5]
  Demo.java[17, 17]: req.getParameter("name");
  Demo.java[19, 17]: doStuff(s1);
  Demo.java[32, 9]: return string;
  Demo.java[27, 5]: writer.println(s2);
```

Figure 26 A sensitive data flow reported by FlowDroid in Visual Studio Code.

References

1 Android Studio. https://developer.android.com/studio. Accessed: 2019-01-10.

2 AppScan. https://www.ibm.com/security/application-security/appscan. Accessed: 2019-01-10.

3 Atom. https://atom.io/. Accessed: 2019-01-10.

4 Clang Static Analyzer. https://clang-analyzer.llvm.org/. Accessed: 2019-01-10.

5 CodeSonar. https://www.grammatech.com/products/codesonar. Accessed: 2019-01-10.

6 Cppcheck. http://cppcheck.sourceforge.net/. Accessed: 2019-01-10.

7 Doop. http://doop.program-analysis.org/. Accessed: 2019-01-10.

8 Eclipse. https://www.eclipse.org/. Accessed: 2019-01-10.

9 Eclipse LSP4J. https://projects.eclipse.org/proposals/eclipse-lsp4j. Accessed: 2019-01-10.

10 Emacs. https://www.gnu.org/software/emacs/. Accessed: 2019-01-10.

11 Facebook Infer. https://fbinfer.com/. Accessed: 2019-01-10.

12 IBM WebSphere. https://www.ibm.com/cloud/websphere-application-platform. Accessed: 2019-01-10.

13 IntelliJ. https://www.jetbrains.com/idea/. Accessed: 2019-01-10.

14 JSON-RPC. https://www.jsonrpc.org/. Accessed: 2019-01-10.

15 Language Server Protocol. https://microsoft.github.io/language-server-protocol/. Accessed: 2019-01-10.

16 Monaco. https://microsoft.github.io/monaco-editor/index.html. Accessed: 2019-01-10.

17 PMD. https://pmd.github.io/. Accessed: 2019-01-10.

18 PyCharm. https://www.jetbrains.com/pycharm/. Accessed: 2019-01-10.

19 Safe. https://github.com/sukyoung/safe. Accessed: 2019-01-10.

20 SARIF Specification. https://github.com/oasis-tcs/sarif-spec. Accessed: 2019-01-10.

21 Soot. https://github.com/Sable/soot. Accessed: 2019-01-10.

22 Souffle. https://github.com/oracle/souffle/wiki. Accessed: 2019-01-10.

23 SpotBugs. https://spotbugs.github.io/. Accessed: 2019-01-10.

24 Spyder. https://www.spyder-ide.org/. Accessed: 2019-01-10.

25 Static Analysis Results: A Format and a Protocol: SARIF and SASP. http://blogs.grammatech.com/static-analysis-results-a-format-and-a-protocol-sarif-sasp. Accessed: 2019-01-10.

26 Sublime. https://www.sublimetext.com/. Accessed: 2019-01-10.

27 Vim. https://www.vim.org/. Accessed: 2019-01-10.

28 Visual Studio Code. https://code.visualstudio.com/. Accessed: 2019-01-10.

29 WALA. https://github.com/wala/WALA. Accessed: 2019-01-10.

30 Xanitizer. https://www.rigs-it.com/xanitizer/. Accessed: 2019-01-10.

31 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. doi:10.1145/2594291.2594299.

32 Eric Bodden, Társis Tolêdo, Márcio Ribeiro, Claus Brabrand, Paulo Borba, and Mira Mezini. SPLLIFT: statically analyzing software product lines in minutes instead of years. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation (PLDI)*, pages 355–364, 2013. URL: http://www.bodden.de/pubs/bmb+13spllift.pdf.

33 Martin Bravenboer and Yannis Smaragdakis. Exception analysis and points-to analysis: better together. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis, ISSTA 2009, Chicago, IL, USA, July 19-23, 2009*, pages 1–12, 2009. doi:10.1145/1572272.1572274.

34 Hongyi Chen, Ho-fung Leung, Biao Han, and Jinshu Su. Automatic privacy leakage detection for massive android apps via a novel hybrid approach. In *IEEE International Conference on Communications, ICC 2017, Paris, France, May 21-25, 2017*, pages 1–7, 2017. doi:10.1109/ICC.2017.7996335.

35 Maria Christakis and Christian Bird. What developers want and need from program analysis: an empirical study. In *ASE*, pages 332–343, 2016.

36 Lisa Nguyen Quang Do, Karim Ali, Benjamin Livshits, Eric Bodden, Justin Smith, and Emerson Murphy-Hill. Just-in-time Static Analysis. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2017, pages 307–317, New York, NY, USA, 2017. ACM. doi:10.1145/3092703.3092705.

37 Lisa Nguyen Quang Do, Karim Ali, Benjamin Livshits, Eric Bodden, Justin Smith, and Emerson R. Murphy-Hill. Cheetah: just-in-time taint analysis for Android apps. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017 - Companion Volume*, pages 39–42, 2017. doi:10.1109/ICSE-C.2017.20.

38 Julian Dolby, Avraham Shinnar, Allison Allain, and Jenna Reinen. Ariadne: Analysis for Machine Learning Programs. In *Proceedings of the 2Nd ACM SIGPLAN International*

Workshop on Machine Learning and Programming Languages, MAPL 2018, pages 1–10, New York, NY, USA, 2018. ACM. doi:10.1145/3211346.3211349.

39 Moritz Eysholdt and Heiko Behrens. Xtext: implement your language faster than the quick and dirty way. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pages 307–309. ACM, 2010.

40 Stephen Fink and Julian Dolby. WALA–The TJ Watson Libraries for Analysis, 2012.

41 Stephen Fink, Julian Dolby, and L Colby. Semi-automatic J2EE transaction configuration, January 2019.

42 Xi Ge and Emerson R. Murphy-Hill. Manual refactoring changes with automated refactoring validation. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, pages 1095–1105, 2014. doi:10.1145/2568225.2568280.

43 Dennis Giffhorn and Gregor Snelting. A new algorithm for low-deterministic security. *International Journal of Information Security*, 14(3):263–287, June 2015. doi:10.1007/s10207-014-0257-6.

44 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 *NDSS*, volume 15, page 110, 2015.

45 Christian Hammer and Gregor Snelting. Flow-Sensitive, Context-Sensitive, and Object-sensitive Information Flow Control Based on Program Dependence Graphs. *International Journal of Information Security*, 8(6):399–422, December 2009. doi:10.1007/s10207-009-0086-1.

46 David Hovemeyer and William Pugh. Finding More Null Pointer Bugs, but Not Too Many. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '07, pages 9–14, New York, NY, USA, 2007. ACM. doi:10.1145/1251535.1251537.

47 Brittany Johnson, Yoonki Song, Emerson R. Murphy-Hill, and Robert W. Bowdidge. Why don't software developers use static analysis tools to find bugs? In *ICSE*, pages 672–681, 2013.

48 Stefan Krüger, Sarah Nadi, Michael Reif, Karim Ali, Mira Mezini, Eric Bodden, Florian Göpfert, Felix Günther, Christian Weinert, Daniel Demmler, et al. CogniCrypt: supporting developers in using cryptography. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 931–936. IEEE Press, 2017.

49 Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. The Soot framework for Java program analysis: a retrospective. In *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, volume 15, page 35, 2011.

50 Li Li, Alexandre Bartel, Tegawendé F. Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick D. McDaniel. IccTA: Detecting Inter-Component Privacy Leaks in Android Apps. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, pages 280–291, 2015. doi:10.1109/ICSE.2015.48.

51 Alfonso Murolo, Fabian Stutz, Maria Husmann, and Moira C. Norrie. Improved Developer Support for the Detection of Cross-Browser Incompatibilities. In *Web Engineering - 17th International Conference, ICWE 2017, Rome, Italy, June 5-8, 2017, Proceedings*, pages 264–281, 2017. doi:10.1007/978-3-319-60131-1_15.

52 Duc-Cuong Nguyen, Dominik Wermke, Yasemin Acar, Michael Backes, Charles Weir, and Sascha Fahl. A Stitch in Time: Supporting Android Developers in Writing Secure Code. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 1065–1077, 2017. doi:10.1145/3133956.3133977.

53 Damien Octeau, Patrick D. McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. Effective Inter-Component Communication Mapping in Android: An Essential Step Towards Holistic Security Analysis. In *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16,*

2013, pages 543–558, 2013. URL: https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/octeau.

54 S. E. Schechter, R. Dhamija, A. Ozment, and I. Fischer. The Emperor's New Security Indicators. In *2007 IEEE Symposium on Security and Privacy (SP '07)*, pages 51–65, May 2007. doi:10.1109/SP.2007.35.

55 Johannes Späth, Karim Ali, and Eric Bodden. Context-, Flow-, and Field-sensitive Data-flow Analysis Using Synchronized Pushdown Systems. *Proc. ACM Program. Lang.*, 3(POPL):48:1–48:29, January 2019. doi:10.1145/3290361.

56 Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. FeatureIDE: An extensible framework for feature-oriented software development. *Science of Computer Programming*, 79:70–85, 2014.

57 Emina Torlak and Satish Chandra. Effective Interprocedural Resource Leak Detection. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 535–544, New York, NY, USA, 2010. ACM. doi:10.1145/1806799.1806876.

58 Omer Tripp, Marco Pistoia, Patrick Cousot, Radhia Cousot, and Salvatore Guarnieri. Andromeda: Accurate and Scalable Security Analysis of Web Applications. In *Fundamental Approaches to Software Engineering - 16th International Conference, FASE 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, pages 210–225, 2013. doi:10.1007/978-3-642-37057-1_15.

59 Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. TAJ: Effective Taint Analysis of Web Applications. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 87–97, New York, NY, USA, 2009. ACM. doi:10.1145/1542476.1542486.

60 Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot: A Java bytecode optimization framework. In *CASCON First Decade High Impact Papers*, pages 214–224. IBM Corp., 2010.

61 Christos V. Vrachas. Integration of static analysis results with ProGuard optimizer for Android applications. *Bachelor Thesis*, 2017.

62 Fengguo Wei, Sankardas Roy, Xinming Ou, and Robby. Amandroid: A Precise and General Inter-component Data Flow Analysis Framework for Security Vetting of Android Apps. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, pages 1329–1341, 2014. doi:10.1145/2660267.2660357.

63 Jing Xie, Bill Chu, Heather Richter Lipford, and John T. Melton. ASIDE: IDE support for web application security. In *Twenty-Seventh Annual Computer Security Applications Conference, ACSAC 2011, Orlando, FL, USA, 5-9 December 2011*, pages 267–276, 2011. doi:10.1145/2076732.2076770.

Semantic Patches for Java Program Transformation

Hong Jin Kang
School of Information Systems, Singapore Management University, Singapore

Ferdian Thung
School of Information Systems, Singapore Management University, Singapore

Julia Lawall
Sorbonne Université/Inria/LIP6, France

Gilles Muller
Sorbonne Université/Inria/LIP6, France

Lingxiao Jiang
School of Information Systems, Singapore Management University, Singapore

David Lo
School of Information Systems, Singapore Management University, Singapore

Abstract

Developing software often requires code changes that are widespread and applied to multiple locations. There are tools for Java that allow developers to specify patterns for program matching and source-to-source transformation. However, to our knowledge, none allows for transforming code based on its control-flow context. We prototype Coccinelle4J, an extension to Coccinelle, which is a program transformation tool designed for widespread changes in C code, in order to work on Java source code. We adapt Coccinelle to be able to apply scripts written in the Semantic Patch Language (SmPL), a language provided by Coccinelle, to Java source files. As a case study, we demonstrate the utility of Coccinelle4J with the task of API migration. We show 6 semantic patches to migrate from deprecated Android API methods on several open source Android projects. We describe how SmPL can be used to express several API migrations and justify several of our design decisions.

2012 ACM Subject Classification Software and its engineering → Software notations and tools

Keywords and phrases Program transformation, Java

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2019.22

Category Experience Report

Supplement Material ECOOP 2019 Artifact Evaluation approved artifact available at
`https://dx.doi.org/10.4230/DARTS.5.2.10`
Coccinelle4J can be found at `https://github.com/kanghj/coccinelle/tree/java`

Funding This research was supported by the Singapore National Research Foundation (award number: NRF2016-NRF-ANR003) and the ANR ITrans project.

1 Introduction

Over ten years ago, Coccinelle was introduced to the systems and Linux kernel developer communities as a tool for automating large-scale changes in C software [23]. Coccinelle particularly targeted so-called *collateral evolutions*, in which a change to a library interface triggers the need for changes in all of the clients of that interface. A goal in the development of Coccinelle was that it should be able to be used directly by Linux kernel developers, based on their existing experience with the source code. Accordingly, Coccinelle provides a language, the Semantic Patch Language (SmPL), for expressing transformations using a

33rd European Conference on Object-Oriented Programming (ECOOP 2019).
Editor: Alastair F. Donaldson; Article No. 22; pp. 22:1–22:27
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

generalization of the familiar patch syntax. Like a traditional patch, a Coccinelle semantic patch consists of fragments of C source code, in which lines to remove are annotated with `-` and lines to add are annotated with `+`. Connections between code fragments that should be executed within the same control-flow path are expressed using "`...`" and arbitrary subterms are expressed using metavariables, raising the level of abstraction so as to allow a single semantic patch to update complex library usages across a code base. This user-friendly transformation specification notation, which does not require users to know about typical program manipulation internals such as abstract-syntax trees and control-flow graphs, has led to Coccinelle's wide adoption by Linux kernel developers, with over 6000 commits to the Linux kernel mentioning use of Coccinelle [14]. Coccinelle is also regularly used by developers of other C software, such as the Windows emulator Wine[1] and the Internet of Things operating system Zephyr.[2]

Software developers who learn about Coccinelle regularly ask whether such a tool exists for languages other than C.[3] To begin to explore the applicability of the Coccinelle approach to specifying transformations to software written in other languages, we have extended the implementation of Coccinelle to support matching and transformation of Java code. Java is currently the most popular programming language according to the TIOBE index.[4] The problem of library changes and library migration has also been documented for Java software [29]. While tools have been developed to automate transformations of Java code [20, 25, 28], none strikes the same balance of closeness to the source language and ease of reasoning about control flow, as provided by Coccinelle. Still, Java offers features that are not found in C, such as exceptions and subtyping. Thus, we believe that Java is an interesting target for understanding the generalizability of the Coccinelle approach, and that an extension of Coccinelle to Java can have a significant practical impact.

Our research goal is to explore what can be done for Java programs with the Coccinelle approach, i.e. transformation rules expressed in a patch-like notation and other features from Coccinelle. This experience paper documents our design decisions for Coccinelle4J, our extension of Coccinelle to handle Java code. We present the challenges we have encountered in the design of Coccinelle4J and the initial implementation. The design has been guided by a study of the changes found in the development history of five well-known open-source Java projects. Through a case study where we transform deprecated API call sites to use replacement API methods, we evaluate Coccinelle4J in terms of its expressiveness and its suitability for use on Java projects.

This paper makes the following contributions:

- We show that the approach of program transformation of Coccinelle, previously only used for C programs, generalizes to Java programs.
- We document the design decisions made to extend Coccinelle to work with Java.
- In the context of migrating APIs, we use Coccinelle4J and show that control-flow information is useful for Java program transformation.

The rest of this paper is organized as follows. Section 2 briefly introduces Coccinelle. Section 3 presents our extensions to support Java. Section 4 looks at a case study involving migrating APIs in open source Java projects. Section 5 discusses related work. Finally, we present our conclusions and discuss possible future directions of this work in Section 6.

[1] `https://wiki.winehq.org/Static_Analysis#Coccinelle`
[2] `https://docs.zephyrproject.org/latest/guides/coccinelle.html`
[3] `https://twitter.com/josh_triplett/status/994753065478582272`
[4] `https://www.tiobe.com/tiobe-index/`, visited January 2019

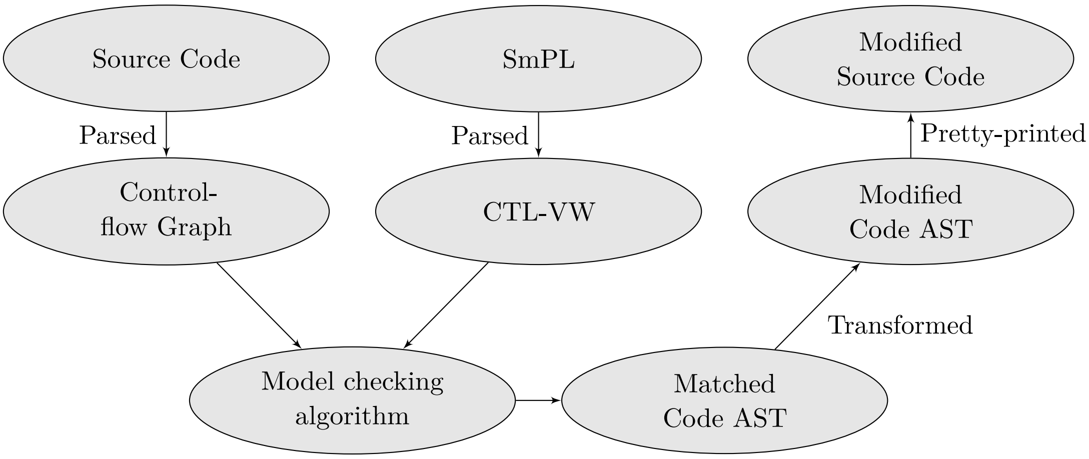


Figure 1 The process of transforming programs for a single rule.

2 Background

Coccinelle is a program transformation tool with the objective of specifying operating system collateral evolutions [19], code modifications required due to changes in interfaces of the operating system kernel or driver support libraries. Similar to widespread changes, such an evolution may involve code that is present in many locations within a software project. It was found that, to automate such changes, it is often necessary to take into account control-flow information, as collateral evolutions may involve error-handling code after invoking functions, or adding arguments based on context [3]. Much of the design of Coccinelle has been based on pragmatism, trading off correctness for ease of use and expressiveness. Coccinelle works on the control-flow graph and Abstract Syntax Tree (AST) for program matching and transformation, and therefore matches on program elements regardless of variations in formatting.

Coccinelle's engine is designed based on model checking. Building on an earlier work by Lacey and De Moor [12] on using Computational Tree Logic (CTL) to reason about compiler optimizations, Brunel et al. [3] propose CTL-VW as a foundation for program matching on control-flow graphs in Coccinelle. CTL-VW extends CTL, by adding predicates over metavariables that can be existentially qualified over program elements, as well as by adding witnesses, which record variable bindings and track locations to be transformed.

As input, Coccinelle accepts source code and a semantic patch describing the program transformation. The source code is parsed, producing an intraprocedural control-flow graph for each function, while the semantic patch language is converted into a CTL-VW formula. For each function definition or other top-level declaration in the source code, the CTL-VW formula is matched against the control-flow graph using a model-checking algorithm. This process is summarized in Figure 1.

Semantic patches are specified in SmPL, which has a declarative syntax resembling patches produced by the familiar `diff` command. A semantic patch describes code that is removed and added, with any relevant surrounding code given as context. '`-`' and '`+`' are used to mark lines containing code to be removed and added, respectively. This syntax follows a WYSIWYG approach and is familiar to developers using revision control software. Listing 1 shows a simplified version of a semantic patch checked into the mainline Linux kernel repository. This semantic patch removes declarations of variables that are assigned a constant value and immediately returned. It makes use of some notation unique to Coccinelle that we will explain further below.

Listing 1 Example of a rule that removes variables that only exist to return a constant.

```
1  @@
2  type T;
3  constant C;
4  identifier ret;
5  @@
6  - T ret = C;
7  ... when != ret
8      when strict
9  return
10 - ret
11 + C
12 ;
```

We briefly describe semantic patches and SmPL, using Listing 1 as an example. A rule in a semantic patch has two parts: the declaration of metavariables (lines 2-4), followed by the specification of the intended transformation over the declared metavariables (lines 6-12). Although our example contains only a single rule, a semantic patch can consist of multiple rules. Rules after the first one can refer to metavariables matched by prior rules.

Coccinelle offers abstractions to reason about paths. The "..." operator in SmPL represents an arbitrary sequence. In the context of Listing 1, the operator represents any sequence of statements over any control flow path. The `when` keyword restricts matches of "...". In Listing 1, the use of `when != ret` means that there should be no occurrences of `ret` in the matched control-flow path, and the use of `when strict` means that this property applies to all control-flow paths starting from the declaration, including those that abort with an error value. Without `when strict`, all paths must be matched *except* those that end with an error abort. A further option is `when exists`, which is satisfied whenever a single path meets the specified criteria on any other `when` constraints. By default, a control-flow path matched with "..." may not contain code that matches the patterns preceding or following the "...", here the declaration of `ret` and the return of its value, thus matching the shortest path between these points. This constraint can be lifted using `when any`. One caveat is that Coccinelle does not account for the run-time values when computing paths, and thus over-approximates the set of paths that can be taken.

As we do not modify the semantics of SmPL, we do not discuss SmPL in further detail. A more complete treatment of SmPL is given by Padioleau et al. [24].

3 Extending Coccinelle to Java

In this section, we document our experience and design decisions made during the development of Coccinelle4J. Our first observation is that much of the syntax of Java is also found in C. For example, both languages contain `if` statements, assignments, function/method calls, etc. And even some differences in the syntax, such as the ability to declare a variable in the header of a `for` loop, amount to only minor variations. Thus, we can use much of the implementation of Coccinelle unchanged and provide a transformation language that has a similar look and feel.

Still, Java is a different language than C, and notably offers an object-oriented programming paradigm, rather than a simple imperative one, as found in C. To identify the language features that are most relevant to widespread changes in Java code, we performed an analysis

Table 1 Number of commits for each type of change.

Type of change	Number of commits
Addition of new methods or modification of method signatures	43
Modification of multiple methods' implementation	33
Changes that are non-functional	23
Modification of imports	14
Changes related to sub-typing or inheritance	9
Modification of annotations	8

based on commits in the past year of five common Java libraries: Gson,[5] Apache Commons IO[6] and Commons Lang,[7] Mockito,[8] and Selenium.[9] Out of the 1179 commits to these projects, we found 130 containing a widespread change, i.e., a change that is both semantically and syntactically similar made in multiple locations. Depending on the widespread change present in the commit, we manually categorized these commits as shown in Table 1. We use the frequency of each category of change to motivate and prioritize the necessary features of Coccinelle that we port to Coccinelle4J. In particular, we notice that the features of class hierarchy and annotations are changed least frequently, so these features are not offered in the current design of Coccinelle4J.

Guided by this analysis, we have developed Coccinelle4J in three phases. As a first milestone for our work, we target the constructs found in Middleweight Java [1]. Middleweight Java is a minimal imperative fragment for Java, designed for the study of the Java type system. In contrast to other formal models of Java, Middleweight Java programs are valid executable Java programs, thus representing a useful first step for our work. Still, most real Java programs involve constructs that go beyond the very limited syntax of Middleweight Java. With ad-hoc testing on some real projects, we identified the need for handling control-flow with exceptions in the context of `try-catch`, and the need to introduce Java-specific *isomorphisms*, Coccinelle meta-rules that make it possible to match semantically equivalent code with concise patterns. These features were added in the second phase. Finally, our third phase introduces the ability to reason to a limited degree about subtyping. Across the three phases, we added or modified a total of 3084 lines of code.

Much like the original design of Coccinelle, we focus on pragmatism and provide most features on a best-effort basis, without proving correctness. Like Coccinelle, Coccinelle4J does not crash or throw any errors when it encounters code that it cannot parse or transform. Instead, it recovers from any errors and ignores the parts of the code that it cannot handle.

3.1 Phase 1: Middleweight Java

Middleweight Java is a minimal but valid subset of Java that still retains features of Java such as field assignments, null pointers, and constructor methods. The syntax of Middleweight Java programs is as follows:

[5] https://github.com/google/gson
[6] https://gitbox.apache.org/repos/asf?p=commons-io.git
[7] https://gitbox.apache.org/repos/asf?p=commons-lang.git
[8] https://github.com/mockito/mockito
[9] https://github.com/SeleniumHQ/selenium

Program

$p ::= cd_1...cd_n; \bar{s}$

Class definition

$cd ::= \texttt{class}\ C\ \texttt{extends}\ C\ \texttt{\{}$

$\quad fd_1...fd_k$

$\quad cnd$

$\quad md_1...md_n$

$\texttt{\}}$

Field definition

$fd ::= C f;$

Constructor definition

$cnd ::= C(C_1 x_1, ..., C_j x_j)\ \texttt{\{}$

$\quad \texttt{super(}e_1, ..., e_k\texttt{)}; s_1...s_n$

$\texttt{\}}$

Method definition

$md ::= \tau\ m(C_1 x_1, ..., C_n x_n)\ \texttt{\{}s_1...s_k\texttt{\}}$

Return type

$\tau ::= C \mid \texttt{void}$

Expression

$e ::= x$ *Variable*

$\mid \texttt{null}$ *Null*

$\mid e\texttt{.}f$ *Field access*

$\mid \texttt{(}C\texttt{)}e$ *Cast*

$\mid pe$ *Promotable expression*

Promotable Expression

$pe :: = e.m(e_1, ..., e_k)$ *Method invocation*

$\mid \texttt{new}\ C\texttt{(}e_1, ...e_k\texttt{)}$ *Object creation*

Statement

$s :: = \texttt{;}$ *No-op*

$\mid pe;$ *Promoted expression*

$\mid \texttt{if (}e\texttt{==}e\texttt{)}\ s_1...s_k\ \texttt{else}\ s_{(k+1)}...s_n$ *Conditional*

$\mid e\texttt{.}f\texttt{=}e;$ *Field assignment*

$\mid C\ x;$ *Local variable declaration*

$\mid x\texttt{=}e;$ *Variable assignment*

$\mid \texttt{return}\ e;$ *Return*

$\mid \texttt{\{}s_1...s_n\texttt{\}}$ *Block*

Middleweight Java is imperative and does not add any new control flow structure as compared to C. We thus map each AST element in Middleweight Java to an AST element in C. When compared to C, the three main additions of Middleweight Java are method invocation, object creation and constructor declarations. We map method invocations and object creations to function calls, and constructor declarations to function declarations. Class definitions are ignored as Coccinelle4J only allows matching and transformations of functions.

While we try to support the syntax of Java, we follow the goals of Coccinelle to decide the scope of our analysis on the source code [14]. One of Coccinelle's goals is to keep performance acceptable when used by developers on their development laptops. Thus, we always favor decisions keeping Coccinelle4J fast.

When the source code references an identifier, we do not attempt to resolve its type information that is not immediately obvious from the file getting parsed, such as the interfaces implemented by its class or its class hierarchy. We note that our choice contrasts with the design of some popular Java code-manipulation tools. For example, both Spoon [25] and Soot [27] provide an option for the user to pass the tool a classpath, allowing developers to specify where third-party library method and types can be found. Our choice is motivated by the statistics of widespread change in Table 1, where changes related to modifications in sub-typing or inheritance only comprise 6% of commits making widespread changes. When these transformations are required, a more appropriate tool such as Spoon can be used instead, as Coccinelle4J is complementary to it. We further discuss this decision and other implications of only allowing a limited form sub-typing and inheritance in Phase 3 (Section 3.3), the performance of Coccinelle4J in Section 4, and a brief comparison to Spoon in Section 4.8.

3.2 Phase 2: Control-flow with Exceptions and Java-specific Isomorphisms

Our second phase of the development of Coccinelle4J adds exceptions in the context of `try-catch`, to allow Coccinelle4J to parse and process a wider range of Java programs. To further improve the usability of Coccinelle4J, we add Java-specific isomorphisms, allowing some kinds of Java pattern code to match related syntactic variants.

3.2.1 Control-flow with exceptions

Middleweight Java does not allow code to throw exceptions, other than `ClassCastException` and `NullPointerException`, and does not provide syntax for `try` and `catch`. While this may be convenient for analysis, exceptions are used heavily in real-world Java programs. For example, about 25% of the Java classes in the source code of the five Java libraries previously discussed in Section 3 contain exception handling code. To allow Coccinelle4J to be more useful in practice, we add support for handling exceptions in the context of `try-catch`. Coccinelle4J does not deal with code throwing exceptions out of a method as it only performs intraprocedural analysis.

Given the code snippet in Listing 2, adapted from the code found in the NextCloud Android project,[10] we construct the control flow graph for the `try` statement as shown in Figure 2. As Coccinelle4J ignores the signatures of invoked methods from project dependencies and even within the same project, it does not know if any of the methods or constructors will throw an exception. It thus over-approximates the set of paths that can be taken, adding edges between all the nodes in the `try` block and the start of the `catch` block.

[10] `https://github.com/nextcloud/android`

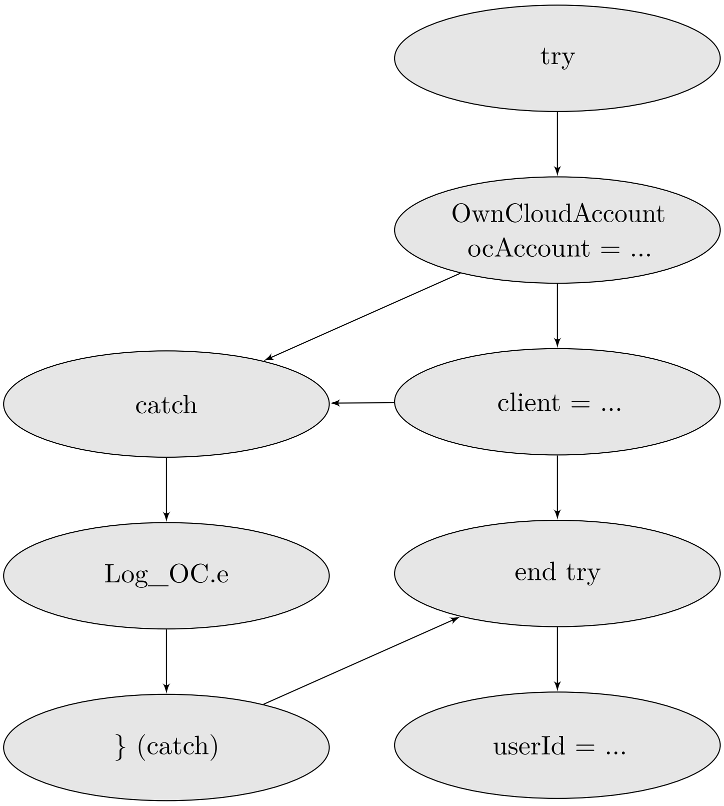


Figure 2 Control flow graph of try-catch.

Listing 2 Code snippet from the NextCloud Android project, modified for conciseness.

```java
public RemoteTrashbinRepository(Context context) {
    AccountManager accountManager = AccountManager.get(context);
    Account account = AccountUtils.getCurrentOwnCloudAccount(
        context);
    try {
        OwnCloudAccount ocAccount = new OwnCloudAccount(account,
            context);
        client = OwnCloudClientManagerFactory.getDefaultSingleton()
            .getClientFor(ocAccount, context);
    } catch (Exception e) {
        Log_OC.e(TAG, e.getMessage());
    }
    userId = accountManager.getUserData(account,
        com.owncloud.android.lib.common.accounts.AccountUtils.
        Constants.KEY_USER_ID);
}
```

There are some ramifications of performing the control flow analysis in this way. For example, consider the following code snippet (Listing 3) and semantic patch (Listing 4), where `thisWontThrow` will never throw an `ExceptionA`. The control-flow graph created by Coccinelle4J has a path where `reachedHere` is reached. As the match succeeds, the Coccinelle adds the new invocation of `log.info`. However, in reality, the code in the `catch (ExceptionA e)` block is unreachable.

Listing 3 Hypothetical code where thisWontThrow() will not throw an exception.

```
ObjectA A = new ObjectA();
try {
  A.thisWontThrow();
  reachedEnd = true;
} catch (ExceptionA e) {
  A.reachedHere();
}
```

Listing 4 Semantic patch adding a log message after A.reachedHere.

```
@@
@@
A.thisWontThrow();
... when exists
A.reachedHere();
+ log.info("reached end of catch block");
```

In practice, we do not expect this to be common in real projects or be surprising to a user of Coccinelle4J. If `ExceptionA` is a checked exception, the Java compiler reports an error if the exception is never thrown in the `try` statement. For unchecked exceptions, as method signatures do not contain information about them, having the method signatures of the invocations in the `try` block will not help us prevent over-approximating the set of possible paths.

3.2.2 Isomorphisms

The fact that SmPL expresses patterns in terms of source code makes it easy to learn, but risks tying the specification to the coding style habits of a given developer. To address this issue, Coccinelle's engine handles *isomorphisms* during matching. Isomorphisms describe semantically equivalent code fragments that are used to expand SmPL rules, allowing syntactically different source code with the same semantics to be matched. For example, it allows `i++` (postfix increment) to be successfully matched against queries for `i=i+1`.

In Coccinelle4J, we disable Coccinelle isomorphism rules that are not relevant for Java (for example, pointer-related isomorphisms), and add Java-specific rules such as considering `A.equals(B)` to be semantically identical to `B.equals(A)`.

One particular difficulty we face in Java is that fully qualified names are considered by Coccinelle4J to be different from non-fully qualified ones even when referring to the same enumeration or method. This results in long patches specifying the same name multiple times using different qualifiers. To resolve this issue, we add a preprocessor to generate isomorphisms based on how names are qualified in a project. For example, when the rules in Listing 5 are enabled, users can specify "sqrt" in SmPL to match `Math.sqrt` or `java.lang.Math.sqrt` in the Java source code. This allows for more concise semantic patches. From our analysis of the 5 Java libraries previously discussed at the start of Section 3, less than 10% of method names and enumeration names were not unique when considering all of the projects together. Therefore, in many contexts, users do not need to distinguish between invocations of these different types of methods as there is often little risk in matching undesired function invocations of the same name.

Listing 5 Generated isomorphism rules, named java_lang_Math_sqrt and Math_sqrt, conflating qualified names and non-qualified names.

```
Expression
@java_lang_Math_sqrt@
@@
sqrt => java.lang.Math.sqrt

Expression
@Math_sqrt@
@@
sqrt => Math.sqrt
```

Isomorphisms can be disabled on an individual basis within each rule by specifying `disable <isomorphism name>` beside the rule name or by passing a command line option to Coccinelle4J. The isomorphisms generated for matching fully qualified names are named "<package name>_<class name>_<identifier>" with the "." in the package name replaced by "_". As such, the rules in Listing 5 can be disabled by specifying `disable java_lang_Math_sqrt` and `disable Math_sqrt`. New isomorphisms can also be added by users.

3.3 Phase 3: Matching programs with sub-typing

Java projects often make use of sub-typing. Developers writing semantic patches may expect that a variable can be matched based on both its declared interface and the type specified in its constructor. Therefore, we permit a limited form of matching on variable types. Coccinelle4J annotates a variable using both its declared interface and its constructor. As a simple example, the metavariable `x`, declared with `ArrayList<Integer>` in the patch in Listing 6 can bind to the variable `numbers` in the statements in Listing 7. Coccinelle originally would only match if `x` was declared with `List<Integer>` in the semantic patch.

Listing 6 Semantic patch matching identifiers of type ArrayList<Integer>.

```
@@
ArrayList<Integer> x;
identifier f;
@@
* f(x)
```

Listing 7 Statements with a List<Integer> variable.

```
List<Integer> numbers = new ArrayList<Integer>();
doSomething(numbers);
```

Coccinelle4J's support for subtyping builds on Coccinelle's support for `typedef`. In C, `typedef` allows developers to introduce synonyms for types. For example, `typedef short s16;` introduces a new type alias `s16` for the type `short`. During program matching, developers may expect that writing `short` in SmPL matches expressions of type `s16` in the source code.

Coccinelle already resolves type information related to `typedef`s during program matching. When parsing C programs, Coccinelle tracks type information for each `typedef` and annotates each program element in the C source code with the types that it can be matched on. Whenever an element declared as a typedef is reached, Coccinelle tries to resolve the type information from the relevant `typedef` declaration. In Coccinelle4J, we reuse this mechanism to deal with sub-typing in Java, maintaining a list of types that each program element can be matched on.

One limitation is that we only match interfaces to object instances of their sub-types if the variables were explicitly declared with their interface. For example, `numbers` in Listing 7 implements multiple interfaces including `List<Integer>` and `Iterable<Integer>`. In Coccinelle4J, we only match `numbers` on the explicitly declared types `List<Integer>` and `ArrayList<Integer>`. Information such as the inheritance path of the identifier's class is also ignored; `numbers` inherits `AbstractList<Integer>`, `AbstractCollection<Integer>`, and `Object`, but Coccinelle4J will not match them. As a result of this decision, in Listing 8, patterns specifying `Deque<UserInfoDetailsItem>` will not match any statement in the Listing even though `LinkedLists` satisfy the `Deque` interface, since it is not declared as a `Deque` here.

Listing 8 Code snippet modified from NextCloud Android. This code snippet matches `List<UserInfoDetailsItem>` or `LinkedList<UserInfoDetailsItem>`, but not `Deque<UserInfoDetailsItem>`.

```
List<UserInfoDetailsItem> result = new LinkedList<>();
addToListIfNeeded(result, R.drawable.ic_phone, userInfo.getPhone(),
                  R.string.user_info_phone);
return result;
```

In practice, allowing a semantic patch specifying an interface/parent class to match on all uses of sub-types/child classes may result in many unwanted results. Classes may implement multiple interfaces, and when given a particular context, it is improbable that every implemented interface is relevant. In many situations where matching an interface or parent class is intended, specifying the interface or its methods will suffice. In most cases, matching on explicit declarations is sufficient, since it is idiomatic in Java to "Refer to objects by their interfaces" [2]. For example, `numbers` is declared with its interface `List<Integer>` in Listing 7 and `result` is declared with `List<UserInfoDetailsItem>` in Listing 8.

We verified this idiom empirically. In the Java libraries we studied, 75% of method parameters and local variable declarations were declared with an interface, whenever such an interface (excluding low-level interfaces from `java.lang` and `java.io`) existed. In cases where variables are not declared with their interfaces and we wish to detect them, specifying the methods on the interface can be attempted next. For example, the `Cloneable` interface is implemented by numerous classes. If the user intends to find parts of programs where objects are cloned, it is sufficient to specify the invocation of the `clone()` method in the semantic patch.

Another reason for this decision is that resolving the type of every identifier will incur a performance penalty, and we would like to avoid the cost of trying to resolve project dependencies or having to fetch type information from third-party libraries. Coccinelle4J is designed to be used by developers for rapid prototyping, one rule at a time, so that users can receive feedback from the tool quickly. Thus, one of our goals is to keep Coccinelle4J lightweight.

4 Case Study: Migrating APIs

Android applications may have many dependencies on third-party libraries or frameworks, and keeping up with changes in those dependencies may introduce a significant maintenance burden. Due to security or performance issues, libraries or frameworks deprecate API methods and these methods are eventually removed [16]. It is therefore important to upgrade the uses of deprecated methods to their more modern alternatives. McDonnell et al. studied the impact of API evolution of Android on developer adoption [17], and they found that it takes time for developers to catch up with new API versions. They also observed that API updates are more probable to lead to defects than other changes. We note that this shows the importance of providing developers with a faster and more reliable way of updating API usage.

We show examples of deprecated Android API methods with typical migration pathways, and show possible semantic patches for these migrations. Using these examples, we show that typical migrations can be written concisely in SmPL. We order these examples in approximately increasing order of complexity. In Section 4.1, 4.2, and 4.3, we show basic features of Coccinelle4J to replace method invocations, arguments to them, and their method receivers. In Section 4.4, we describe a case requiring a change in logic due to a different return type of the replacement API, and in Section 4.5, we distinguish between the different ways the deprecated API can be used based on its context. In Section 4.6, we replace the parameters of a method overriding a deprecated method from its parent class. Finally, we discuss an example where we encounter some false positives during program matching due to the limitations described earlier.

As some method invocations may have numerous call sites across a project, migration of these API methods is a form of widespread change. We show that the use of semantic patches may reduce developer effort. This also serves as a short guide on specifying program transformations in SmPL, and we show how context information may be useful. While performing the migrations, there are two questions that we wish to answer:

- Is SmPL expressive enough to perform widespread changes in Java projects?
- How much development effort can a developer potentially save by the use of semantic patches?

4.1 Removing sticky broadcasts

We use SmPL to demonstrate two related trivial migrations of replacing invocations of a deprecated method while keeping the same arguments, and removing the invocations of another method.

The use of sticky broadcasts was discouraged in the release of Android API level 21 due to problems such as the lack of security, therefore the methods `sendStickyBroadcast(Intent)` and `removeStickyBroadcast(Intent)` were deprecated. The official API reference[11] recommends `sendBroadcast(Intent)` to replace `sendStickyBroadcast(Intent)`, while usage of `removeStickyBroadcast(Intent)` should be removed. It was suggested that the developer should also provide some mechanism for other applications to fetch the data in the sticky broadcasts. Providing this latter mechanism is highly dependent on the Android application, and requires knowledge of what the broadcast receivers expect of the broadcast. However, replacing and removing these method calls can be expressed in SmPL.

Listings 9 and 10 illustrate the use of these API methods in the NextCloud Android project, while Listing 11 shows a semantic patch that handles this migration. The call site of `removeStickyBroadcast` is an example motivating that migration of function invocations on exceptional control flow may be required, as `removeStickyBroadcast` may be included in a `finally` block to ensure that it always executes. In this patch, we use the '|' operator, i.e., the *sequential disjunction* operator of SmPL, to remove invocations of `removeStickyBroadcast`. The '|' operator allows matching and transformation using any of the terms delimited by '|' within the parenthesis. Earlier matches take precedence over later ones, for a given node in the control-flow graph. In this case, the rule matches on either `sendStickyBroadcast` or `removeStickyBroadcast`. This allows for the succinct expression of multiple matching code that have similar contexts.

[11] `https://developer.android.com/reference/android/content/Context.html#sendStickyBroadcast(android.content.Intent)`

Listing 9 Example call site of sendStickyBroadcast in the NextCloud Android project.

```
private void sendBroadcastUploadsAdded() {
    Intent start = new Intent(getUploadsAddedMessage());
    // nothing else needed right now
    start.setPackage(getPackageName());
    sendStickyBroadcast(start);
}
```

Listing 10 Example call site of removeStickyBroadcast in the NextCloud Android project.

```
try {
    ... // omitted for brevity
    removeStickyBroadcast(intent);
    Log_OC.d(TAG, "Setting progress visibility to " + mSyncInProgress);
} catch (RuntimeException e) {
    // comments omitted for brevity
    removeStickyBroadcast(intent);
    DataHolderUtil.getInstance().delete(
        intent.getStringExtra(FileSyncAdapter.EXTRA_RESULT));
}
```

Listing 11 Semantic patch for migrating from sendStickyBroadcast.

```
@@
Intent intent;
@@
(
- sendStickyBroadcast(intent);
+ sendBroadcast(intent);
|
- removeStickyBroadcast(intent);
)
```

4.2 setTextSize → setTextZoom

Li et al. have inferred a collection of mappings from deprecated Android API methods to their replacement methods [16]. A typical migration in this collection requires modifying both method names and method arguments, either by adding or removing arguments, or by changing the arguments' type. One such transformation was required when `WebSettings.setTextSize(WebSettings.TextSize)` was deprecated and replaced by `WebSettings.setTextZoom(int)`. This involves transforming both the method name and the method arguments. Listing 12 shows part of the patch on the Lucid-Browser project[12] and Listing 13 then shows a semantic patch producing it. The semantic patch modifies both the method's name and its arguments in a single rule.

Listing 12 Part of a patch replacing invocations of setTextSize in the Lucid-Browser project.

```
if (Properties.webpageProp.fontSize==0)
-   this.getSettings().setTextSize(WebSettings.TextSize.SMALLEST);
+   this.getSettings().setTextZoom(50);
if (Properties.webpageProp.fontSize==1)
-   this.getSettings().setTextSize(WebSettings.TextSize.SMALLER);
+   this.getSettings().setTextZoom(75);
```

[12] `https://github.com/powerpoint45/Lucid-Browser`

```
if (Properties.webpageProp.fontSize==2)
-   this.getSettings().setTextSize(WebSettings.TextSize.NORMAL);
+   this.getSettings().setTextZoom(100);
if (Properties.webpageProp.fontSize==3)
-   this.getSettings().setTextSize(WebSettings.TextSize.LARGER);
+   this.getSettings().setTextZoom(150);
if (Properties.webpageProp.fontSize==4)
-   this.getSettings().setTextSize(WebSettings.TextSize.LARGEST);
+   this.getSettings().setTextZoom(200);
```

Listing 13 Semantic patch replacing usage of setTextSize.

```
@@
expression E;
@@
(
- E.setTextSize(LARGEST);
+ E.setTextZoom(200);
|
- E.setTextSize(LARGER);
+ E.setTextZoom(150);
|
- E.setTextSize(NORMAL);
+ E.setTextZoom(100);
|
- E.setTextSize(SMALLER);
+ E.setTextZoom(75);
|
- E.setTextSize(SMALLEST);
+ E.setTextZoom(50);
)
```

This example illustrates the usefulness of the isomorphisms conflating fully qualified and non-fully qualified class names. While in most projects, `WebSettings.TextSize.LARGEST` is often qualified with `WebSettings.TextSize`, some projects, such as K-9 Mail,[13] use the shorter `TextSize.LARGEST`. Listings 14 and 15 show the isomorphism rules generated for `LARGEST`.

Listing 14 Isomorphism rule for expanding LARGEST for Lucid Project.

```
Expression
@ WebSettings_TextSize_LARGEST @
@@
LARGEST => WebSettings.TextSize.LARGEST
```

Listing 15 Isomorphism rule for expanding LARGEST for K-9 Mail.

```
Expression
@ TextSize_LARGEST @
@@
LARGEST => TextSize.LARGEST
```

4.3 Resources.getColor → ContextCompat.getColor

Another migration pathway is changing the method receiver. We use the deprecation of `Resources.getColor(int)` as an example. One approach to replace this deprecated method is to use the static method `ContextCompat.getColor(Context, int)`, which was specifically introduced to help migrate from the deprecated method. A commit making such a change

[13] https://github.com/k9mail/k-9/commit/f8695f9a61c8a411a09ccee8c8bf739149f0f17e

can be found in the Kickstarter Android project.[14] A patch, similar to that commit, that can be applied on Kickstarter Android is in Listing 16 and the semantic patch producing this patch is in Listing 17.

Listing 16 Part of a patch replacing getColor from Kickstarter Android.

```
return new NotificationCompat.Builder(context)
       .setSmallIcon(R.drawable.ic_kickstarter_k)
-      .setColor(context.getResources().getColor(R.color.green))
+      .setColor(ContextCompat.getColor(context, R.color.green))
       .setContentText(text)
       .setContentTitle(title)
       .setStyle(new NotificationCompat.BigTextStyle().bigText(text))
       .setAutoCancel(true);
```

Listing 17 Semantic patch replacing uses of getColor.

```
@@
Context ctx;
expression E;
@@
- ctx.getResources().getColor(E)
+ ContextCompat.getColor(ctx, E)
```

4.4 AudioManager.shouldVibrate(int) → AudioManager.getRingerMode()

In some cases, migration requires more changes than just replacing the deprecated method with another and requires additional logic to be added by the developer. For example, the migration pathway for `AudioManager.shouldVibrate` was to replace it with a comparison of the ringer mode retrieved from `AudioManager.getRingerMode`. This deprecation required an application to maintain its own policy for allowing vibration of the phone based on the phone's current ringer mode. Support for both older and newer Android versions may be kept by using the deprecated method only on earlier Android versions, while using the replacement method on later Android versions. An example showing the result of the required transformation can be found in the Signal Android application[15] shown in Listing 18. Listing 19 shows a semantic patch that adds a new function that dispatches a call to the right method after checking the Android version, and replaces the deprecated method invocation with this method. A default policy of allowing vibration on non-silent modes is assumed in this patch, but developers can modify the semantic patch to customize the vibrate policy for their own applications appropriately.

Listing 18 Example code invoking the deprecated method from Signal Android.

```
private boolean shouldVibrate(Context context, MediaPlayer player,
        int ringerMode, boolean vibrate) {
    if (player == null) {
      return true;
    }

```

[14] https://github.com/kickstarter/android-oss/commit/053b0a32731bd9a4e9dd42c297565f87145a964b

[15] https://github.com/signalapp/Signal-Android/blob/f9adb4e4554a44fd65b77320e34bf4bccf7924ce/src/org/thoughtcrime/securesms/webrtc/audio/IncomingRinger.java

```
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.JELLY_BEAN) {
      return shouldVibrateNew(context, ringerMode, vibrate);
    } else {
      return shouldVibrateOld(context, vibrate);
    }
  }

  @TargetApi(Build.VERSION_CODES.HONEYCOMB)
  private boolean shouldVibrateNew(Context context,
          int ringerMode, boolean vibrate) {
    Vibrator vibrator = (Vibrator) context.getSystemService(
        Context.VIBRATOR_SERVICE);

    if (vibrator == null || !vibrator.hasVibrator()) {
      return false;
    }

    if (vibrate) {
      return ringerMode != AudioManager.RINGER_MODE_SILENT;
    } else {
      return ringerMode == AudioManager.RINGER_MODE_VIBRATE;
    }
  }

  private boolean shouldVibrateOld(Context context, boolean vibrate) {
    AudioManager audioManager = ServiceUtil.getAudioManager(context);
    return vibrate &&
        audioManager.shouldVibrate(AudioManager.VIBRATE_TYPE_RINGER);
  }
```

Listing 19 Semantic Patch replacing uses of shouldVibrate.

```
@@
identifier am, f, ctx;
expression vibrate_type;
@@
+ boolean shouldVibrate(AudioManager am, Context ctx, int vibrateType) {
+     if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.JELLY_BEAN) {
+         Vibrator vibrator = (Vibrator) ctx. getSystemService(
+             Context.VIBRATOR_SERVICE );
+         if (vibrator == null || !vibrator.hasVibrator()) {
+                 return false;
+         }
+         return am.getRingerMode() != AudioManager.RINGER_MODE_SILENT;
+     } else {
+         return audioManager.shouldVibrate(vibrateType);
+     }
+ }
f(..., Context ctx, ...) {
...
- am.shouldVibrate(vibrate_type)
+ shouldVibrate(am, ctx, vibrate_type)
...
}
```

4.5 Display.getHeight() and Display.getWidth() → Display.getSize(Point)

Deprecated methods can be used in different ways, and migration requires consideration of how they were used. For example, in the release of Android API Level 15, `Display.getHeight` and `Display.getWidth` were deprecated in favor of constructing a `Point` object, initializing it using `Display.getSize(Point)`, before obtaining the height and width using `Point.y` and

`Point.x`. Listings 20 and 21 show two examples of code (from the Materialistic for Hacker News application[16] and the Glide library[17]) managing the deprecation. Both examples check for the currently installed Android version, and invoke the deprecated method only on earlier versions of Android where the method has not been deprecated.

These two examples show the need to distinguish between code requiring a `Point` object instance, and code requiring just the height of a display. In Listing 21, a `Point` object is already constructed and only the way it is initialized requires modification. In Listing 20, however, only the height of the display is required. Listing 22 shows a simplified semantic patch that consists of two rules to fix the two variants of this deprecation. The tokens `<...` and `...>` form a variant of "`...`" indicating that matching `display.getHeight()` within the path is optional and can occur multiple times. In this case, we use `<... ...>` to specify that we wish to transform all of its occurrences. The use of `... when != Point(...)` in rule2 omits matches where a `Point` object has already been created in the control flow context, in order to distinguish between the two cases. Listing 23 shows one example patch produced by Coccinelle4J that is semantically similar to a commit fixing the deprecation in the Android framework itself.[18]

Listing 20 Example from Materialistic of correct usage of the deprecated method, getHeight, after checking for the device's Android version.

```java
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.HONEYCOMB_MR2) {
    displayDimens = new Point();
    display.getSize(displayDimens);
} else {
    displayDimens = new Point(display.getWidth(), display.getHeight());
}
... // omitted for brevity
```

Listing 21 Example from Glide of correct usage of the deprecated method, getHeight, after checking for the device's Android version.

```java
... // omitted for brevity
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.HONEYCOMB_MR2) {
    Point point = new Point();
    display.getSize(point);
    return point.y;
} else {
    return display.getHeight();
}
```

Listing 22 Simplified version of a semantic patch to migrate invocations of getWidth and getHeight. The use of != Point(...) prevents rule2 from erroneously matching code transformed for the first case.

```
@rule1@
Display display;
identifier p;
type T;
@@
(
- p = new Point(display.getWidth(),
-               display.getHeight());
+ p = new Point();
```

[16] https://github.com/hidroh/materialistic/commit/baaf9eeb28f09b8d0d875107687a595a2f377f79

[17] https://github.com/bumptech/glide/commit/827fc08222eb61595ab0d5fdebeea0033e9e8382

[18] https://github.com/aosp-mirror/platform_frameworks_base/commit/ac8dea12c17aa047e03a358110aeb60401d36aa2#diff-40287e2f0fc8c327021400d034442324

```
+ display.getSize(p);
|
- T p = new Point(display.getWidth(),
-                 display.getHeight());
+ T p = new Point();
+ display.getSize(p);
)
// '<...' indicates that all occurrences should be replaced
<...
(
- display.getHeight()
+ p.y
|
- display.getWidth()
+ p.x
)
...>

@rule2@
identifier display, f;
expression E;
@@
f(...) {
...
Display display = E;
+ Point p = new Point();
+ display.getSize(p);
// 'when' omits matches where a Point object has already been created
<... when != Point(...)
(
- display.getHeight()
+ p.y
|
- display.getWidth()
+ p.x
)
...>
}
```

■ **Listing 23** Example patch created by Coccinelle4J using the semantic patch above, semantically equivalent to a commit fixing this deprecation in the Android framework.

```
  public static void dragQuarterScreenDown(
            InstrumentationTestCase test, Activity activity) {
   Display display = activity.getWindowManager().getDefaultDisplay();
-     int screenHeight = display.getHeight();
-     int screenWidth = display.getWidth();
+     Point p = new Point();
+     display.getSize(p);
+     int screenHeight = p.y;
+     int screenWidth = p.x;
```

4.6 WebChromeClient.onConsoleMessage(String, int, String) → WebChromeClient.onConsoleMessage(ConsoleMessage)

We show how method signatures of classes extending other classes with deprecated methods can be transformed using SmPL. Developers can override `onConsoleMessage(String, int, String)` of `WebChromeClient`, which was deprecated in favour of an overloaded method with a different set of parameters `onConsoleMessage(ConsoleMessage)`. Listing 24 shows example code from the MGit project[19] overriding this method, and Listing 25 shows a semantic patch changing the parameters of `onConsoleMessage(String, int, String)`. The use of `<...` and `...>` specifies that all occurrences of `p1`, `p2`, and `p3` should be transformed.

Listing 24 An example replacement site of onConsoleMessage from MGit.

```
mFileContent.setWebChromeClient(new WebChromeClient() {
    @Override
    public void onConsoleMessage(String message, int lineNumber,
                                 String sourceID) {
        Log.d("MyApplication", message + " -- From line " + lineNumber
                + " of " + sourceID);
    }
... // other overridden methods
})
```

Listing 25 Semantic patch updating method signatures of onConsoleMessage.

```
@@
identifier p1, p2, p3;
@@
- onConsoleMessage(String p1, int p2, String p3) {
+ onConsoleMessage(ConsoleMessage cs) {
<...
(
- p1
+ cs.message()
|
- p2
+ cs.lineNumber()
|
- p3
+ cs.sourceId()
)
...>
}
```

4.7 Resources.getDrawable(int) → Resources.getDrawable(int, Theme)

In Phase 3 of developing Coccinelle4J, we made some trade-offs that introduced the limitations discussed in Section 3. While we do not run into problems writing the majority of the patches, some patches would have benefited from Coccinelle4J having the ability to resolve types, one of the features deliberately omitted to keep Coccinelle4J fast. In the release of Android API level 22, `Resources.getDrawable(int)` was deprecated in favour of `getDrawable(int, Theme)`, the same method overloaded with an additional `Theme` parameter. After inspecting calls to `getDrawable` in a project, a developer may write a semantic patch to find these calls and transform them using the simple patch in Listing 26.

Listing 26 Semantic patch updating invocations to getDrawable.

```
@@
expression E;
expression p;
@@
- E.getDrawable(p)
+ E.getDrawable(p, getContext().getTheme())
```

[19] `https://github.com/maks/MGit`

While all the locations that should be transformed are correctly transformed, this also results in several false positives, as illustrated by the patch in Listing 27, based on code from the NextCloud Android project. As the semantic patch does not assert that the metavariable `E` is of type `Resources`, Coccinelle4J updates a `getDrawable` invocation on a `TypedArray` instead of `Resources`. This results in code that does not compile as `TypedArray` does not have a method with a `getDrawable(int, Theme)` signature.

Listing 27 Example patch created by Coccinelle4J for the above semantic patch to replace getDrawable.

```
      public SimpleListItemDividerDecoration(Context context) {
          super(context, DividerItemDecoration.VERTICAL);
          final TypedArray styledAttributes =
             context.obtainStyledAttributes(ATTRS);
-         divider = styledAttributes.getDrawable(0);
+         divider = styledAttributes.getDrawable(0,
+             getContext().getTheme());
          leftPadding = Math.round(72 *
             (context.getResources().getDisplayMetrics().xdpi /
             DisplayMetrics.DENSITY_DEFAULT));
          styledAttributes.recycle();
      }
```

On the other hand, if one does specify the need for a `Resource`, as illustrated in Listing 28, then Coccinelle4J misses locations where a `Resource` variable is not explicitly declared. For example, a `Resource` may be produced from an invocation as part of a larger expression, as in `getResources().getDrawable(R.drawable.ic_activity_light_grey)`. The metavariable `Resources R` will not match the invocation of `getResources`. In cases like this, developers may be surprised that there is no way to write a rule in SmPL with a metavariable binding to the invocation of `getResources` based on its return type.

Listing 28 Semantic patch updating invocations to getDrawable.

```
@@
Resources R;
expression p;
@@
- R.getDrawable(p)
+ R.getDrawable(p, getContext().getTheme())
```

4.8 Evaluation

Using the case of performing changes to multiple locations, we evaluate Coccinelle4J based on how much it helped development efficiency, the ease of specifying patches, and the speed at which it applied patches to projects. We also report limitations we experienced with Coccinelle4J.

We use development efficiency metrics that were previously used to evaluate Coccinelle for backporting device drivers [26]. Table 2 shows the ratio of the number of source-code insertions and deletions that Coccinelle4J generates, the number of lines in the semantic patch, excluding whitespace, and the number of files changed by Coccinelle4J. Comparing these quantities indicates the amount of savings from the use of semantic patches. We use the patches updating the uses of `sendStickyBroadcast` on the NextCloud Android project (Section 4.1),[20] `setTextSize` on the Lucid-Browser (Section 4.2), `getColor` on the Kickstarter

[20] `https://github.com/nextcloud/android`

Table 2 Summary of the patches generated by Coccinelle4J in our case study.

Patch	Semantic patch size	Lines generated	Ratio	Files changed
sendStickyBroadcast	9	19	2.11	5
setTextSize	19	10	0.53	1
getColor	6	16	2.67	4
shouldVibrate	35	17	0.49	1
getHeight	44	3	0.07	1
onConsoleMessage	18	14	0.78	2

Android project (Section 4.3), `shouldVibrate` on Signal (Section 4.4), `getHeight` on Glide (Section 4.5), and `onConsoleMessage` on the MGit project (Section 4.6). In total, these projects contain 311755 lines of code and 2030 source files.

The development effort saved of using Coccinelle4J increases in proportion with the number of replacement sites. For changes that have to be made at multiple locations, Coccinelle4J will save more effort. In our examples, as there were only a few uses of `shouldVibrate`, `getHeight`, and `onConsoleMessage` in MGit, Signal, and Glide, the number of lines of code in the semantic patch is greater than the number of lines of code of the actual change. However, the amount of code changed is an incomplete measure of the development effort saved, as Coccinelle4J also helps in locating the files requiring modification, as well as in identifying the correct locations in each file. Furthermore, as these are methods of the Android API, they will be commonly used by many projects, and each patch will have numerous matches when considering all Android repositories.

In all our examples, the semantic patches to update the call sites of the deprecated methods are concise. The semantic patches are declarative and describe what code will be changed after their application to the source code.

We briefly compare the expressiveness of SmPL to Spoon [25]. With Spoon, there are currently no abstractions over control-flow constraints. Transformations in Spoon are less declarative compared to SmPL. Using the example of `onConsoleMessage` (Listing 25) discussed earlier in Section 4.6, where we change its parameters, Listing 29 shows part of a processor written in Spoon. As compared to the semantic patch, the Spoon processor is imperative and requires reasoning about Spoon's meta-model, whereas SmPL takes a WYSIWYG approach.

Listing 29 Replacing parameters of onConsoleMessage using Spoon.

```
// ...
// Code for selecting the onConsoleMessage method omited
public void process(CtMethod method) {
    List<CtParameter> params = method.getParameters();
    while (!params.isEmpty()) {
        CtParameter param = params.get(0);
        param.delete();
        params.remove(param);
    }
    CtParameter<ConsoleMessage> newParam = getFactory().Core()
        .createParameter();
    newParam.setSimpleName("consoleMessage");
    newParam.setType(getFactory().Type()
        .createReference(ConsoleMessage.class));
    method.addParameter(newParam);
    // ...
}
// ...
```

Listing 30 Replacing parameters of onConsoleMessage using SmPL.

```
@@
identifier p1, p2, p3;
@@
- onConsoleMessage(String p1, int p2, String p3) {
+ onConsoleMessage(ConsoleMessage consoleMessage) {
...
}
```

We further elaborate on other convenience features Coccinelle4J provides. While Spoon has templates for transformation, developers have to specify template parameters and write logic in Spoon processors to extract or construct the corresponding arguments. Using the example discussed in Section 4.2 about replacing `setTextSize` with `setTextZoom`, we define a Spoon template in Listing 31. This template has two parameters, an expression returning a `WebSettings` that `setTextSize` is invoked on, and a literal that is passed as an argument to `setTextSize`. A Spoon processor (Listing 32) has to select these meta-model elements from the matching statement, and pass them as arguments to the template to generate a new statement, before replacing the original statement invoking `setTextSize`. The corresponding patch in SmPL was given previously in Section 4.2. In contrast to Spoon, a user of Coccinelle4J does not need to think about these low-level details, as Coccinelle4J binds metavariables to subterms while matching code and automatically applies them during transformation.

Listing 31 Spoon template for producing statements using setTextZoom.

```
public class ReplaceTemplate extends StatementTemplate {
    public ReplaceTemplate(CtExpression<WebSettings> _settings,
            CtLiteral<Integer> _value) {
        this._settings = _settings;
        this._value = _value;
    }

    @Parameter
    CtExpression<WebSettings> _settings;

    @Parameter
    CtLiteral<Integer> _value;

    @Override
    public void statement() {
        _settings.S().setTextZoom(_value.S());
    }
}
```

Listing 32 Spoon processor using the Spoon template replacing invocations of setTextSize.

```
public class SetTextzoomProcessor
        extends AbstractProcessor<CtStatement> {
    ... // other details omitted for brevity
    @Override
    public void process(CtStatement stmt) {
        ... // omitted code to select the expressions:
        // 1. oldValue refers to the argument of setTextSize
        // 2. settingsExpression refers to the expression
        // returning the WebSettings that setTextSize is invoked on.
        if (oldValue.getVariable().getSimpleName().equals("SMALLEST")
            ) {
            value.setValue(50);
        } else if (oldValue.getVariable().getSimpleName()
                    .equals("SMALLER")) {
            value.setValue(75);
        } else if (...) {
            ...
        }
        ... // more code to set a correct value for the template

        ReplaceTemplate template = new ReplaceTemplate(
            settingsExpression, value);
        CtStatement newstmt = template.apply(stmt.getParent(
            new TypeFilter<>(CtType.class)));
        stmt.replace(newstmt);
    }

}
```

We also briefly compare Coccinelle4J to Refaster [31], a tool to refactor code by writing templates of code before and after its transformation. While both SmPL and Refaster templates allow developers to declaratively refine transformation rules, the design of Coccinelle4J allows reasoning on control-flow paths and fundamentally differs from Refaster. The practical advantages are:

- Constraints can be specified on control-flow paths and matching is supported within arbitrary control-flow paths (including around loops), see Listing 1,
- Metavariables can have different values in different control-flow paths, while being forced to have consistent values within a control-flow path,
- Coccinelle4J interleaves addition or removal of code in the same rule. This makes it easy to position code changes in a long series of code elements, see Listing 22.

It may not be easy to add these advantages, which are inherent in Coccinelle4J, to Refaster. Additionally, Refaster lacks several features of Coccinelle4J, such as:

- inheritance of metavariables between rules,
- interaction with scripting languages (OCaml or Python).

Performance-wise, Coccinelle4J runs quickly. Our experiments were performed on a 2017 Macbook Pro with 2.3 GHz Intel Core i5, 8 GB 2133 MHz LPDDR3, likely similar to an average developer's working laptop. We report the time required for Coccinelle4J to perform each transformation. We also report the time required to build the project, but exclude the time for downloading the project dependencies in the first build. We consider this time to be an upper bound on the time that project developers will wait for a tool to complete running. The results are given in Table 3.

In each case, the time to apply a semantic patch on an entire project is nearly negligible, even when there are multiple transformation sites, and it is only a small fraction of the time required to build the project. As such, we conclude that other than the time required to write a semantic patch, usage of the tool will not affect a developer's time negatively.

A limitation of Coccinelle4J is that users may write patches that gives false negatives while matching programs. When writing a semantic patch, it is often the case that one starts with a simple semantic patch that misses more complex cases. We believe that this limitation is circumvented as semantic patches are concise and developers can quickly refine

Table 3 Time to apply semantic patch compared to time to build the project (rounded to the nearest second).

Patch	Project	Time to apply patch	Time to build project
sendStickyBroadcast	NextCloud	3s	46s
setTextSize	Lucid-Browser	1s	21s
getColor	Kickstarter	1s	1 min 6s
shouldVibrate	Signal	1s	1 min 54s
getHeight	Glide	0s	33s
onConsoleMessage	MGit	0s	57s

a semantic patch after inspecting the output produced by Coccinelle4J. The time to apply the updated semantic patch to a project is also negligible. This short feedback loop allows for a fast iterative development process and facilitates exploratory programming, much like working with a read-eval-print loop (REPL). We find this aspect of development lacking in other program transformation tools.

On the other hand, Coccinelle4J inherits some limitations of Coccinelle. One painful aspect of working with Coccinelle4J is it does not always report errors when parsing SMPL in a user-friendly way. This limitation is inherent in yacc-like parsers. While Coccinelle reports the position and identifier in the semantic patch where the error was detected, it may not be immediately clear to a new user of Coccinelle what the error is or how to correct it. Editing semantic patches may be difficult due to the current lack of support in tools familiar to Java developers. For example, while Coccinelle provides some support for Vim and Emacs, there is lack of support for other popular text editors or IDEs that Java developers may be more familiar with. Without basic support that developers may be accustomed to, such as syntax highlighting or code completion, developers may make mistakes that are hard to notice. However, this limitation is circumvented by the ease of adding support for SmPL in text editors, due to the reuse of Java syntax as the code matching language. Providing such support may be a next step to improve the ergonomics of writing SmPL.

Since Coccinelle4J does not guarantee correctness, there is the danger that it may generate semantically incorrect patches (false positives). We believe that this risk is mitigated by two factors. Firstly, most incorrect patches will be caught by the Java compiler. As Coccinelle4J is targeted at changing code in multiple locations, we expect most patches produced by Coccinelle4J to fall into categories of widespread changes, which researchers have categorized for Java projects [30]. Code changes in most of these categories will result in compiler errors if mistakes are made. The iterative process of refining patches also allows mistakes to be detected and identified quickly. Developers can fix incorrect changes manually or revert them with revision control tools. Secondly, in large Java projects where widespread changes are most relevant, there are strict code review processes where errors will be caught by more experienced project maintainers. As such, despite not having the guarantee of producing correct patches, the use of Coccinelle4J will not negatively impact the software development process.

5 Related Work

5.1 Program matching and transformation for Java

There are several existing tools for Java program transformation. Refaster [31] and Spoon [25], already considered in the previous section, are designed to be easy for developers to use. Like Coccinelle4J, they work on the AST and match elements independent of white-space and formatting. However, unlike Coccinelle4J, neither Spoon nor Refaster takes control flow into account when matching code and does not allow for the specification of constraints over control-flow paths. More generally, there is no equivalent to the "..." abstraction in SmPL that abstracts over an arbitrary sequence of code. On the other hand, Spoon allows rules to reason about more precise type information. As such, our work on Coccinelle4J is complementary to the work on Spoon.

Stratego [28], Rascal [10], and TXL [6] are other tools for program transformation that can work on Java. However, these tools requires developers to invest time to learn syntax and formalisms that do not resemble Java.

Soot [27] is a framework for Java program optimization and also offers program analysis tools. Although it provides tools that can compute control flow graphs, it does not provide any program transformation tool that acts on these results.

Another class of Java transformation tools operate on bytecode, unlike Coccinelle4J which works on Java source code. JMangler [11], ASM [4] and Javassist[5] are examples of tools providing APIs for bytecode manipulation, that can be used for tasks like creating new class definitions derived from other classes.

5.2 Migration of APIs

There are several approaches to automated API usage updates. For example, Henkel and Diwan [8] presented a tool to capture API refactoring actions when a developer updates the API usage. LibSync [21] recommends potential edit operations for adapting code based on clients that have already migrated from an API. Semdiff [7] recommends calls to a framework based on how the framework adapted to its own changes. HiMa [18] performs pairwise comparisons in the evolutionary history of a project to construct framework-evolution rules.

Several works also propose program transformation languages to describe rules for mapping APIs calls to alternative APIs. Nita and Notkin describe Twinning [22], a rule-based language that allows developers to specify mappings of blocks of API invocations to sequences of alternative API invocations. SWIN [15] extends Twinning, including adding type safety of transformations. The type safety of SWIN is proved on Featherweight Java [9], a minimal core calculus similar to Middleweight Java.

The approaches above do not allow for context-sensitive many-to-many mappings. Many-to-many mappings refer to the transformation of a sequence of statements using an old API to a new sequence of statements using a replacement API. Wang et al. [29] highlight the difficulty of API migrations when these mappings are required. They show the need to account for control-flow as the statements requiring transformation may take multiple forms. They propose *guided-normalisation* and a language *PATL* for transforming Java programs between different APIs. The semantics of PATL are formalised on Middleweight Java.

Their work differs from ours as their focus is transforming programs between two APIs, a single task where program transformation is useful, while our work targets the more general task of program transformation itself. We use migrations of deprecated APIs as an example only to demonstrate the utility of our program transformation tool. While transforming programs between APIs often focuses on short sequences of function invocations, Coccinelle4J can express constraints and transformations of the code over a longer range, including involving information collected from multiple field and method declarations. There is also no equivalent of the "..." operator in PATL.

An experience report about the use of automated API migration techniques suggests that the difficulty of API migrations lies in the change of API parameter types, rather than selecting an alternative API [13]. This is motivation for our work, as we support the transformation of code and arguments to methods based on the code context. Semantic patches allow argument values to be determined based on code context, including specifying constraints over paths, the use of multiple rules, and information collected from multiple fields and method declarations.

6 Conclusion and Future Work

While SmPL has been shown to be useful for program transformation on C code, its use has not been explored in the context of Java projects. Semantic patches have the benefit of being declarative, are relatively easy to specify, and are unique in how they allow the expression of control-flow patterns. To introduce a tool with these features into the Java ecosystem, we have developed Coccinelle4J, a prototype extending Coccinelle, to support some Java language features. We document our implementation and the design decisions made. Finally, we look at several cases of updating call sites of deprecated Android API in six projects to show the utility of Coccinelle4J. Based on this case study, we discuss its suitability for use in Java projects.

As future work, we hope to evaluate the use of semantic patches and Coccinelle4J for other uses, such as to fix common bugs. From our preliminary work on migrating deprecated Android API, a further extension may be to create a public dataset of reliable semantic patches that developers can apply to their Android projects.

References

1 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.

2 Joshua Bloch. *Effective Java*. Addison-Wesley Professional, 2008.

3 Julien Brunel, Damien Doligez, René Rydhof Hansen, Julia L Lawall, and Gilles Muller. A foundation for flow-based program matching: Using temporal logic and model checking. In *POPL*, pages 114–126. ACM, 2009.

4 Eric Bruneton, Romain Lenglet, and Thierry Coupaye. ASM: A code manipulation tool to implement adaptable systems. *Adaptable and extensible component systems*, 30(19), 2002.

5 Shigeru Chiba. Javassist-a reflection-based programming wizard for Java. In *Proceedings of OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, page 174. ACM, 1998.

6 James R Cordy. The TXL source transformation language. *Science of Computer Programming*, 61(3):190–210, 2006.

7 Barthelemy Dagenais and Martin P Robillard. SemDiff: Analysis and recommendation support for API evolution. In *Proceedings of the 31st International Conference on Software Engineering*, pages 599–602. IEEE Computer Society, 2009.

8 Johannes Henkel and Amer Diwan. CatchUp! Capturing and replaying refactorings to support API evolution. In *27th International Conference on Software Engineering*, pages 274–283. IEEE, 2005.

9 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.

10 Paul Klint, Tijs Van Der Storm, and Jurgen Vinju. Rascal: A domain specific language for source code analysis and manipulation. In *2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 168–177. IEEE, 2009.

11 Günter Kniesel, Pascal Costanza, and Michael Austermann. JMangler-a framework for load-time transformation of Java class files. In *SCAM*, pages 100–110. IEEE, 2001.

12 David Lacey and Oege de Moor. Imperative Program Transformation by Rewriting. In Reinhard Wilhelm, editor, *Compiler Construction*, pages 52–68, 2001.

13 Maxime Lamothe and Weiyi Shang. Exploring the Use of Automated API Migrating Techniques in Practice: An Experience Report on Android. In *15th International Conference on Mining Software Repositories, 2018*, 2018.

14 Julia Lawall and Gilles Muller. Coccinelle: 10 years of automated evolution in the Linux kernel. In *USENIX Annual Technical Conference*, pages 601–614, 2018.

15 Jun Li, Chenglong Wang, Yingfei Xiong, and Zhenjiang Hu. Swin: Towards type-safe Java program adaptation between APIs. In *Proceedings of the 2015 Workshop on Partial Evaluation and Program Manipulation*, pages 91–102. ACM, 2015.

16 Li Li, Jun Gao, Tegawendé F Bissyandé, Lei Ma, Xin Xia, and Jacques Klein. Characterising deprecated Android APIs. In *Proceedings of the 15th International Conference on Mining Software Repositories*, pages 254–264. ACM, 2018.

17 Tyler McDonnell, Baishakhi Ray, and Miryung Kim. An empirical study of API stability and adoption in the Android ecosystem. In *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, pages 70–79. IEEE, 2013.

18 Sichen Meng, Xiaoyin Wang, Lu Zhang, and Hong Mei. A history-based matching approach to identification of framework evolution. In *Proceedings of the 34th International Conference on Software Engineering*, pages 353–363. IEEE Press, 2012.

19 Gilles Muller, Yoann Padioleau, Julia L Lawall, and René Rydhof Hansen. Semantic patches considered helpful. *ACM SIGOPS Operating Systems Review*, 40(3):90–92, 2006.

20 Beevi S Nadera, D Chitraprasad, and Vinod SS Chandra. The varying faces of a program transformation systems. *ACM Inroads*, 3(1):49–55, 2012.

21 Hoan Anh Nguyen, Tung Thanh Nguyen, Gary Wilson Jr, Anh Tuan Nguyen, Miryung Kim, and Tien N Nguyen. A graph-based approach to API usage adaptation. In *OOPSLA*, pages 302–321. ACM, 2010.

22 Marius Nita and David Notkin. Using twinning to adapt programs to alternative APIs. In *2010 ACM/IEEE 32nd International Conference on Software Engineering,*, volume 1, pages 205–214. IEEE, 2010.

23 Yoann Padioleau, Julia L. Lawall, René Rydhof Hansen, and Gilles Muller. Documenting and automating collateral evolutions in Linux device drivers. In *EuroSys*, pages 247–260, 2008.

24 Yoann Padioleau, Julia L Lawall, and Gilles Muller. SmPL: A domain-specific language for specifying collateral evolutions in Linux device drivers. *Electronic Notes in Theoretical Computer Science*, 166:47–62, 2007.

25 Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. Spoon: A library for implementing analyses and transformations of Java source code. *Software: Practice and Experience*, 46(9):1155–1179, 2016.

26 Luis R Rodriguez and Julia Lawall. Increasing automation in the backporting of Linux drivers using Coccinelle. In *Dependable Computing Conference (EDCC), 2015 Eleventh European*, pages 132–143. IEEE, 2015.

27 Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot: A Java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON 1999)*, page 13, 1999.

28 Eelco Visser. Stratego: A language for program transformation based on rewriting strategies system description of Stratego 0.5. In *International Conference on Rewriting Techniques and Applications*, pages 357–361. Springer, 2001.

29 Chenglong Wang, Jiajun Jiang, Jun Li, Yingfei Xiong, Xiangyu Luo, Lu Zhang, and Zhenjiang Hu. Transforming Programs between APIs with Many-to-Many Mappings. In *30th European Conference on Object-Oriented Programming*, 2016.

30 Shaowei Wang, David Lo, and Xingxiao Jiang. Understanding widespread changes: A taxonomic study. In *Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on*, pages 5–14. IEEE, 2013.

31 Louis Wasserman. Scalable, example-based refactorings with Refaster. In *Proceedings of the 2013 ACM Workshop on Refactoring Tools*, pages 25–28. ACM, 2013.

Minimal Session Types

Alen Arslanagić
University of Groningen, The Netherlands

Jorge A. Pérez
University of Groningen, The Netherlands
http://www.jperez.nl/

Erik Voogd
University of Groningen, The Netherlands

Abstract

Session types are a type-based approach to the verification of message-passing programs. They have been much studied as type systems for the π-calculus and for languages such as Java. A session type specifies what and when should be exchanged through a channel. Central to session-typed languages are constructs in types and processes that specify *sequencing* in protocols.

Here we study *minimal session types*, session types without sequencing. This is arguably the simplest form of session types. By relying on a core process calculus with sessions and higher-order concurrency (abstraction-passing), we prove that every process typable with standard (non minimal) session types can be compiled down into a process typed with minimal session types. This means that having sequencing constructs in both processes and session types is redundant; only sequentiality in processes is indispensable, as it can precisely codify sequentiality in types.

Our developments draw inspiration from work by Parrow on behavior-preserving decompositions of untyped processes. By casting Parrow's results in the realm of typed processes, our results reveal a conceptually simple formulation of session types and a principled avenue to the integration of session types into languages without sequencing in types.

2012 ACM Subject Classification Theory of computation → Type structures; Theory of computation → Process calculi; Software and its engineering → Concurrent programming structures; Software and its engineering → Message passing

Keywords and phrases Session types, process calculi, π-calculus

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2019.23

Category Pearl

Related Version Extended version with omitted proofs: http://arxiv.org/abs/1906.03836.

Supplement Material ECOOP 2019 Artifact Evaluation approved artifact available at https://dx.doi.org/10.4230/DARTS.5.2.5

Funding Work partially supported by the Netherlands Organization for Scientific Research (NWO) under the VIDI Project No. 016.Vidi.189.046 (Unifying Correctness for Communicating Software).

Acknowledgements We are grateful to the anonymous reviewers for their remarks and questions. Pérez is also with CWI, Amsterdam and NOVA LINCS – the NOVA Laboratory for Computer Science and Informatics, Universidade Nova de Lisboa, Portugal (Ref. UID/CEC/04516/2019).

1 Introduction

Session types are a type-based approach to the verification of message-passing programs. A session type specifies what and when should be exchanged through a channel; this makes them a useful tool to enforce safety and liveness properties related to communication correctness. Session types have had a significant impact on the foundations of programming languages [15],

33rd European Conference on Object-Oriented Programming (ECOOP 2019).
Editor: Alastair F. Donaldson; Article No. 23; pp. 23:1–23:28
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

but also on their practice [1]. In particular, the interplay of session types and object-oriented languages has received much attention (cf. [8, 7, 14, 11, 2, 18, 25]). In this work, our goal is to understand to what extent session types can admit simpler, more fundamental formulations. This foundational question has concrete practical ramifications, as we discuss next.

In session-typed languages, *sequencing* constructs in types and processes specify the intended structure of message-passing protocols. In the session type $S = ?(\mathsf{Int}); ?(\mathsf{Int}); !\langle \mathsf{Bool} \rangle; \mathtt{end}$, sequencing (denoted ';') allows us to specify a protocol for a channel that *first* receives (?) two integers, *then* sends (!) a Boolean, and *finally* ends. As such, S could type a service that checks for integer equality. Sequencing in types goes hand-in-hand with sequencing in processes, which is specified using prefix constructs (denoted '.'). The π-calculus process $P = s?(x_1).s?(x_2).s!\langle b \rangle.\mathbf{0}$ is an implementation of the equality service: it *first* expects two values on name s, *then* outputs a Boolean on s, and *finally* stops. Thus, name s in P conforms to type S. Session types can also specify sequencing within labeled choices and recursion; these typed constructs are also in close match with their respective process expressions.

Originally developed on top of the π-calculus for the analysis of message-passing protocols between exactly two parties [12], session types have been extended in many directions. We find, for instance, multiparty session types [13] and extensions with dependent types, assertions, exceptions, and time (cf. [6, 15] for surveys). All these extensions seek to address natural research questions on the expressivity and applicability of session types theories.

Here we address a different, if opposite, question: *is there a minimal formulation of session types?* This is an appealing question from a theoretical perspective, but seems particularly relevant to the practice of session types: identifying the "core" of session types could enable their integration in languages whose type systems do not have advanced constructs present in session types (such as sequencing). For instance, the Go programming language offers primitive support for message-passing concurrency; it comes with a static verification mechanism which can only enforce that messages exchanged along channels correspond with their declared payload types – it cannot ensure essential correctness properties associated to the structure of protocols. This observation has motivated the development of advanced static verification tools based on session types for Go programs [22, 21].

This paper identifies an elementary formulation of session types and studies its properties. We call them *minimal session types*: these are session types without sequencing. That is, in session types such as '$!\langle U \rangle; S$' and '$?(U); S$', we decree that S can only correspond to $\mathtt{end}$, the type of the terminated protocol.

Adopting this elementary formulation entails dispensing with sequencing, which is one of the most distinctive features of session types. While this may appear as a far too drastic restriction, it turns out that it is not: our main result is that for every process P that is well-typed under standard (non minimal) session types, there is a *process decomposition* $\mathcal{D}(P)$ that is well-typed using minimal session types. Intuitively, $\mathcal{D}(P)$ codifies the sequencing information given by the session types (protocols) of P using additional synchronizations. This shows that having sequencing in both types and processes is redundant; only sequencing at the level of processes is truly fundamental. To define $\mathcal{D}(P)$ we draw inspiration from a known result by Parrow [24], who proved that untyped π-calculus processes can be decomposed as a collection of *trios processes*, i.e., processes with at most three nested prefixes [24].

The question of how to relate session types with other type systems has attracted interest in the past. Session types have been encoded into generic types [10] and linear types [5, 3, 4]. As such, these prior studies concern the *relative expressiveness* of session types: where the expressivity of session types stands with respect to that of some other type system. In sharp contrast, we study the *absolute expressiveness* of session types: how session types can be explained in terms of themselves. To our knowledge, this is the first study of its kind.

The process language that we consider for decomposition into minimal session types is HO, the core process calculus for session-based concurrency studied by Kouzapas et al. [19, 20]. HO is a very small language: it supports abstraction-passing only and lacks name-passing and recursion; still, it is also very expressive, because both features can be expressed in it in a fully abstract way. As such, HO is an excellent candidate for a decomposition. Being a higher-order language, HO is very different from the (untyped, first-order) π-calculus considered by Parrow in [24]. Also, the session types of HO severely constrain the range and kind of conceivable decompositions. Therefore, our results are not an expected consequence of Parrow's: essential aspects of our decomposition into processes typable with minimal session types are only possible in a higher-order setting, not considered in [24].

Summing up, in this paper we make the following contributions:

1. We identify the class of *minimal session types* as a simple fragment of standard session types that retains its absolute expressiveness.
2. We show how to decompose processes typable with standard session types into processes typable with minimal session types. We prove that this decomposition satisfies a typability result for a rich typed language that includes labeled choices and recursive types.
3. We develop optimizations of our decomposition that bear witness to its robustness.

The rest of the paper is organized as follows. § 2 summarizes the syntax, semantics, and session type system for HO, the core process calculus for session-based concurrency. § 3 presents the decomposition of well-typed HO processes into minimal session types. The decomposition is presented incrementally, starting with a core fragment that is later extended with further features. § 4 presents optimizations of the decomposition. § 5 elaborates further on related works and § 6 concludes.

2 The Source Language

We recall the syntax, semantics, and type system for HO, the higher-order process calculus for session-based concurrency studied by Kouzapas et al. [19, 20].[1] HO is arguably the simplest language for session types: it supports passing of abstractions (functions from names to processes) but does not support name-passing nor process recursion. Still, HO is very expressive: it can encode name-passing, recursion, and polyadic communication via type-preserving encodings that are fully-abstract with respect to contextual equivalence [19].

2.1 Syntax and Semantics

The syntax of names, variables, values, and HO processes is defined as follows:

$$n, m \;::=\; a, b \;\mid\; s, \overline{s} \qquad u, w \;::=\; n \;\mid\; x, y, z \qquad V, W \;::=\; x, y, z \;\mid\; \lambda x.\, P$$

$$P, Q \;::=\; u!\langle V\rangle.P \;\mid\; u?(x).P \;\mid\; u \triangleleft l.P \;\mid\; u \triangleright \{l_i : P_i\}_{i\in I} \;\mid\; V\, u \;\mid\; P \mid Q \;\mid\; (\nu\, n)P \;\mid\; \mathbf{0}$$

We use $a, b, c, \ldots$ to range over *shared names*, and $s, \overline{s}, \ldots$ to range over *session names*. Shared names are used for unrestricted, non-deterministic interactions; session names are used for linear, deterministic interactions. We write n, m to denote session or shared names, and assume that the sets of session and shared names are disjoint. The *dual* of n is denoted $\overline{n}$; we define $\overline{\overline{s}} = s$ and $\overline{a} = a$, i.e., duality is only relevant for session names. Variables are

[1] We summarize the content from [19, 20] that concerns HO; the notions and results given in [19, 20] are given for HOπ, a super-calculus of HO.

denoted with $x, y, z, \ldots$. An abstraction $\lambda x.\, P$ is a process P with parameter x. *Values* $V, W, \ldots$ include variables and abstractions, but not names. A tuple of variables $(x_1, \ldots, x_k)$ is denoted $\widetilde{x}$ (and similarly for names and values). We use ϵ to denote the empty tuple.

Processes $P, Q, \ldots$ include usual π-calculus output and input prefixes, denoted $u!\langle V\rangle.P$ and $u?(x).P$, respectively. Processes $u \triangleleft l.P$ and $u \triangleright \{l_i : P_i\}_{i\in I}$ are selecting and branching constructs, respectively, commonly used in session calculi to express deterministic choices [12]. Process $V\,u$ is the application which substitutes name u on abstraction V. Constructs for inaction $\mathbf{0}$, parallel composition $P_1 \mid P_2$, and name restriction $(\nu\, n)P$ are standard. HO lacks name-passing and recursion, but they are expressible in the language (see Exam. 2.1 below).

We sometimes omit trailing $\mathbf{0}$'s, so we may write, e.g., $u!\langle V\rangle$ instead of $u!\langle V\rangle.\mathbf{0}$. Also, we write $u!\langle\rangle.P$ and $u?().P$ whenever the exchanged value is not relevant (cf. Rem. 3.7).

Session name restriction $(\nu\, s)P$ simultaneously binds session names s and $\overline{s}$ in P. Functions $\mathtt{fv}(P)$, $\mathtt{fn}(P)$, and $\mathtt{fs}(P)$ denote, respectively, the sets of free variables, names, and session names in P, and are defined as expected. If $\mathtt{fv}(P) = \emptyset$, we call P *closed.* We write $P\{{}^{u}\!/\!{}_{y}\}$ (resp., $P\{{}^{V}\!/\!{}_{y}\}$) for the capture-avoiding substitution of name u (resp., value V) for y in process P. We identify processes up to consistent renaming of bound names, writing $\equiv_\alpha$ for this congruence. We shall rely on Barendregt's variable convention, which ensures that free and bound names are different in every mathematical context.

The operational semantics of HO is defined in terms of a *reduction relation*, denoted $\longrightarrow$. Reduction is closed under *structural congruence*, denoted $\equiv$, which is defined as the smallest congruence on processes such that:

$$P \mid \mathbf{0} \equiv P \quad P_1 \mid P_2 \equiv P_2 \mid P_1 \quad P_1 \mid (P_2 \mid P_3) \equiv (P_1 \mid P_2) \mid P_3 \quad (\nu\, n)\mathbf{0} \equiv \mathbf{0}$$

$$P \mid (\nu\, n)Q \equiv (\nu\, n)(P \mid Q)\ (n \notin \mathtt{fn}(P)) \quad P \equiv Q \text{ if } P \equiv_\alpha Q$$

We assume the expected extension of $\equiv$ to values V. The reduction relation expresses the behavior of processes; it is defined as follows:

$$\begin{array}{rll}
(\lambda x.\, P)\, u \longrightarrow P\{{}^{u}\!/\!{}_{x}\} & & [\text{App}] \\
n!\langle V\rangle.P \mid \overline{n}?(x).Q \longrightarrow P \mid Q\{{}^{V}\!/\!{}_{x}\} & & [\text{Pass}] \\
n \triangleleft l_j.Q \mid \overline{n} \triangleright \{l_i : P_i\}_{i\in I} \longrightarrow Q \mid P_j \quad (j \in I) & & [\text{Sel}] \\
P \longrightarrow P' \Rightarrow (\nu\, n)P \longrightarrow (\nu\, n)P' & & [\text{Res}] \\
P \longrightarrow P' \Rightarrow P \mid Q \longrightarrow P' \mid Q & & [\text{Par}] \\
P \equiv Q \longrightarrow Q' \equiv P' \Rightarrow P \longrightarrow P' & & [\text{Cong}]
\end{array}$$

Rule [App] defines name application. Rule [Pass] defines a shared or session interaction, depending on the nature of n. Rule [Sel] is the standard rule for labelled choice/selection. Other rules are standard π-calculus rules. We write $\longrightarrow^k$ for a k-step reduction, and $\longrightarrow^*$ for the reflexive, transitive closure of $\longrightarrow$.

We illustrate HO processes and their semantics by means of an example.

▶ **Example 2.1** (Encoding Name-Passing). HO lacks name-passing, and so the reduction

$$n!\langle m\rangle.P \mid \overline{n}?(x).Q \longrightarrow P \mid Q\{{}^{m}\!/\!{}_{x}\} \tag{1}$$

is not supported by the language. Still, as explained in [19], name-passing can be encoded in a fully-abstract way using abstraction-passing, by "packing" the name m in an abstraction. Let $[\![\cdot]\!]$ be the encoding defined as

$$[\![n!\langle m\rangle.P]\!] = n!\big\langle \lambda z.\ z?(x).(x\, m)\big\rangle.[\![P]\!]$$

$$[\![n?(x).Q]\!] = n?(y).(\nu\, s)(y\, s \mid \overline{s}!\langle \lambda x.\, [\![Q]\!]\rangle)$$

and as an homomorphism for the other constructs. Reduction (1) can be mimicked as

$$\begin{aligned}
\llbracket n!\langle m\rangle.P \mid \overline{n}?(x).Q\rrbracket = {} & n!\langle \lambda z.\ z?(x).(x\, m)\rangle.\llbracket P\rrbracket \mid n?(y).(\nu\, s)(y\, s \mid \overline{s}!\langle \lambda x.\, \llbracket Q\rrbracket\rangle) \\
\longrightarrow {} & \llbracket P\rrbracket \mid (\nu\, s)(\lambda z.\ z?(x).(x\, m)\, s \mid \overline{s}!\langle \lambda x.\, \llbracket Q\rrbracket\rangle) \\
\longrightarrow {} & \llbracket P\rrbracket \mid (\nu\, s)(s?(x).(x\, m) \mid \overline{s}!\langle \lambda x.\, \llbracket Q\rrbracket\rangle) \\
\longrightarrow {} & \llbracket P\rrbracket \mid (\lambda x.\, \llbracket Q\rrbracket)\, m \\
\longrightarrow {} & \llbracket P\rrbracket \mid \llbracket Q\rrbracket\{^{m}\!/\!_{x}\}
\end{aligned}$$

⌟

▶ Remark 2.2 (Polyadic Communication). HO as presented above allows only for *monadic communication*, i.e., the exchange of tuples of values with length 1. We will find it convenient to use HO with *polyadic communication*, i.e., the exchange of tuples of values $\widetilde{V}$, with length $k \geq 1$. In HO, polyadicity appears in session synchronizations and applications, but not in synchronizations on shared names. This entails having the following reduction rules:

$$\begin{aligned}
(\lambda \widetilde{x}.\, P)\, \widetilde{u} &\longrightarrow P\{^{\widetilde{u}}\!/\!_{\widetilde{x}}\} \\
s!\langle \widetilde{V}\rangle.P \mid \overline{s}?(\widetilde{x}).Q &\longrightarrow P \mid Q\{^{\widetilde{V}}\!/\!_{\widetilde{x}}\}
\end{aligned}$$

where the simultaneous substitutions $P\{^{\widetilde{u}}\!/\!_{\widetilde{x}}\}$ and $P\{^{\widetilde{V}}\!/\!_{\widetilde{x}}\}$ are as expected. This polyadic HO can be readily encoded into (monadic) HO [20]; for this reason, by a slight abuse of notation we will often write HO when we actually mean "polyadic HO".

2.2 Session Types for HO

We give essential definitions and properties for the session type system for HO, following [20].

▶ **Definition 2.3** (Session Types for HO [20]). *Let us write* $\diamond$ *to denote the process type. The syntax of types for* HO *is defined as follows:*

$$\begin{array}{lcl}
U & ::= & C \!\rightarrow\! \diamond \quad \mid \quad C \!\multimap\! \diamond \\
C & ::= & S \quad \mid \quad \langle U\rangle \\
S & ::= & \mathtt{end} \quad \mid \quad !\langle U\rangle; S \quad \mid \quad ?(U); S \quad \mid \quad \oplus\{l_i : S_i\}_{i\in I} \quad \mid \quad \&\{l_i : S_i\}_{i\in I} \quad \mid \quad \mu \mathtt{t}.S \quad \mid \quad \mathtt{t}
\end{array}$$

Value types U include $C \!\rightarrow\! \diamond$ and $C \!\multimap\! \diamond$, which denote *shared* and *linear* higher-order types, respectively. Shared channel types are denoted $\langle S\rangle$ and $\langle U\rangle$. Session types, denoted by S, follow the standard binary session type syntax [12]. Type end is the termination type. The *output type* $!\langle U\rangle; S$ first sends a value of type U and then follows the type described by S. Dually, $?(U); S$ denotes an *input type*. The *branching type* $\&\{l_i : S_i\}_{i\in I}$ and the *selection type* $\oplus\{l_i : S_i\}_{i\in I}$ are used to type the branching and selection constructs that define the labeled choice. We assume the *recursive type* $\mu\mathtt{t}.S$ is guarded, i.e., type $\mu\mathtt{t}.\mathtt{t}$ is not allowed.

In session types theories *duality* is a key notion: implementations derived from dual session types will respect their protocols at run-time, avoiding communication errors. Intuitively, duality is obtained by exchanging ! by ? (and vice versa) and $\oplus$ by & (and vice versa), including the fixed point construction. We write S dual T if session types S and T are dual according to this intuition; the formal definition is coinductive, and given in [20].

We consider shared, linear, and session *environments*, denoted Γ, Λ, and Δ, resp.:

$$\begin{array}{lcl}
\Gamma & ::= & \emptyset \quad \mid \quad \Gamma, x : C \!\rightarrow\! \diamond \quad \mid \quad \Gamma, u : \langle U\rangle \qquad \Lambda \ ::= \ \emptyset \quad \mid \quad \Lambda, x \!:\! C \!\multimap\! \diamond \\
\Delta & ::= & \emptyset \quad \mid \quad \Delta, u \!:\! S
\end{array}$$

$$\text{(Prom)}\ \frac{\Gamma; \emptyset; \emptyset \vdash V \triangleright C \multimap \diamond}{\Gamma; \emptyset; \emptyset \vdash V \triangleright C \rightarrow \diamond} \qquad \text{(EProm)}\ \frac{\Gamma; \Lambda, x : C \multimap \diamond; \Delta \vdash P \triangleright \diamond}{\Gamma, x : C \rightarrow \diamond; \Lambda; \Delta \vdash P \triangleright \diamond} \qquad \text{(Abs)}\ \frac{\Gamma; \Lambda; \Delta_1 \vdash P \triangleright \diamond \quad \Gamma; \emptyset; \Delta_2 \vdash x \triangleright C}{\Gamma \backslash x; \Lambda; \Delta_1 \backslash \Delta_2 \vdash \lambda x.\, P \triangleright C \multimap \diamond}$$

$$\text{(App)}\ \frac{\Gamma; \Lambda; \Delta_1 \vdash V \triangleright C \rightsquigarrow \diamond \qquad \rightsquigarrow \in \{\multimap, \rightarrow\} \quad \Gamma; \emptyset; \Delta_2 \vdash u \triangleright C}{\Gamma; \Lambda; \Delta_1, \Delta_2 \vdash V\, u \triangleright \diamond}$$

$$\text{(Send)}\ \frac{u : S \in \Delta_1, \Delta_2 \quad \Gamma; \Lambda_1; \Delta_1 \vdash P \triangleright \diamond \quad \Gamma; \Lambda_2; \Delta_2 \vdash V \triangleright U}{\Gamma; \Lambda_1, \Lambda_2; ((\Delta_1, \Delta_2) \setminus u : S), u :!\langle U \rangle; S \vdash u!\langle V \rangle.P \triangleright \diamond}$$

$$\text{(Rcv)}\ \frac{\Gamma; \Lambda_1; \Delta, u : S \vdash P \triangleright \diamond \quad \Gamma; \Lambda_2; \emptyset \vdash x \triangleright U}{\Gamma \backslash x; \Lambda_1 \backslash \Lambda_2; \Delta, u :?(U); S \vdash u?(x).P \triangleright \diamond}$$

$$\text{(Req)}\ \frac{\Gamma; \emptyset; \emptyset \vdash u \triangleright \langle U \rangle \quad \Gamma; \Lambda; \Delta_1 \vdash P \triangleright \diamond \quad \Gamma; \emptyset; \Delta_2 \vdash V \triangleright U}{\Gamma; \Lambda; \Delta_1, \Delta_2 \vdash u!\langle V \rangle.P \triangleright \diamond} \qquad \text{(Acc)}\ \frac{\Gamma; \emptyset; \emptyset \vdash u \triangleright \langle U \rangle \quad \Gamma; \Lambda_1; \Delta \vdash P \triangleright \diamond \quad \Gamma; \Lambda_2; \emptyset \vdash x \triangleright U}{\Gamma \backslash x; \Lambda_1 \backslash \Lambda_2; \Delta \vdash u?(x).P \triangleright \diamond}$$

Figure 1 Selected Typing Rules for HO. See [20] for a full account.

Γ maps variables and shared names to value types; Λ maps variables to linear higher-order types. Δ maps session names to session types. While Γ admits weakening, contraction, and exchange principles, both Λ and Δ are only subject to exchange. The domains of Γ, Λ, and Δ are assumed pairwise distinct. $\Delta_1 \cdot \Delta_2$ is the disjoint union of Δ_1 and Δ_2.

We write $\Gamma \backslash x$ to denote $\Gamma \backslash \{x : C\}$, i.e., the environment obtained from Γ by removing the assignment $x : C \rightarrow \diamond$, for some C. Notations $\Delta \backslash u$ and $\Gamma \backslash \widetilde{x}$ will have expected readings. With a slight abuse of notation, given a tuple of variables $\widetilde{x}$, we sometimes write $(\Gamma, \Delta)(\widetilde{x})$ to denote the tuple of types assigned to variables in $\widetilde{x}$.

The typing judgements for values V and processes P are denoted

$$\Gamma; \Lambda; \Delta \vdash V \triangleright U \qquad \text{and} \qquad \Gamma; \Lambda; \Delta \vdash P \triangleright \diamond$$

Fig. 1 shows selected typing rules; see [20] for a full account. The shared type $C \rightarrow \diamond$ is derived using Rule (Prom) only if the value has a linear type with an empty linear environment. Rule (EProm) allows us to freely use a shared type variable as linear. Abstraction values are typed with Rule (Abs). Application typing is governed by Rule (App): the type C of an application name u must match the type of the application variable x ($C \multimap \diamond$ or $C \rightarrow \diamond$). In Rule (Send), the type U of value V should appear as a prefix in the session type $!\langle U \rangle; S$ of u. Rule (Rcv) is its dual. Rules (Req) and (Acc) type interaction along shared names; the type of the sent/received object V (i.e., U) should match the type of the subject s ($\langle U \rangle$).

To state type soundness, we require two auxiliary definitions on session environments. First, a session environment Δ is *balanced* (written $\mathsf{balanced}(\Delta)$) if whenever $s : S_1, \overline{s} : S_2 \in \Delta$ then S_1 dual S_2. Second, we define the reduction relation $\longrightarrow$ on session environments as:

$$\begin{aligned} \Delta, s :!\langle U \rangle; S_1, \overline{s} :?(U); S_2 &\longrightarrow \Delta, s : S_1, \overline{s} : S_2 \\ \Delta, s : \oplus\{l_i : S_i\}_{i \in I}, \overline{s} : \&\{l_i : S_i'\}_{i \in I} &\longrightarrow \Delta, s : S_k, \overline{s} : S_k' \ (k \in I) \end{aligned}$$

▶ **Theorem 2.4** (Type Soundness [20]). Suppose $\Gamma; \emptyset; \Delta \vdash P \triangleright \diamond$ with $\mathsf{balanced}(\Delta)$. Then $P \longrightarrow P'$ implies $\Gamma; \emptyset; \Delta' \vdash P' \triangleright \diamond$ and $\Delta = \Delta'$ or $\Delta \longrightarrow \Delta'$ with $\mathsf{balanced}(\Delta')$.

▶ Remark 2.5 (Typed Polyadic Communication). When using processes with polyadic communication (cf. Rem. 2.2), we shall assume the extension of the type system defined in [20].

▶ **Notation 1** (Type Annotations). *We shall often annotate bound names and variables with their respective type. We will write, e.g.,* $(\nu\, s : S)P$ *to denote that the type of* s *in* P *is* S*. Similarly for values: we shall write* $\lambda u : C.\, P$*. Also, letting* $\rightsquigarrow \in \{\multimap, \rightarrow\}$*, we may write* $\lambda u : C^{\rightsquigarrow}.\, P$ *to denote that the value is linear (if* $\rightsquigarrow = \multimap$*) or shared (if* $\rightsquigarrow = \rightarrow$*). That is, we write* $\lambda u : C^{\rightsquigarrow}.\, P$ *if* $\Gamma; \Lambda; \Delta \vdash \lambda u.\, P \triangleright C \rightsquigarrow \diamond$*, for some* Γ*,* Λ*, and* Δ*.*

Having introduced the core session process language HO, we now move to detail its type-preserving decomposition into minimal session types.

3 Decomposing Session-Typed Processes

3.1 Key Ideas

Our goal is to transform an HO process P, typable with the session types in Def. 2.3, into another HO process, denoted $\mathcal{D}(P)$, typable using *minimal session types* (cf. Def. 3.1 below). By means of this transformation on processes, which we call a *decomposition*, the sequencing in session types for P is codified in $\mathcal{D}(P)$ by using additional actions. To ensure that this transformation on P is sound, we must also decompose its session types; our main result says that if P is well-typed under session types $S_1, \ldots, S_n$, then $\mathcal{D}(P)$ is typable using the minimal session types $\mathcal{G}(S_1), \ldots, \mathcal{G}(S_n)$, where $\mathcal{G}(\cdot)$ is a decomposition function that "slices" a session type (as in Def. 2.3) into a *list* of minimal session types (cf. Def. 3.2 below).

To define the decomposition $\mathcal{D}(P)$, in Def. 3.8 we rely on a *breakdown function* that translates P into a composition of *trios processes* (or simply *trios*). A trio is a process with exactly three nested prefixes. Roughly speaking, if P is a sequential process with k nested actions, then $\mathcal{D}(P)$ will contain k trios running in parallel: each trio in $\mathcal{D}(P)$ will enact exactly one prefix from P; the breakdown function must be carefully designed to ensure that trios trigger each other in such a way that $\mathcal{D}(P)$ preserves the prefix sequencing in P.

We borrow from Parrow [24] some useful terminology and notation on trios. The *context* of a trio is a tuple of variables $\widetilde{x}$, possibly empty, which makes variable bindings explicit. We use a reserved set of *propagator names* (or simply *propagators*), denoted with $c_k, c_{k+1}, \ldots$, to carry contexts and trigger the subsequent trio. A process with less than three sequential prefixes is called a *degenerate trio*. Also, a *leading trio* is the one that receives a context, performs an action, and triggers the next trio; a *control trio* only activates other trios.

The breakdown function works on both processes and values. The breakdown of process P is denoted by $\mathcal{B}^k_{\tilde{x}}(P)$, where k is the index for the propagators c_k, and $\widetilde{x}$ is the context to be received by the previous trio. Similarly, the breakdown of a value V is denoted by $\mathcal{V}^k_{\tilde{x}}(V)$.

We present the decomposition of well-typed HO processes (and its associated typability results) incrementally – this is useful to gradually illustrate our ideas and highlight the several ways in which our developments differ from Parrow's. In § 3.2, we consider a "core fragment" of HO, which contains output and input prefixes, application, restriction, parallel composition, and inaction. Hence, this fragment does not have labeled choice and recursion, nor recursive types. In § 3.3 we shall extend the decomposition functions with selection and branching; an extension that supports names with recursive types is presented in § 3.4.

3.2 The Core Fragment

We present our approach for a core fragment of HO. We start introducing some preliminary definitions, including the definition of breakdown function. Then we give our main result: Thm. 3.11 (Page 13) asserts that if process P is well-typed with standard session types, then $\mathcal{D}(P)$ is well-typed with minimal session types. This theorem relies crucially on Thm. 3.10 (Page 13), which specifies the way in which the breakdown function preserves typability.

3.2.1 Preliminaries

We start by introducing minimal session types as a fragment of Def. 2.3:

▶ **Definition 3.1** (Minimal Session Types). *The syntax of* minimal session types *for* HO *is defined as follows:*

$$
\begin{array}{rcl}
U & ::= & \widetilde{C} \rightarrow \diamond \mid \widetilde{C} \multimap \diamond \\
C & ::= & M \mid \langle U \rangle \\
M & ::= & \mathtt{end} \mid !\langle \widetilde{U} \rangle; \mathtt{end} \mid ?(\widetilde{U}); \mathtt{end}
\end{array}
$$

Clearly, this minimal type structure induces a reduced set of typable HO processes. We shall implicitly assume a type system for HO based on these minimal session types by considering the expected specializations of the notions, typing rules, and results summarized in § 2.2.

We now define how to “slice” a session type into a *list* of minimal session types.

▶ **Definition 3.2** (Decomposing Session Types). *Let S be a session type, U be a higher-order type, C be a name type, and $\langle U \rangle$ be a shared type, all as in Def. 2.3. The* type decomposition function *$\mathcal{G}(\cdot)$ is defined as:*

$$
\begin{aligned}
\mathcal{G}(!\langle U \rangle; S) &= \begin{cases} !\langle \mathcal{G}(U) \rangle; \mathtt{end} & \text{if } S = \mathtt{end} \\ !\langle \mathcal{G}(U) \rangle; \mathtt{end}, \mathcal{G}(S) & \text{otherwise} \end{cases} \\
\mathcal{G}(?(U); S) &= \begin{cases} ?(\mathcal{G}(U)); \mathtt{end} & \text{if } S = \mathtt{end} \\ ?(\mathcal{G}(U)); \mathtt{end}, \mathcal{G}(S) & \text{otherwise} \end{cases} \\
\mathcal{G}(\mathtt{end}) &= \mathtt{end} \\
\mathcal{G}(C \multimap \diamond) &= \mathcal{G}(C) \multimap \diamond \\
\mathcal{G}(C \rightarrow \diamond) &= \mathcal{G}(C) \rightarrow \diamond \\
\mathcal{G}(\langle U \rangle) &= \langle \mathcal{G}(U) \rangle \\
\mathcal{G}(S_1, \ldots, S_n) &= \mathcal{G}(S_1), \ldots, \mathcal{G}(S_n)
\end{aligned}
$$

Thus, intuitively, if a session type S contains k input/output actions, the list $\mathcal{G}(S)$ will contain k minimal session types. We write $|\mathcal{G}(S)|$ to denote the length of $\mathcal{G}(S)$.

▶ **Example 3.3.** Let $S = ?(\mathsf{Int}); ?(\mathsf{Int}); !\langle \mathsf{Bool} \rangle; \mathtt{end}$ be the session type given in § 1. Then $\mathcal{G}(S)$ is the list of minimal session types given by $?(\mathsf{Int}); \mathtt{end}, ?(\mathsf{Int}); \mathtt{end}, !\langle \mathsf{Bool} \rangle; \mathtt{end}$. ⌟

The breakdown function $\mathcal{B}^k_{\tilde{x}}(\,\cdot\,)$ will operate on processes with *indexed* names (cf. Def. 3.6). Indexes are relevant for session names: a name s_i will execute the i-th action in session s. For this reason, to extend the decomposition function $\mathcal{G}(\cdot)$ to typing environments, we consider names u_i in Γ and Δ. To define the decomposition of environments, we rely on the following notation. Given a tuple of names $\widetilde{s} = s_1, \ldots, s_n$ and a tuple of (session) types $\widetilde{S} = S_1, \ldots, S_n$ of the same length, we write $\widetilde{s} : \widetilde{S}$ to denote a list of typing assignments $s_1 : S_1, \ldots, s_n : S_n$.

▶ **Definition 3.4** (Decomposition of Environments). *Let Γ, Λ, and Δ be typing environments. We define $\mathcal{G}(\Gamma)$, $\mathcal{G}(\Lambda)$, and $\mathcal{G}(\Delta)$ inductively as follows:*

$$\begin{aligned}
\mathcal{G}(\Delta, u_i : S) &= \mathcal{G}(\Delta), (u_i, \ldots, u_{i+|\mathcal{G}(S)|-1}) : \mathcal{G}(S)\\
\mathcal{G}(\Gamma, u_i : \langle U\rangle) &= \mathcal{G}(\Gamma), u_i : \mathcal{G}(\langle U\rangle)\\
\mathcal{G}(\Gamma, x : U) &= \mathcal{G}(\Gamma), x : \mathcal{G}(U)\\
\mathcal{G}(\Lambda, x : U) &= \mathcal{G}(\Lambda), x : \mathcal{G}(U)\\
\mathcal{G}(\emptyset) &= \emptyset
\end{aligned}$$

In order to determine the required number of propagators $(c_k, c_{k+1}, \ldots)$ required in the breakdown of processes and values, we mutually define their *degree*:

▶ **Definition 3.5** (Degree of a Process and Value). *Let P be an* HO *process. The* degree *of P, denoted $|P|$, is inductively defined as follows:*

$$|P| = \begin{cases} |V| + |Q| + 1 & \textit{if } P = u_i!\langle V\rangle.Q\\ |Q| + 1 & \textit{if } P = u_i!\langle y\rangle.Q \textit{ or } P = u_i?(y).Q\\ |V| + 1 & \textit{if } P = V\, u_i\\ |P'| & \textit{if } P = (\nu\, s : S)P'\\ |Q| + |R| + 1 & \textit{if } P = Q \mid R\\ 1 & \textit{if } P = y\, u_i \textit{ or } P = \mathbf{0} \end{cases}$$

The degree *of a value V, denoted $|V|$, is defined as follows:*

$$|V| = \begin{cases} |P| & \textit{if } V = \lambda x : C^{\multimap}.\, P\\ 0 & \textit{if } V = \lambda x : C^{\rightarrow}.\, P \textit{ or } V = y \end{cases}$$

We define an auxiliary function that “initializes” the indices of a tuple of names.

▶ **Definition 3.6** (Name and Process Initialization). *Let $\widetilde{u} = (a, b, s, s', \ldots)$ be a finite tuple of names. We shall write* $\mathsf{init}(\widetilde{u})$ *to denote the tuple $(a_1, b_1, s_1, s'_1, \ldots)$. We will say that a process has been* initialized *if all of its names have some index.*

▶ Remark 3.7. Recall that we write ‘$c_k?()$’ and ‘$\overline{c_k}!\langle\rangle$’ to denote input and output prefixes in which the value communicated along c_k is not relevant. While ‘$c_k?()$’ stands for ‘$c_k?(x)$’, ‘$\overline{c_k}!\langle\rangle$’ stands for ‘$\overline{c_k}!\langle\lambda x.\, \mathbf{0}\rangle$’. Their corresponding minimal types are $?(\mathtt{end}\rightarrow\diamond);\mathtt{end}$ and $!\langle\mathtt{end}\rightarrow\diamond\rangle;\mathtt{end}$, which are denoted by $?(\cdot);\mathtt{end}$ and $!\langle\cdot\rangle;\mathtt{end}$, respectively.

Recall that P is *closed* if $\mathtt{fv}(P) = \emptyset$. We now define the *decomposition* of a process.

▶ **Definition 3.8** (Decomposing Processes). *Let P be a closed* HO *process such that $\widetilde{u} = \mathtt{fn}(P)$. The* decomposition *of P, denoted $\mathcal{D}(P)$, is defined as:*

$$\mathcal{D}(P) = (\nu\, \widetilde{c})\big(\overline{c_k}!\langle\rangle.\mathbf{0} \mid \mathcal{B}^k_\epsilon\big(P\sigma\big)\big)$$

where: $k > 0$; $\widetilde{c} = (c_k, \ldots, c_{k+|P|-1})$; $\sigma = \{{}^{\mathsf{init}(\widetilde{u})}/_{\widetilde{u}}\}$; and the breakdown function *$\mathcal{B}^k_{\widetilde{x}}(\,\cdot\,)$, where $\widetilde{x}$ is a tuple of variables, is defined inductively in § 3.2.2.*

The bulk of the decomposition of a process is given by the breakdown function, detailed next.

3.2.2 The Breakdown Function

Given a context $\widetilde{x}$ and a $k > 0$, the breakdown function $\mathcal{B}^k_{\tilde{x}}\big(\,\cdot\,\big)$ is defined on the structure of initialized processes, relying on the breakdown function on values $\mathcal{V}^k_{\tilde{y}}\big(\,\cdot\,\big)$. The definition relies on type information; we describe each of its cases next.

Output. The decomposition of $u_i!\langle V\rangle.Q$ is the most interesting case: an output prefix sends a value V (i.e., an abstracted process) that has to be broken down as well. We then have:

$$\mathcal{B}^k_{\tilde{x}}\big(u_i!\langle V\rangle.Q\big) = c_k?(\widetilde{x}).u_i!\big\langle \mathcal{V}^{k+1}_{\tilde{y}}\big(V\sigma\big)\big\rangle.\overline{c_{k+l+1}}!\langle\widetilde{z}\rangle \mid \mathcal{B}^{k+l+1}_{\tilde{z}}\big(Q\sigma\big)$$

Process $\mathcal{B}^k_{\tilde{x}}\big(u_i!\langle V\rangle.Q\big)$ consists of a leading trio that mimics an output action in parallel with the breakdown of the continuation Q. The context $\widetilde{x}$ must include the free variables of V and Q, denoted $\widetilde{y}$ and $\widetilde{z}$, respectively. These tuples are not necessarily disjoint: variables with shared types can appear free in both V and Q. The output object V is then broken down with parameters $\widetilde{y}$ and $k+1$; the latter serves to consistently generate propagators for the trios in the breakdown of V, denoted $\mathcal{V}^{k+1}_{\tilde{y}}\big(V\sigma\big)$ (see below for its definition). The substitution σ increments the index of session names; it is applied to both V and Q before they are broken down. We then distinguish two cases:

- If name u_i is linear (i.e., it has a session type) then its future occurrences are renamed into u_{i+1}, and $\sigma = \{^{u_{i+1}}/_{u_i}\}$;
- Otherwise, if u_i is not linear, then $\sigma = \{\}$.

Note that if u_i is linear then it appears either in V or Q and σ affects only one of them. The last prefix in the leading trio activates the breakdown of Q with its corresponding context $\widetilde{z}$. To avoid name conflicts with the propagators used in the breakdown of V, we use $\overline{c_{k+l+1}}$, with $l = |V|$ as a trigger for the continuation.

We remark that the same breakdown strategy is used when V stands for a variable y. Since by definition $|y| = 0$, $\mathcal{V}^k_{\tilde{y}}\big(y\big) = y$, and $y\sigma = y$, we have:

$$\mathcal{B}^k_{\tilde{x}}\big(u_i!\langle y\rangle.Q\big) = c_k?(\widetilde{x}).u_i!\langle y\rangle.\overline{c_{k+1}}!\langle\widetilde{z}\rangle \mid \mathcal{B}^{k+1}_{\tilde{z}}\big(Q\sigma\big)$$

We may notice that variable y is not propagated further if it does not appear in Q.

Input. The breakdown of an input prefix is defined as follows:

$$\mathcal{B}^k_{\tilde{x}}\big(u_i?(y).Q\big) = c_k?(\widetilde{x}).u_i?(y).\overline{c_{k+1}}!\langle\widetilde{x}'\rangle \mid \mathcal{B}^{k+1}_{\tilde{x}'}\big(Q\sigma\big)$$

where $\widetilde{x}' = \mathtt{fv}(Q)$. A leading trio mimics the input action and possibly extends the context with the received variable y. The substitution σ is defined as in the output case.

Application. The breakdown of $V\,u_i$ is as follows:

$$\mathcal{B}^k_{\tilde{x}}\big(V\,u_i\big) = c_k?(\widetilde{x}).\mathcal{V}^{k+1}_{\tilde{x}}\big(V\big)\,\widetilde{m}$$

A degenerate trio receives a context $\widetilde{x}$ and then proceeds with the application. We break down V with $\widetilde{x}$ as a context since these variables need to be propagated to the abstracted process. We use $k+1$ as a parameter to avoid name conflicts. Name u_i is decomposed into a tuple $\widetilde{m}$ using type information: if $u_i : C$ then $\widetilde{m} = (u_i, \ldots, u_{i+|\mathcal{G}(C)|-1})$ and so the length of $\widetilde{m}$ is $|\mathcal{G}(C)|$; each name in $\widetilde{m}$ will perform exactly one action. When V is a variable y, we have:

$$\mathcal{B}^k_{\tilde{x}}\big(y\,u_i\big) = c_k?(y).y\,\widetilde{m}$$

Notice that by construction $\widetilde{x} = y$.

Restriction. We define the breakdown of a restricted process as follows:

$$\mathcal{B}^k_{\widetilde{x}}\big((\nu\, s : C)P'\big) = (\nu\, \widetilde{s} : \mathcal{G}(C))\, \mathcal{B}^k_{\widetilde{x}}\big(P'\sigma\big)$$

By construction, $\widetilde{x} = \mathtt{fv}(P')$. Similarly as in the decomposition of u_i into $\widetilde{m}$ discussed above, we use the type C of s to obtain the tuple $\widetilde{s}$ of length $|\mathcal{G}(C)|$. We initialize the index of s in P' by applying the substitution σ. This substitution depends on C: if it is a shared type then $\sigma = \{{}^{s_1}\!/\!_{s}\}$; otherwise, if C is a session type, then $\sigma = \{{}^{s_1\overline{s_1}}\!/\!_{s\overline{s}}\}$.

Composition. The breakdown of a process $Q \mid R$ is as follows:

$$\mathcal{B}^k_{\widetilde{x}}\big(Q \mid R\big) = c_k?(\widetilde{x}).\overline{c_{k+1}}!\langle\widetilde{y}\rangle.\overline{c_{k+l+1}}!\langle\widetilde{z}\rangle \mid \mathcal{B}^{k+1}_{\widetilde{y}}\big(Q\big) \mid \mathcal{B}^{k+l+1}_{\widetilde{z}}\big(R\big)$$

A control trio triggers the breakdowns of Q and R; it does not mimic any action of the source process. The tuple $\widetilde{y} \subseteq \widetilde{x}$ (resp. $\widetilde{z} \subseteq \widetilde{x}$) collects the free variables in Q (resp. R). To avoid name conflicts, the trigger for the breakdown of R is $\overline{c_{k+l+1}}$, with $l = |Q|$.

Inaction. To breakdown $\mathbf{0}$, we define a degenerate trio with only one input prefix that receives a context that by construction will always be empty, i.e., $\widetilde{x} = \epsilon$:

$$\mathcal{B}^k_{\widetilde{x}}\big(\mathbf{0}\big) = c_k?().\mathbf{0}$$

Value. In defining the breakdown function for values we distinguish two main cases:

- If $V = \lambda y : C^{\leadsto}.\, P$, where $\leadsto \in \{\multimap, \rightarrow\}$, then we have:

  $$\mathcal{V}^k_{\widetilde{x}}\big(\lambda y : C^{\leadsto}.\, P\big) = \lambda \widetilde{y} : \mathcal{G}(C)^{\leadsto}.\, (\nu\, \widetilde{c})\big(\overline{c_k}!\langle\widetilde{x}\rangle \mid \mathcal{B}^k_{\widetilde{x}}\big(P\{{}^{y_1}\!/\!_{y}\}\big)\big)$$

  We use type C to decompose y into the tuple $\widetilde{y}$. We abstract over $\widetilde{y}$; the body of the abstraction is the composition of a control trio and the breakdown of P, with name index initialized with the substitution $\{{}^{y_1}\!/\!_{y}\}$. If $\leadsto = \rightarrow$ then we restrict the propagators $\widetilde{c} = (c_k, \ldots, c_{k+|P|-1})$: this enables us to type the value in a shared environment. When $\leadsto = \multimap$ we do not have to restrict the propagators, and $\widetilde{c} = \epsilon$.
- If $V = y$, then the breakdown function is the identity: $\mathcal{V}^k_{\widetilde{x}}\big(y\big) = y$.

Tab. 1 summarizes the definition of the breakdown, spelling out the side conditions involved. We illustrate it by means of an example:

▶ **Example 3.9** (Breaking Down Name-Passing)**.** Consider the following process P, in which a channel m is passed, through which a Boolean value is sent back:

$$P = (\nu\, u)(u!\langle m\rangle.\overline{m}?(b).\mathbf{0} \mid \overline{u}?(x).x!\langle \mathsf{true}\rangle.\mathbf{0})$$

P is not an HO process as it features name-passing. We then use the encoding described in Exam. 2.1 to construct its encoding into HO. We thus obtain $[\![P]\!] = (\nu\, u)(Q \mid R)$, where

$$\begin{aligned} Q &= u!\langle V\rangle.\overline{m}?(y).(\nu\, s)(y\, s \mid \overline{s}!\langle\lambda b.\, \mathbf{0}\rangle.\mathbf{0}) & V &= \lambda z.\, z?(x).(x\, m) \\ R &= \overline{u}?(y).(\nu\, s)(y\, s \mid \overline{s}!\langle W\rangle.\mathbf{0}) & W &= \lambda x.\, x!\langle W'\rangle.\mathbf{0} \text{ with } W' = \lambda z.\, z?(x).(x\, \mathsf{true}) \end{aligned}$$

By Exam. 2.1, we know that $[\![\cdot]\!]$ requires exactly four reduction steps to mimic a name-passing synchronization. We show here part of the reduction chain of $[\![P]\!]$:

$$[\![P]\!] \longrightarrow^4 [\![\overline{m}?(b).\mathbf{0} \mid m!\langle\mathsf{true}\rangle.\mathbf{0}]\!] \longrightarrow^4 \mathbf{0} \tag{2}$$

We will now investigate the decomposition of $[\![P]\!]$ and its reduction chain. First, we use Def. 3.5 to compute $|V| = |W'| = 2$, and so $|W| = 4$. Then $|Q| = |y\, s \mid \overline{s}!\langle\lambda b.\, \mathbf{0}\rangle.\mathbf{0}| + |V| + 2 = 9$, and similarly, $|R| = 9$. Therefore, $|[\![P]\!]| = 19$. Following Def. 3.8, we see that $\sigma = \{{}^{m_1\overline{m_1}}\!/\!_{m\overline{m}}\}$, which we silently apply. Using $k = 1$, we then have the decomposition shown in Tab. 2.

Table 1 The breakdown function for processes and values (core fragment).

P	$\mathcal{B}^{k}_{\tilde{x}}(P)$	
$u_i!\langle V\rangle.Q$	$c_k?(\widetilde{x}).u_i!\langle \mathcal{V}^{k+1}_{\tilde{y}}(V\sigma)\rangle.\overline{c_{k+l+1}}!\langle\widetilde{z}\rangle \mid$ $\mathcal{B}^{k+l+1}_{\tilde{z}}(Q\sigma)$	$\widetilde{y} = \texttt{fv}(V),\ \widetilde{z} = \texttt{fv}(Q)$ $l = \lvert V\rvert$ $\sigma = \begin{cases} \{{}^{u_{i+1}}/_{u_i}\} & \text{if } u_i : S \\ \{\} & \text{otherwise} \end{cases}$
$u_i?(y).Q$	$c_k?(\widetilde{x}).u_i?(y).\overline{c_{k+1}}!\langle\widetilde{x}'\rangle \mid \mathcal{B}^{k+1}_{\tilde{x}'}(Q\sigma)$	$\widetilde{x}' = \texttt{fv}(Q)$ $\sigma = \begin{cases} \{{}^{u_{i+1}}/_{u_i}\} & \text{if } u_i : S \\ \{\} & \text{otherwise} \end{cases}$
$V\ u_i$	$c_k?(\widetilde{x}).\mathcal{V}^{k+1}_{\tilde{x}}(V)\ \widetilde{m}$	$u_i : C$ $\widetilde{x} = \texttt{fv}(V)$ $\widetilde{m} = (u_i, \ldots, u_{i+\lvert\mathcal{G}(C)\rvert-1})$
$(\nu\, s : C)P'$	$(\nu\,\widetilde{s} : \mathcal{G}(C))\,\mathcal{B}^{k}_{\tilde{x}}(P'\sigma)$	$\widetilde{x} = \texttt{fv}(P')$ $\widetilde{s} = (s_1, \ldots, s_{\lvert\mathcal{G}(C)\rvert})$ $\sigma = \begin{cases} \{{}^{s_1\overline{s_1}}/_{s\overline{s}}\} & \text{if } C = S \\ \{{}^{s_1}/_{s}\} & \text{if } C = \langle U\rangle \end{cases}$
$Q \mid R$	$c_k?(\widetilde{x}).\overline{c_{k+1}}!\langle\widetilde{y}\rangle.\overline{c_{k+l+1}}!\langle\widetilde{z}\rangle \mid$ $\mathcal{B}^{k+1}_{\tilde{y}}(Q) \mid \mathcal{B}^{k+l+1}_{\tilde{z}}(R)$	$\widetilde{y} = \texttt{fv}(Q)$ $\widetilde{z} = \texttt{fv}(R)$ $l = \lvert Q\rvert$
$\mathbf{0}$	$c_k?().\mathbf{0}$	
V	$\mathcal{V}^{k}_{\tilde{x}}(V)$	
y	y	
$\lambda u : C^{\leadsto}.\, P$	$\lambda\widetilde{y} : \mathcal{G}(C)^{\leadsto}.\,(\nu\,\widetilde{c})\big(\overline{c_k}!\langle\widetilde{x}\rangle \mid$ $\mathcal{B}^{k}_{\tilde{x}}(P\{{}^{y_1}/_{y}\})\big)$ $\widetilde{c} = \begin{cases} \epsilon & \text{if } \leadsto = \multimap \\ (c_k, \ldots, c_{k+\lvert P\rvert-1}) & \text{if } \leadsto = \rightarrow \end{cases}$	$\widetilde{x} = \texttt{fv}(V)$ $\widetilde{y} = (y_1, \ldots, y_{\lvert\mathcal{G}(C)\rvert})$

Tab. 2 we have omitted substitutions that have no effect and trailing $\mathbf{0}$s. The first interesting process appears after synchronizations on c_1, c_2, and c_{11}. At that point, the process will be ready to mimic the first action that is performed by $[\![P]\!]$, i.e., u_1 will send $\mathcal{V}^{3}_{\epsilon}(V)$, the breakdown of V. Next, c_{12}, c_{13}, and c_{14} will synchronize, and $\mathcal{V}^{3}_{\epsilon}(V)$ is passed further along, until s_1 is ready to be applied to it in the breakdown of R. At this point, we know that $[\![P]\!] \longrightarrow^{7} (\nu\,\widetilde{c})P'$, where $\widetilde{c} = (c_3, \ldots, c_{10}, c_{15}, \ldots, c_{19})$, and

$$
\begin{aligned}
P' = {} & \overline{c_5}!\langle\rangle.\mathbf{0} \mid c_5?().\overline{m_1}?(y).\overline{c_6}!\langle y\rangle.\mathbf{0} \\
& \mid (\nu\, s_1)(c_6?(y).\overline{c_7}!\langle y\rangle.\overline{c_8}!\langle\rangle.\mathbf{0} \mid c_7?(y).y\, s_1 \mid c_8?().\overline{s_1}!\langle\mathcal{V}^{9}_{\epsilon}(\lambda b.\,\mathbf{0})\rangle.\overline{c_{10}}!\langle\rangle.\mathbf{0} \mid c_{10}?().\mathbf{0}) \\
& \mid (\nu\, s_1)\big(\mathcal{V}^{3}_{\epsilon}(V)\ s_1 \mid \overline{s_1}!\langle\mathcal{V}^{15}_{\epsilon}(W)\rangle.\overline{c_{19}}!\langle\rangle.\mathbf{0} \mid c_{19}?().\mathbf{0}\big)
\end{aligned}
$$

After s_1 is applied, the trio guarded by c_3 will be activated, where z_1 has been substituted by s_1. Then $\overline{s_1}$ and s_1 will synchronize, and the breakdown of W is passed along. Then c_4 and c_{19} synchronize, and now m_1 is ready to be applied to $\mathcal{V}^{15}_{\epsilon}(W)$, which was the input

Table 2 The process decomposition discussed in Exam. 3.9.

$$\begin{aligned}
\mathcal{D}(\llbracket P \rrbracket) &= (\nu\, c_1, \ldots, c_{19})\Big(\overline{c_1}!\langle\rangle \mid (\nu\, u_1)\big(c_1?().\overline{c_2}!\langle\rangle.\overline{c_{11}}!\langle\rangle \mid \mathcal{B}^{2}_{\epsilon}\big(Q\big) \mid \mathcal{B}^{11}_{\epsilon}\big(R\big)\big)\Big)\\
\mathcal{B}^{2}_{\epsilon}\big(Q\big) &= c_2?().u_1!\langle \mathcal{V}^{3}_{\epsilon}\big(V\big)\rangle.\overline{c_5}!\langle\rangle \mid c_5?().\overline{m_1}?(y).\overline{c_6}!\langle y\rangle \mid\\
&\qquad (\nu\, s_1)(c_6?(y).\overline{c_7}!\langle y\rangle.\overline{c_8}!\langle\rangle \mid c_7?(y).(y\, s_1) \mid c_8?().\overline{s_1}!\langle \mathcal{V}^{9}_{\epsilon}\big(\lambda b.\,\mathbf{0}\big)\rangle.\overline{c_{10}}!\langle\rangle \mid c_{10}?())\\
\mathcal{B}^{11}_{\epsilon}\big(R\big) &= c_{11}?().\overline{u_1}?(y).\overline{c_{12}}!\langle y\rangle \mid\\
&\qquad (\nu\, s_1)\big(c_{12}?(y).\overline{c_{13}}!\langle y\rangle.\overline{c_{14}}!\langle\rangle \mid c_{13}?(y).(y\, s_1) \mid c_{14}?().\overline{s_1}!\langle \mathcal{V}^{15}_{\epsilon}\big(W\big)\rangle.\overline{c_{19}}!\langle\rangle \mid c_{19}?()\big)\\
\mathcal{V}^{3}_{\epsilon}\big(V\big) &= \lambda z_1.\,(\overline{c_3}!\langle\rangle \mid c_3?().z_1?(x).\overline{c_4}!\langle x\rangle \mid c_4?(x).(x\, m_1))\\
\mathcal{V}^{9}_{\epsilon}\big(\lambda b.\,\mathbf{0}\big) &= \lambda b_1.\,(\overline{c_9}!\langle\rangle \mid c_9?())\\
\mathcal{V}^{15}_{\epsilon}\big(W\big) &= \lambda x_1.\,(\overline{c_{15}}!\langle\rangle \mid c_{15}?().x_1!\langle \mathcal{V}^{16}_{\epsilon}\big(W'\big)\rangle.\overline{c_{18}}!\langle\rangle \mid c_{18}?())\\
\mathcal{V}^{16}_{\epsilon}\big(W'\big) &= \lambda z_1.\,(\overline{c_{16}}!\langle\rangle \mid c_{16}?().z_1?(x).\overline{c_{17}}!\langle x\rangle \mid c_{17}?(x).(x\, \mathsf{true}))
\end{aligned}$$

for c_4 in the breakdown of V. After this application, c_5 and c_{15} can synchronize with their duals, and we know that $(\nu\,\widetilde{c})P' \longrightarrow^{8} (\nu\,\widetilde{c}')P''$, where $\widetilde{c}' = (c_6, \ldots, c_{10}, c_{16}, c_{17}, c_{18})$, and

$$\begin{aligned}
P'' = {} & \overline{m_1}?(y).\overline{c_6}!\langle y\rangle.\mathbf{0} \mid m_1!\langle \mathcal{V}^{15}_{\epsilon}\big(W'\big)\rangle.\overline{c_{17}}!\langle\rangle.\mathbf{0} \mid c_{17}?().\mathbf{0}\\
& \mid (\nu\, s_1)(c_6?(y).\overline{c_7}!\langle y\rangle.\overline{c_8}!\langle\rangle.\mathbf{0} \mid c_7?(y).y\, s_1 \mid c_8?().\overline{s_1}!\langle \mathcal{V}^{9}_{\epsilon}\big(\lambda b.\,\mathbf{0}\big)\rangle.\overline{c_{10}}!\langle\rangle.\mathbf{0} \mid c_{10}?().\mathbf{0})
\end{aligned}$$

Remarkably, P'' is standing by to mimic the encoded exchange of value $\mathtt{true}$. Indeed, the decomposition of the four-step reduced process in (2) will reduce in three steps to a process that is equal (up to $\equiv_\alpha$) to the process we obtained here. This strongly suggests a tight operational correspondence between a process and its decomposition. ⌟

We may now state our technical results:

▶ **Theorem 3.10** (Typability of Breakdown). *Let P be an initialized process and V be a value.*

1. *If $\Gamma; \Lambda; \Delta \vdash P \triangleright \diamond$ then*

$$\mathcal{G}(\Gamma_1); \emptyset; \mathcal{G}(\Delta), \Theta \vdash \mathcal{B}^{k}_{\tilde{x}}\big(P\big) \triangleright \diamond \quad (k > 0)$$

where: $\widetilde{x} = \mathtt{fv}(P)$; $\Gamma_1 = \Gamma \setminus \widetilde{x}$; and $\mathsf{balanced}(\Theta)$ *with $dom(\Theta) = \{c_k, c_{k+1}, \ldots, c_{k+|P|-1}\} \cup \{\overline{c_{k+1}}, \ldots, \overline{c_{k+|P|-1}}\}$ and $\Theta(c_k) = ?(\widetilde{M}); \mathtt{end}$, where $\widetilde{M} = (\mathcal{G}(\Gamma), \mathcal{G}(\Lambda))(\widetilde{x})$.*

2. *If $\Gamma; \Lambda; \Delta \vdash V \triangleright C {-\!\!\circ} \diamond$ then*

$$\mathcal{G}(\Gamma); \mathcal{G}(\Lambda); \mathcal{G}(\Delta), \Theta \vdash \mathcal{V}^{k}_{\tilde{x}}\big(V\big) \triangleright \mathcal{G}(C) {-\!\!\circ} \diamond \quad (k > 0)$$

where: $\widetilde{x} = \mathtt{fv}(V)$; and $\mathsf{balanced}(\Theta)$ *with $dom(\Theta) = \{c_k, \ldots, c_{k+|V|-1}\} \cup \{\overline{c_k}, \ldots, \overline{c_{k+|V|-1}}\}$ and $\Theta(c_k) = ?(\widetilde{M}); \mathtt{end}$ and $\Theta(\overline{c_k}) = !\langle\widetilde{M}\rangle; \mathtt{end}$, where $\widetilde{M} = (\mathcal{G}(\Gamma), \mathcal{G}(\Lambda))(\widetilde{x})$.*

3. *If $\Gamma; \emptyset; \emptyset \vdash V \triangleright C \rightarrow \diamond$ then $\mathcal{G}(\Gamma); \emptyset; \emptyset \vdash \mathcal{V}^{k}_{\tilde{x}}\big(V\big) \triangleright \mathcal{G}(C) \rightarrow \diamond$, where $\widetilde{x} = \mathtt{fv}(V)$ and $k > 0$.*

Proof. By mutual induction on the structure of P and V. ◀

Using the above theorem, we can prove our main result:

▶ **Theorem 3.11** (Typability of the Decomposition). *Let P be a closed* HO *process with $\widetilde{u} = \mathtt{fn}(P)$. If $\Gamma; \emptyset; \Delta \vdash P \triangleright \diamond$ then $\mathcal{G}(\Gamma\sigma); \emptyset; \mathcal{G}(\Delta\sigma) \vdash \mathcal{D}(P) \triangleright \diamond$, where $\sigma = \{^{\mathsf{init}(\widetilde{u})}/_{\widetilde{u}}\}$.*

Proof. Direct from the definitions, using Thm. 3.10. ◀

3.3 Extensions (I): Select and Branching

We now show how to extend the decomposition to handle select and branch processes, which implement labeled (deterministic) choice in session protocols, as well as their corresponding session types. As we will see, in formalizing this extension we shall appeal to the expressive power of abstraction-passing. We start by extending the syntax of minimal session types:

▶ **Definition 3.12** (Minimal Session Types (with Labeled Choice)). *The syntax of* minimal session types *for* HO *is defined as follows:*

$$M \ ::= \ \mathtt{end} \ \mid \ !\langle\widetilde{U}\rangle;\mathtt{end} \ \mid \ ?(\widetilde{U});\mathtt{end} \ \mid \ \oplus\{l_i : M_i\}_{i\in I} \ \mid \ \&\{l_i : M_i\}_{i\in I}$$

where U and C are defined as in Def. 3.1.

We may then extend Def. 3.2 to branch and select types as follows:

▶ **Definition 3.13** (Decomposing Session Types, Extended (I)). *The decomposition function on types as given in Def. 3.2 is extended as follows:*

$$\mathcal{G}(\&\{l_i : S_i\}_{i\in I}) = \&\{l_i :!\langle\mathcal{G}(S_i)\multimap\diamond\rangle;\mathtt{end}\}_{i\in I}$$
$$\mathcal{G}(\oplus\{l_i : S_i\}_{i\in I}) = \oplus\{l_i :?(\mathcal{G}(\overline{S_i})\multimap\diamond);\mathtt{end}\}_{i\in I}$$

The above definition for decomposed types already suggests our strategy to breakdown branching and selection processes: we will exploit abstraction-passing to exchange one abstraction per each branch of the labeled choice. This intuition will become clearer shortly.

We now extend the definition of the degree of a process/value (cf. Def. 3.5) to account for branch and select processes:

▶ **Definition 3.14** (Degree of a Process, Extended). *The* degree *of a process P, denoted $|P|$, is as given in Def. 3.5, extended as follows:*

$$|P| = \begin{cases} 1 & \textit{if } P = u_i \triangleright \{l_j : P_j\}_{j\in I} \\ |P'| + 2 & \textit{if } P = u_i \triangleleft l_j.P' \end{cases}$$

The definition of process decomposition (cf. Def. 3.8) does not require modifications; it relies on the extended definition of the breakdown function for processes $\mathcal{B}^k_{\tilde{x}}\big(\cdot\big)$ that combines the definitions in Tab. 1 with those in Tab. 3 (see below). The breakdown of values $\mathcal{V}^k_{\tilde{x}}\big(\cdot\big)$ is as before, and relies on the extended definition of $\mathcal{B}^k_{\tilde{x}}\big(\cdot\big)$.

We now present and describe the breakdown of branching and selection processes:
Branching. The breakdown of a branching process $u_i \triangleright \{l_j : P_j\}_{j\in I}$ is as follows:

$$\mathcal{B}^k_{\tilde{x}}\big(u_i \triangleright \{l_j : P_j\}_{j\in I}\big) = c_k?(\widetilde{x}).u_i \triangleright \{l_j : u_i!\big\langle N_{u,j}\big\rangle\}_{j\in I}$$
$$\text{where} \ \ N_{u,j} = \lambda\widetilde{y^u_j} : \mathcal{G}(S_j).\,(\nu\,\widetilde{c}_j)\big(\overline{c_{k+1}}!\langle\widetilde{x}\rangle \mid \mathcal{B}^{k+1}_{\tilde{x}}\big(P_j\{y^u_1/u_i\}\big)\big)$$

The first prefix receives the context $\widetilde{x}$. The next two prefixes are along u_i: the first one mimics the branching action of P, whereas the second outputs an abstraction $N_{u,j}$. This output does not have a counterpart in P; it is meant to synchronize with an input in the breakdown of the corresponding selection process (see below). $N_{u,j}$ encapsulates the breakdown of subprocess P_j. It has the same structure as the breakdown of a value $\lambda y : C^{\rightarrow}.\,P$ in Tab. 1: it is a composition of a control trio and the breakdown of P_j; the generated propagators, denoted $\widetilde{c}_j$, are restricted. We use types to define $N_{u,j}$: we assume S_j is the session type of u_i in the j-th branch of P. We abstract over $\widetilde{y^u_j} = (y^u_1, \ldots, y^u_{|\mathcal{G}(S_j)|})$. We substitute u_i with y^u_1 in P_j before breaking it down: this way, u_i is decomposed and bound by abstraction.

Table 3 The breakdown function for processes (extension with selection and branching).

$\mathcal{B}^k_{\widetilde{x}}\big(u_i \triangleright \{l_j{:}P_j\}_{j\in I}\big)$	
$c_k?(\widetilde{x}).u_i \triangleright \{l_j : u_i!\langle N_{u,j}\rangle\}_{j\in I}$ where: $N_{u,j} = \lambda \widetilde{y}^u_j : \mathcal{G}(S_j).\, (\nu\, \widetilde{c}_j)\big(\overline{c_{k+1}}!\langle\widetilde{x}\rangle \mid \mathcal{B}^{k+1}_{\widetilde{x}}\big(P_j\{y^u_1/u_i\}\big)\big)$	$\widetilde{y}^u_j = (y^u_1, \dots, y^u_{\lvert\mathcal{G}(S_j)\rvert})$ $\widetilde{c}_j = (c_{k+1}, \dots, c_{k+\lvert P_j\rvert})$
$\mathcal{B}^k_{\widetilde{x}}\big(u_i \triangleleft l_j.P'\big)$	
$c_k?(\widetilde{x}).\overline{c_{k+1}}!\langle M_j\rangle \mid$ $(\nu\, \widetilde{u} : \mathcal{G}(S_j))\big(c_{k+1}?(y).y\, \widetilde{\overline{u}} \mid \mathcal{B}^{k+2}_{\widetilde{x}}\big(P'\{u_{i+1}/u_i\}\big)\big)$ where: $M_j = \lambda \widetilde{y}.\, u_i \triangleleft l_j.u_i?(z).\overline{c_{k+2}}!\langle\widetilde{x}\rangle.z\, \widetilde{y}$	$\widetilde{y} = (y_1, \dots, y_{\lvert\mathcal{G}(S_j)\rvert})$ $\widetilde{u} = (u_{i+1}, \dots, u_{i+\lvert\mathcal{G}(S_j)\rvert})$ $\widetilde{\overline{u}} = (\overline{u_{i+1}}, \dots, \overline{u_{i+\lvert\mathcal{G}(S_j)\rvert}})$

Selection. The breakdown of a selection process $u_i \triangleleft l_j.P'$ is as follows:

$$\mathcal{B}^k_{\widetilde{x}}\big(u_i \triangleleft l_j.P'\big) = c_k?(\widetilde{x}).\overline{c_{k+1}}!\langle M_j\rangle \mid (\nu\, \widetilde{u} : \mathcal{G}(S_j))(c_{k+1}?(y).y\, \widetilde{\overline{u}} \mid \mathcal{B}^{k+2}_{\widetilde{x}}\big(P'\{u_{i+1}/u_i\}\big))$$
$$\text{where } M_j = \lambda \widetilde{y}.\, u_i \triangleleft l_j.u_i?(z).\overline{c_{k+2}}!\langle\widetilde{x}\rangle.z\, \widetilde{y}$$

After receiving the context $\widetilde{x}$, the abstraction M_j is sent along $\overline{c_{k+1}}$, and is to be received by the second subprocess in the composition. This sequence of actions allows us to preserve the intended trio structure. We use S_j, the type of u_i in P', to construct a corresponding tuple $\widetilde{u}$, with type $\mathcal{G}(S_j)$. We apply the abstraction M_j, received along c_{k+1}, to $\widetilde{\overline{u}}$ (the duals of $\widetilde{u}$). At this point, the selection action in P can be mimicked, and so label l_j is chosen from the breakdown of a corresponding branching process. As discussed above, such a breakdown will send an abstraction $N_{u,j}$ with type $\overline{S_j} \multimap \diamond$, which encapsulates the breakdown of the chosen subprocess. Before running $N_{u,j}$ with names $\widetilde{\overline{u}}$, we trigger the breakdown of P' with an appropriate substitution.

Summing up, our strategy for breaking down labeled choices exploits higher-order concurrency to uniformly handle the fact that the subprocesses of a branching process have a different session type and degree. Interestingly, it follows the intuition that branching and selection correspond to a form of output and input actions involving labels, respectively.

▶ Remark 3.15. Theorems 3.10 and 3.11 hold also for the extension with selection and branching .

▶ **Example 3.16** (Breaking down Selection and Branching)**.** We illustrate the breaking down of selection and branching processes by considering a basic mathematical server Q that allows clients to add or subtract two integers. The server contains two branches: one sends an abstraction V_+ that implements integer addition, the other sends an abstraction V_- implementing subtraction. A client R selects the first option to add integers 16 and 26:

$$Q \triangleq u \triangleright \{\mathsf{add} : u!\langle V_+\rangle.\mathbf{0}, \mathsf{sub} : u!\langle V_-\rangle.\mathbf{0}\}$$
$$R \triangleq \overline{u} \triangleleft \mathsf{add}.\overline{u}?(x).x\, (16{,}26)$$

The composition $P \triangleq (\nu\, u)(Q \mid R)$ reduces in two steps to a process $V_+\, (16{,}26)$:

$$P \longrightarrow (\nu\, u)(u!\langle V_+\rangle.\mathbf{0} \mid \overline{u}?(x).x\, (16{,}26)) \longrightarrow V_+\, (16{,}26) \tag{3}$$

We will investigate the decomposition of P, and its reduction chain. First, by Def. 3.5 and Def. 3.14, we have: $|Q| = 1$, $|R| = 4$, and $|P| = 6$. Following the extension of Def. 3.8, using $k = 1$, and observing that $\sigma_1 = \{\}$, we obtain:

$$\mathcal{D}(P) = (\nu\, c_1 \dots c_6)\big(\overline{c_1}!\langle\rangle.\mathbf{0} \mid (\nu\, u_1)(c_1?().\overline{c_2}!\langle\rangle.\overline{c_3}!\langle\rangle.\mathbf{0} \mid \mathcal{B}^2_{\epsilon}\big(Q\sigma_2\big) \mid \mathcal{B}^3_{\epsilon}\big(R\sigma_2\big)\big)\big)$$

where $\sigma_2 = \{{}^{u_1\overline{u_1}}/_{u\overline{u}}\}$. The breakdown of Q is obtained by applying the first rule in Tab. 3:

$$\begin{aligned}\mathcal{B}^2_\epsilon\big(Q\sigma_2\big) = c_2?().u_1 \triangleright \{&\mathsf{add} : (\nu\, c_3c_4)u_1!\big\langle \lambda y_1.\, \overline{c_3}!\langle\rangle.\mathbf{0} \mid \mathcal{B}^3_\epsilon\big(u_1!\langle V_+\rangle.\mathbf{0}\{{}^{y_1}/_{u_1}\}\big)\big\rangle.\mathbf{0},\\ &\mathsf{sub} : (\nu\, c_3c_4)u_1!\big\langle \lambda y_1.\, \overline{c_3}!\langle\rangle.\mathbf{0} \mid \mathcal{B}^3_\epsilon\big(u_1!\langle V_-\rangle.\mathbf{0}\{{}^{y_1}/_{u_1}\}\big)\big\rangle.\mathbf{0}\}\end{aligned}$$

The breakdown of R is obtained by applying the second rule in Tab. 3:

$$\begin{aligned}\mathcal{B}^3_\epsilon\big(R\sigma_2\big) = {}& c_3?().\overline{c_4}!\langle \lambda y_1.\, \overline{u_1} \triangleleft \mathsf{add}.\overline{u_1}?(z).\overline{c_5}!\langle\rangle.z\, y_1\rangle.\mathbf{0}\\ &\mid (\nu\, u_2)(c_4?(y).y\,\overline{u_2} \mid \mathcal{B}^5_\epsilon\big(\overline{u_1}?(x).x\,(16{,}26)\{{}^{u_2}/_{\overline{u_1}}\}\big)\big)\end{aligned}$$

We will now follow the chain of reductions of the process $\mathcal{D}(P)$. First, c_1, c_2, and c_3 will synchronize, after which c_4 will pass the abstraction. Let $\mathcal{D}(P) \longrightarrow^4 P'$, then we know:

$$\begin{aligned}P' = (\nu\, c_5c_6)(\nu\, u_1)\big(u_1 \triangleright \{&\mathsf{add} : (\nu\, c_3c_4)u_1!\big\langle \lambda y_1.\, \overline{c_3}!\langle\rangle.\mathbf{0} \mid \mathcal{B}^3_\epsilon\big(y_1!\langle V_+\rangle.\mathbf{0}\big)\big\rangle.\mathbf{0},\\ &\mathsf{sub} : (\nu\, c_3c_4)u_1!\big\langle \lambda y_1.\, \overline{c_3}!\langle\rangle.\mathbf{0} \mid \mathcal{B}^3_\epsilon\big(y_1!\langle V_-\rangle.\mathbf{0}\big)\big\rangle.\mathbf{0}\}\\ &\mid (\nu\, u_2)(\lambda y_1.\, \overline{u_1} \triangleleft \mathsf{add}.\overline{u_1}?(z).\overline{c_5}!\langle\rangle.z\, y_1\, \overline{u_2} \mid \mathcal{B}^5_\epsilon\big(u_2?(x).x\,(16{,}26)\big)\big)\big)\end{aligned}$$

In P', $\overline{u_2}$ will be applied to the abstraction with variable y_1. After that, the choice for the process labeled by add is made. Process P' will reduce further as $P' \longrightarrow^2 P'' \longrightarrow^2 P'''$, where:

$$\begin{aligned}P'' &= (\nu\, c_5c_6)(\nu\, u_1)\big((\nu\, c_3c_4)u_1!\big\langle \lambda y_1.\, \overline{c_3}!\langle\rangle.\mathbf{0} \mid \mathcal{B}^3_\epsilon\big(y_1!\langle V_+\rangle.\mathbf{0}\big)\big\rangle.\mathbf{0}\\ &\qquad\qquad \mid (\nu\, u_2)(\overline{u_1}?(z).\overline{c_5}!\langle\rangle.z\,\overline{u_2} \mid \mathcal{B}^5_\epsilon\big(u_2?(x).x\,(16{,}26)\big)\big)\big)\\ P''' &= (\nu\, c_3c_4c_5c_6)\big((\nu\, u_2)\overline{c_5}!\langle\rangle.\overline{c_3}!\langle\rangle.\mathbf{0} \mid \mathcal{B}^3_\epsilon\big(u_2!\langle V_+\rangle.\mathbf{0}\big) \mid \mathcal{B}^5_\epsilon\big(u_2?(x).x\,(16{,}26)\big)\big)\big)\end{aligned}$$

Interestingly, P''' strongly resembles a decomposition of the one-step reduced process in (3). This advocates the operational correspondence between a process and its decomposition. ⌟

3.4 Extensions (II): Recursion

We extend the decomposition to handle HO processes in which names can be typed with recursive session types $\mu\mathsf{t}.S$. We consider recursive types which are *simple* and *contractive*, i.e., in $\mu\mathsf{t}.S$, the body $S \neq \mathsf{t}$ does not contain recursive types. Unless stated otherwise, we shall handle *tail-recursive* session types such as, e.g., $S = \mu\mathsf{t}.?(\mathsf{Int});?(\mathsf{Bool});!\langle\mathsf{Bool}\rangle;\mathsf{t}$. Non-tail-recursive session types such as $\mu\mathsf{t}.?((\widetilde{T},\mathsf{t}){\rightarrow}\diamond);\mathtt{end}$, which is essential in the fully abstract encoding of HOπ into HO [19], can also be accommodated; see Rem. 3.29 below.

We start by extending minimal session types (Def. 3.1) with minimal recursive types:

▶ **Definition 3.17** (Minimal Recursive Session Types). *The syntax of* minimal recursive session types *for* HO *is defined as follows:*

$$\begin{aligned}M &::= \gamma \mid\ !\langle\widetilde{U}\rangle;\gamma \mid\ ?(\widetilde{U});\gamma \mid \mu\mathsf{t}.M\\ \gamma &::= \mathtt{end} \mid \mathsf{t}\end{aligned}$$

Thus, types such as $\mu\mathsf{t}.!\langle U\rangle;\mathsf{t}$ and $\mu\mathsf{t}.?(U);\mathsf{t}$ are minimal recursive session types: in fact they are tail-recursive session types with exactly one session prefix. We extend Def. 3.2 as follows:

▶ **Definition 3.18** (Decomposing Session Types, Extended (II))**.** *Let $\mu\mathsf{t}.S$ be a recursive session type. The decomposition function given in Def. 3.2 is extended as:*

$$\mathcal{G}(\mathsf{t}) = \mathsf{t} \qquad \mathcal{G}(\mu\mathsf{t}.S) = \begin{cases} \mathcal{R}(S) & \textit{if } \mu\mathsf{t}.S \textit{ is tail-recursive} \\ \mu\mathsf{t}.\mathcal{G}(S) & \textit{otherwise} \end{cases}$$

$$\mathcal{R}(\mathsf{t}) = \epsilon \qquad \mathcal{R}(!\langle U\rangle;S) = \mu\mathsf{t}.!\langle\mathcal{G}(U)\rangle;\mathsf{t},\mathcal{R}(S)$$

$$\mathcal{R}(?(U);S) = \mu\mathsf{t}.?(\mathcal{G}(U));\mathsf{t},\mathcal{R}(S)$$

We shall also use the function $\mathcal{R}^\star(\cdot)$, which is defined as follows:

$$\mathcal{R}^\star(?(U);S) = \mathcal{R}^\star(S) \qquad \mathcal{R}^\star(!\langle U\rangle;S) = \mathcal{R}^\star(S) \qquad \mathcal{R}^\star(\mu\mathsf{t}.S) = \mathcal{R}(S)$$

Hence, $\mathcal{G}(\mu\mathsf{t}.S)$ is a list of minimal recursive session types, obtained using the auxiliary function $\mathcal{R}(\cdot)$ on S: if S has k prefixes then the list $\mathcal{G}(\mu\mathsf{t}.S)$ will contain k minimal recursive session types. The auxiliary function $\mathcal{R}^\star(\cdot)$ decomposes *guarded* recursive session types: it skips session prefixes until a type of form $\mu\mathsf{t}.S$ is encountered; when that occurs, the recursive type is decomposed using $\mathcal{R}(\cdot)$. We illustrate Def. 3.18 with two examples:

▶ **Example 3.19** (Decomposing a Recursive Type)**.** Let $S = \mu\mathsf{t}.S'$ be a recursive session type, with $S' =?(\mathsf{Int});?(\mathsf{Bool});!\langle\mathsf{Bool}\rangle;\mathsf{t}$. By Def. 3.18, since S is tail-recursive, $\mathcal{G}(S) = \mathcal{R}(S')$. Further, $\mathcal{R}(S') = \mu\mathsf{t}.?(\mathcal{G}(\mathsf{Int}));\mathsf{t},\mathcal{R}(?(\mathsf{Bool});!\langle\mathsf{Bool}\rangle;\mathsf{t})$. By definition of $\mathcal{R}(\cdot)$, we obtain $\mathcal{G}(S) = \mu\mathsf{t}.?(\mathsf{Int});\mathsf{t},\mu\mathsf{t}.?(\mathsf{Bool});\mathsf{t},\mu\mathsf{t}.!\langle\mathsf{Bool}\rangle;\mathsf{t},\mathcal{R}(t)$ (using $\mathcal{G}(\mathsf{Int}) = \mathsf{Int}$ and $\mathcal{G}(\mathsf{Bool}) = \mathsf{Bool}$). Since $\mathcal{R}(\mathsf{t}) = \epsilon$, we obtain $\mathcal{G}(S) = \mu\mathsf{t}.?(\mathsf{Int});\mathsf{t},\mu\mathsf{t}.?(\mathsf{Bool});\mathsf{t},\mu\mathsf{t}.!\langle\mathsf{Bool}\rangle;\mathsf{t}$. ⌟

▶ **Example 3.20** (Decomposing an Unfolded Recursive Type)**.** Let $T =?(\mathsf{Bool});!\langle\mathsf{Bool}\rangle;S$ be a derived unfolding of S from Exam. 3.19. Then, by Def. 3.18, $\mathcal{R}^\star(T)$ is the list of minimal recursive types obtained as follows: first, $\mathcal{R}^\star(T) = \mathcal{R}^\star(!\langle\mathsf{Bool}\rangle;\mu\mathsf{t}.S')$ and after one more step, $\mathcal{R}^\star(!\langle\mathsf{Bool}\rangle;\mu\mathsf{t}.S') = \mathcal{R}^\star(\mu\mathsf{t}.S')$. Finally, we have $\mathcal{R}^\star(\mu\mathsf{t}.S') = \mathcal{R}(S')$. We get the same list of minimal types as in Exam. 3.19: $\mathcal{R}^\star(T) = \mu\mathsf{t}.?(\mathsf{Int});\mathsf{t},\mu\mathsf{t}.?(\mathsf{Bool});\mathsf{t},\mu\mathsf{t}.!\langle\mathsf{Bool}\rangle;\mathsf{t}$. ⌟

We now explain how to decompose processes whose names are typed with recursive types. In the core fragment, we decompose a name u into a sequence of names $\tilde{u} = (u_1,\ldots,u_n)$: each $u_i \in \tilde{u}$ is used exactly by one trio to perform exactly one action; the session associated to u_i ends after its single use, as prescribed by its minimal session type. The situation is different when names can have recursive types, for the names $\tilde{u}$ should be propagated in order to be used infinitely many times. As a simple example, consider the process

$$R = r?(x).r!\langle x\rangle.V\,r$$

where name r has type $S = \mu\mathsf{t}.?(\mathsf{Int});!\langle\mathsf{Int}\rangle;\mathsf{t}$ and the higher-order type of V is $S \rightarrow \diamond$. Processes of this form are key in the encoding of recursion given in [19]. A naive decomposition of R, using the approach we defined for processes without recursive types, would result into

$$\mathcal{B}^1_\epsilon(R) = c_1?().r_1?(x).\overline{c_2}!\langle x\rangle.\mathbf{0} \mid c_2?(x).r_2!\langle x\rangle.\overline{c_3}!\langle\rangle.\mathbf{0} \mid c_3?().\mathcal{V}_\epsilon(V)\,(r_3,r_4)$$

There are several issues with this breakdown. One of them is typability: we have that $r_1 : \mu\mathsf{t}.?(\mathsf{Int});\mathsf{t}$, but subprocess $\overline{c_2}!\langle x\rangle.\mathbf{0}$ is not typable under a linear environment containing such a judgment. Another, perhaps more central, issue concerns $\tilde{r}$: the last trio (which mimics application) should apply to the sequence of names (r_1,r_2), rather than to (r_3,r_4). We address both issues by devising a mechanism that propagates names with recursive types (such as (r_1,r_2)) among the trios that use some of them. This entails decomposing R in such a way that the first two trios propagate r_1 and r_2 after they have used them; the trio simulating $V\,r$ should then have a way to access the propagated names (r_1,r_2).

We illustrate the key insights underpinning our solution by means of two examples. The first one illustrates how to break down input and output actions on names with recursive types (the "first part" of R). The second example shows how to break down an application where a value is applied to a tuple of names with recursive names (the "second part" of R).

▶ **Example 3.21** (Decomposing Processes with Recursive Names (I)). Let $P = r?(x).r!\langle x\rangle.P'$ be a process where r has type $S = \mu \mathsf{t}.?(\mathsf{Int});!\langle \mathsf{Int}\rangle;\mathsf{t}$ and $r \in \mathtt{fn}(P')$. To define $\mathcal{B}^1_\epsilon\big(P\big)$ in a compositional way, names (r_1, r_2) should be provided to its first trio; they cannot be known beforehand. To this end, we introduce a new control trio that will hold these names:

$$c^r?(b).b\,(r_1, r_2)$$

where the *shared* name c^r provides a decomposition of the (recursive) name r. The intention is that each name with a recursive type r will get its own dedicated propagator channel c^r. Since there is only one recursive name in P, its decomposition will be of the following form:

$$\mathcal{D}(P) = (\nu\,\tilde{c})(\nu\,c^r)\big(c^r?(b).b\,(r_1, r_2) \mid \overline{c_1}!\langle\rangle.\mathbf{0} \mid \mathcal{B}^1_\epsilon\big(P\big)\big)$$

The new control trio can be seen as a server that provides names: each trio that mimics some action on r should request the sequence $\tilde{r}$ from the server on c^r. This request will be realized by a higher-order communication: trios should send an abstraction to the server; such an abstraction will contain further actions of a trio and it will be applied to the sequence $\tilde{r}$. Following this idea, we may refine the definition of $\mathcal{D}(P)$ by expanding $\mathcal{B}^1_\epsilon\big(P\big)$:

$$\mathcal{D}(P) = (\nu\,\tilde{c})(\nu\,c^r)\big(c^r?(b).b\,(r_1, r_2) \mid \overline{c_1}!\langle\rangle.\mathbf{0} \mid c_1?().c^r!\langle N_1\rangle.\mathbf{0} \mid c_2?(y).c^r!\langle N_2\rangle.\mathbf{0} \mid \mathcal{B}^3_\epsilon\big(P'\big)\big)$$

The trios involving names with recursive types have now a different shape. After being triggered by a previous trio, rather than immediately mimicking an action, they will send an abstraction to the server available on c^r. The abstractions N_1 and N_2 are defined as follows:

$$N_1 = \lambda(z_1, z_2).\, z_1?(x).\overline{c_2}!\langle x\rangle.c^r?(b).b\,(z_1, z_2) \quad N_2 = \lambda(z_1, z_2).\, z_2!\langle x\rangle.\overline{c_3}!\langle\rangle.c^r?(b).b\,(z_1, z_2)$$

Hence, the formal arguments for these values are meant to correspond to $\tilde{r}$. The server on name c^r will appropriately instantiate these names. Notice that all names in $\tilde{r}$ are propagated, even if the abstractions only use some of them. For instance, N_1 only uses r_1, whereas N_2 uses r_2. After simulating an action on r_i and activating the next trio, these values reinstate the server on c^r for the benefit of future trios mimicking actions on r. ⌟

▶ **Example 3.22** (Decomposing Processes with Recursive Names (II)). Let $S = \mu \mathsf{t}.?(\mathsf{Int});!\langle \mathsf{Int}\rangle;\mathsf{t}$ and $T = \mu \mathsf{t}.?(\mathsf{Bool});!\langle \mathsf{Bool}\rangle;\mathsf{t}$, and define $Q = V\,(u, v)$ as a process where $u : S$ and $v : T$, where V is some value of type $(S, T) \rightarrow \diamond$. The decomposition of Q is as in the previous example, except that now we need two servers, one for u and one for v:

$$\mathcal{D}(Q) = (\nu\,c_1\tilde{c})(\nu\,c^u c^v)\big(c^u?(b).b\,(u_1, u_2) \mid c^v?(b).b\,(v_1, v_2) \mid \overline{c_1}!\langle\rangle.\mathbf{0} \mid \mathcal{B}^1_\epsilon\big(Q\big)\big)$$

where $\tilde{c} = (c_2, \ldots, c_{|Q|})$. We should break down Q in such a way that it could communicate with both servers to collect sequences $\tilde{u}$ and $\tilde{v}$. To this end, we define a process in which abstractions are nested using output prefixes and whose innermost process is an application. After successive communications with multiple servers this innermost application will have collected all names in $\tilde{u}$ and $\tilde{v}$. We apply this idea to breakdown Q:

$$\mathcal{B}^1_\epsilon\big(Q\big) = c_1?().c^u!\big\langle\lambda(x_1, x_2).\, c^v!\langle\lambda(y_1, y_2).\, \mathcal{V}^2_\epsilon\big(V\big)\,(x_1, x_2, y_1, y_2)\rangle.\mathbf{0}\big\rangle.\mathbf{0}$$

Observe that we use two nested outputs, one for each name with recursive types in Q. We now look at the reductions of $\mathcal{D}(Q)$ to analyze how the communication of nested abstractions allows us to collect all name sequences needed. After the first reduction along c_1 we have:

$$\begin{aligned}\mathcal{D}(Q) \longrightarrow &(\nu\, \tilde{c})(\nu\, c^u c^v)\big(c^u?(b).b\,(u_1,u_2) \mid c^v?(b).b\,(v_1,v_2) \mid \\ &c^u!\langle\lambda(x_1,x_2).\, c^v!\langle\lambda(y_1,y_2).\, \mathcal{V}^2_\epsilon\big(V\big)\,(x_1,x_2,y_1,y_2)\rangle.\mathbf{0}\rangle.\mathbf{0} = R^1\end{aligned}$$

From R^1 we have a synchronization along name c^u:

$$\begin{aligned}R^1 \longrightarrow &(\nu\, \tilde{c})(\nu\, c^u c^v)\big(\lambda(x_1,x_2).\, c^v!\langle\lambda(y_1,y_2).\, \mathcal{V}^2_\epsilon\big(V\big)\,(x_1,x_2,y_1,y_2)\rangle.\mathbf{0}\,(u_1,u_2) \mid \\ &c^v?(b).b\,(v_1,v_2)\big) = R^2\end{aligned}$$

Upon receiving the value, the server applies it to (u_1,u_2) obtaining the following process:

$$R^2 \longrightarrow (\nu\, \tilde{c})(\nu\, c^u c^v)\big(c^v!\langle\lambda(y_1,y_2).\, \mathcal{V}^2_\epsilon\big(V\big)\,(u_1,u_2,y_1,y_2)\rangle.\mathbf{0} \mid c^v?(b).b\,(v_1,v_2)\big) = R^3$$

Up to here, we have partially instantiated name variables of a value with the sequence $\tilde{u}$. Next, the first trio in R^3 can communicate with the server on name c^v:

$$\begin{aligned}R^3 \longrightarrow &(\nu\, \tilde{c})(\nu\, c^u c^v)\big(\lambda(y_1,y_2).\, \mathcal{V}^2_\epsilon\big(V\big)\,(u_1,u_2,y_1,y_2)\,(v_1,v_2)\big) \\ \longrightarrow &(\nu\, \tilde{c})(\nu\, c^u c^v)\big(\mathcal{V}^2_\epsilon\big(V\big)\,(u_1,u_2,v_1,v_2)\big)\end{aligned}$$

This completes the instantiation of name variables with appropriate sequences of names with recursive types. At this point, $\mathcal{D}(Q)$ can proceed to mimic the application in Q. ⌟

These two examples illustrate the main ideas of the decomposition of processes that involve names with recursive types. Tab. 4 presents a formal account of the extension of the definition of process decomposition given in Def. 3.8. Before explaining the table in detail, we require an auxiliary definition.

Given an unfolded recursive session type S, the auxiliary function $f(S)$ returns the position of the top-most prefix of S within its body. (Whenever $S = \mu \mathsf{t}.S'$, we have $f(S) = 1$.)

▶ **Definition 3.23** (Index function). *Let S be an (unfolded) recursive session type. The function $f(S)$ is defined as follows:*

$$f(S) = \begin{cases} f'_0(S'\{S/\mathsf{t}\}) & \text{if } S = \mu \mathsf{t}.S' \\ f'_0(S) & \text{otherwise}\end{cases}$$

where: $f'_l(\mu \mathsf{t}.S) = |\mathcal{R}(S)| - l + 1, \qquad f'_l(!\langle U\rangle;S) = f'_{l+1}(S), \qquad f'_l(?(U);S) = f'_{l+1}(S).$

▶ **Example 3.24.** Let $S' = ?(\mathsf{Bool});!\langle\mathsf{Bool}\rangle;S$ where S is as in Exam. 3.19. Then $f(S') = 2$ since the top-most prefix of S' (‘$?(\mathsf{Bool});$’) is the second prefix in the body of S. ⌟

Given a typed process P, we write $\mathtt{rn}(P)$ to denote the set of free names of P whose types are recursive. As mentioned above, for each $r \in \mathtt{rn}(P)$ with $r : S$ we shall rely on a control trio of the form $c^r?(b).b\,\widetilde{r}$, where $\widetilde{r} = r_1, \ldots, r_{|\mathcal{G}(S)|}$.

▶ **Definition 3.25** (Decomposition of a Process with Recursive Session Types). *Let P be a closed* HO *process with $\widetilde{u} = \mathtt{fn}(P)$ and $\widetilde{v} = \mathtt{rn}(P)$. The* decomposition *of P, denoted $\mathcal{D}(P)$, is defined as:*

$$\mathcal{D}(P) = (\nu\, \widetilde{c})(\nu\, \widetilde{c_r})\Big(\prod_{r \in \tilde{v}} c^r?(b).b\,\widetilde{r} \mid \overline{c_k}!\langle\rangle.\mathbf{0} \mid \mathcal{B}^k_\epsilon\big(P\sigma\big)\Big)$$

where: $k > 0$; $\widetilde{c} = (c_k, \ldots, c_{k+|P|-1})$; $\widetilde{c_r} = \bigcup_{r \in \tilde{v}} c^r$; $\sigma = \{\mathsf{init}(\widetilde{u})/\widetilde{u}\}$.

Table 4 The breakdown function for processes and values (extension with recursive types).

$\mathcal{B}^{k}_{\tilde{x}}\big(r!\langle V\rangle.Q\big)$	
$c_k?(\widetilde{x}).c^r!\big\langle N_V\big\rangle \mid\ \mathcal{B}^{k+l+1}_{\tilde{w}}\big(Q\big)$ where: $N_V = \lambda\widetilde{z}.\, z_{f(S)}!\big\langle\mathcal{V}^{k+1}_{\tilde{y}}\big(V\big)\big\rangle.$ $\overline{c_{k+l+1}}!\langle\widetilde{w}\rangle.c^r?(b).(b\,\widetilde{z})$	$r : S \wedge \mathsf{tr}(S)$ $\widetilde{y} = \mathtt{fv}(V),\ \widetilde{w} = \mathtt{fv}(Q)$ $l = \lvert V\rvert$ $\widetilde{z} = (z_1, \ldots, z_{\lvert\mathcal{R}^\star(S)\rvert})$
$\mathcal{B}^{k}_{\tilde{x}}\big(r?(y).Q\big)$	
$c_k?(\widetilde{x}).c^r!\big\langle N_y\big\rangle \mid \mathcal{B}^{k+1}_{\tilde{x}'}\big(Q\big)$ where: $N_y = \lambda\widetilde{z}.\, z_{f(S)}?(y).\overline{c_{k+1}}!\langle\widetilde{x}'\rangle.c^r?(b).(b\,\widetilde{z})$	$r : S \wedge \mathsf{tr}(S)$ $\widetilde{x}' = \mathtt{fv}(Q)$ $\widetilde{z} = (z_1, \ldots, z_{\lvert\mathcal{R}^\star(S)\rvert})$
$\mathcal{B}^{k}_{\tilde{x}}\big(V\,(\widetilde{r}, u_i)\big)$	
$c_k?(\widetilde{x}).\,\overbrace{c^{r_1}!\big\langle\lambda\widetilde{z}_1.c^{r_2}!\langle\lambda\widetilde{z}_2.\cdots.c^{r_n}!\langle\lambda\widetilde{z}_n.}^{n=\lvert\tilde{r}\rvert}Q\rangle\,\rangle\big\rangle$ where: $Q = \mathcal{V}^{k+1}_{\tilde{x}}\big(V\big)\,(\widetilde{z}_1, \ldots, \widetilde{z}_n, \widetilde{m})$	$\forall r_i \in \widetilde{r}.(r_i : S_i \wedge \mathsf{tr}(S_i) \wedge$ $\widetilde{z}_i = (z^i_1, \ldots, z^i_{\lvert\mathcal{R}^\star(S_i)\rvert}))$ $u_i : C$ $\widetilde{m} = (u_i, \ldots, u_{i+\lvert\mathcal{G}(C)\rvert-1})$
$\mathcal{B}^{k}_{\tilde{x}}\big((\nu\, s : \mu\mathsf{t}.S)P'\big)$	
$(\nu\,\widetilde{s} : \mathcal{R}(S))(\nu\, c^s)c^s?(b).(b\,\widetilde{s}) \mid$ $(\nu\, c^{\bar{s}})c^{\bar{s}}?(b).(b\,\widetilde{\bar{s}}) \mid\ \mathcal{B}^{k}_{\tilde{x}}\big(P'\big)$	$\mathsf{tr}(\mu\mathsf{t}.S)$ $\widetilde{s} = (s_1, \ldots, s_{\lvert\mathcal{R}(S)\rvert})$ $\widetilde{\bar{s}} = (\overline{s_1}, \ldots, \overline{s_{\lvert\mathcal{R}(S)\rvert}})$
$\mathcal{V}^{k}_{\tilde{x}}\big(\lambda(\widetilde{y}, z) : (\widetilde{S}, C)^{\leadsto}.\,P\big)$	
$\lambda(\widetilde{y^1}, \ldots, \widetilde{y^n}, \widetilde{z}) : (\widetilde{T})^{\leadsto}.\,N$ where: $\widetilde{T} = (\mathcal{G}(S_1), \ldots, \mathcal{G}(S_n), \mathcal{G}(C))$ $N = (\nu\,\widetilde{c})\prod_{i\in\lvert\widetilde{y}\rvert}(c^{y_i}?(b).(b\,\widetilde{y}^i)) \mid \overline{c_k}!\langle\widetilde{x}\rangle \mid$ $\mathcal{B}^{k}_{\tilde{x}}\big(P\{^{z_1}/_{z}\}\big)$	$\forall y_i \in \widetilde{y}.(y_i : S_i \wedge \mathsf{tr}(S_i) \wedge$ $\widetilde{y^i} = (y^i_1, \ldots, y^i_{\lvert\mathcal{G}(S_i)\rvert}))$ $\widetilde{z} = (z_1, \ldots, z_{\lvert\mathcal{G}(C)\rvert})$ $\widetilde{c} = \begin{cases} \epsilon & \text{if } \leadsto = \multimap \\ (c_k, \ldots, c_{k+\lvert P\rvert-1}) & \text{if } \leadsto = \longrightarrow \end{cases}$

We now describe the required extensions for the function $\mathcal{B}^{k}_{\tilde{x}}\big(\,\cdot\,\big)$. We will use predicate $\mathsf{tr}(S)$ on types to indicate that S is a tail-recursive session type. Tab. 4 describes the breakdown of prefixes whose type is recursive; all other prefixes can be treated as in Tab. 1.

Output. The breakdown of process $r!\langle V\rangle.Q$, when r has a recursive type S, is as follows:

$$\begin{aligned}\mathcal{B}^{k}_{\tilde{x}}\big(r!\langle V\rangle.Q\big) = {} & c_k?(\widetilde{x}).c^r!\big\langle N_V\big\rangle \mid \mathcal{B}^{k+l+1}_{\tilde{w}}\big(Q\big) \\ & \text{where } N_V = \lambda\widetilde{z}.\, z_{f(S)}!\big\langle\mathcal{V}^{k+1}_{\tilde{y}}\big(V\big)\big\rangle.\overline{c_{k+l+1}}!\langle\widetilde{w}\rangle.c^r?(b).(b\,\widetilde{z})\end{aligned}$$

The decomposition consists of a leading trio that mimics the output action running in parallel with the breakdown of Q. After receiving the context $\widetilde{x}$, the leading trio sends an abstraction N_V along c^r. Value N_V performs several tasks. First, it collects the sequence $\tilde{r}$; then, it mimics the output action of P along one of them $(r_{f(S)})$ and triggers the next trio, with context $\widetilde{w}$; finally, it reinstates the server on c^r for the next trio that uses r. Notice that differently from what is done in Tab. 1, indexing is not relevant when breaking down names with recursive types. Also, since by definition $\mathcal{V}^k_{\tilde{y}}\big(y\big) = y$, $y\sigma = y$, and $|y| = 0$, when the communicated value V is a variable y we obtain the following:

$$\mathcal{B}^k_{\tilde{x}}\big(r!\langle y\rangle.Q\big) = c_k?(\widetilde{x}).c^r!\big\langle \lambda\widetilde{z}.\, z_{f(S)}!\langle y\rangle.\overline{c_{k+1}}!\langle\widetilde{w}\rangle.c^r?(b).(b\,\widetilde{z})\big\rangle \mid \mathcal{B}^{k+1}_{\tilde{w}}\big(Q\big)$$

Input. The breakdown of process $r?(y).Q$, when r has recursive session type S, is as follows:

$$\mathcal{B}^k_{\tilde{x}}\big(r?(y).Q\big) = c_k?(\widetilde{x}).c^r!\big\langle \lambda\widetilde{z}.\, z_{f(S)}?(y).\overline{c_{k+1}}!\langle\widetilde{x}'\rangle.c^r?(b).(b\,\widetilde{z})\big\rangle \mid \mathcal{B}^{k+1}_{\tilde{x}'}\big(Q\big)$$

The breakdown follows the lines of the output case, but also of the linear case in Tab. 1, with additional structure needed to implement the reception of $\tilde{r}$, using one of the received names ($r_{f(S)}$) as a subject for the input action and propagating those names further.

Application. For simplicity we consider applications $V\,(\widetilde{r}, u_i)$, where names in $\widetilde{r}$ have recursive types and only name u_i has a non-recursive type; the general case involving different orders in names and multiple names with non-recursive types is as expected. We have:

$$\mathcal{B}^k_{\tilde{x}}\big(V\,(\widetilde{r}, u_i)\big) = c_k?(\widetilde{x}).\,\overbrace{c^{r_1}!\big\langle\lambda\widetilde{z}_1.c^{r_2}!\langle\lambda\widetilde{z}_2.\cdots.c^{r_n}!\langle\lambda\widetilde{z}_n.}^{n=|\tilde{r}|}\,\mathcal{V}^{k+1}_{\tilde{x}}\big(V\big)\,(\widetilde{z}_1,\ldots,\widetilde{z}_n,\widetilde{m})\rangle\,\rangle\big\rangle$$

We rely on types to decompose every name in $(\widetilde{r}, u_i)$. Letting $|\tilde{r}| = n$ and $i \in \{1,\ldots,n\}$, for each $r_i \in \widetilde{r}$ (with $r_i : S_i$) we generate a sequence $\widetilde{z}_i = (z^i_1,\ldots,z^i_{|\mathcal{R}^\star(S_i)|})$ as in the output case. Since name u_i has a non-recursive session type, we decompose it as in Tab. 1. Subsequently, we define an output action on propagator c^{r_1} that sends a value containing n abstractions that occur nested within output prefixes: for each $j \in \{1,\ldots,n-1\}$, each abstraction binds $\widetilde{z}_j$ and sends the next abstraction along $c^{r_{j+1}}$. The innermost abstraction abstracts over $\widetilde{z}_n$ and encapsulates process $\mathcal{V}^{k+1}_{\tilde{x}}\big(V\big)\,(\widetilde{z}_1,\ldots,\widetilde{z}_n,\widetilde{m})$, which mimics the application in the source process. By this abstraction nesting we bind all variables $\widetilde{z}_i$ in Q. This structure can be seen as an encoding of partial application: by virtue of a single synchronization on c^{r_i} part of variables (i.e., $\widetilde{z}_i$) will be instantiated.

The breakdown of a value application of the form $y\,(\widetilde{r}, u_i)$ results into a specific form of the breakdown:

$$\mathcal{B}^k_{\tilde{x}}\big(y\,(\widetilde{r}, u_i)\big) = c_k?(\widetilde{x}).\,\overbrace{c^{r_1}!\big\langle\lambda\widetilde{z}_1.c^{r_2}!\langle\lambda\widetilde{z}_2.\cdots.c^{r_n}!\langle\lambda\widetilde{z}_n.}^{n=|\tilde{r}|}\,y\,(\widetilde{z}_1,\ldots,\widetilde{z}_n,\widetilde{m})\rangle\,\rangle\big\rangle$$

Restriction. The restriction process $(\nu\, s : \mu\mathsf{t}.S)P'$ is translated as follows:

$$\mathcal{B}^k_{\tilde{x}}\big((\nu\, s : \mu t.S)P'\big) = (\nu\,\widetilde{s} : \mathcal{R}(S))(\nu\, c^s)c^s?(b).(b\,\widetilde{s}) \mid (\nu\, c^{\overline{s}})c^{\overline{s}}?(b).(b\,\widetilde{\overline{s}}) \mid \mathcal{B}^k_{\tilde{x}}\big(P'\big)$$

We decompose s into $\widetilde{s} = (s_1,\ldots,s_{|\mathcal{R}(S)|})$ and $\overline{s}$ into $\widetilde{\overline{s}} = (\overline{s_1},\ldots,\overline{s_{|\mathcal{R}(S)|}})$. The breakdown introduces two servers in parallel with the breakdown of P'; these servers provide names for s and $\overline{s}$ along c^s and $c^{\overline{s}}$, respectively. The server on c^s (resp. $c^{\overline{s}}$) receives a value and applies it to the sequence $\widetilde{s}$ (resp. $\widetilde{\overline{s}}$). We restrict over $\widetilde{s}$ and propagators c^s and $c^{\overline{s}}$.

Value. The polyadic value $\lambda(\widetilde{y}, z) : (\widetilde{S}, C)^{\leadsto}.\,P$, where $\leadsto\in \{\multimap, \rightarrow\}$, is decomposed as follows:

$$\mathcal{V}^k_{\tilde{x}}\big(\lambda(\widetilde{y}, z) : (\widetilde{S}, C)^{\leadsto}.\,P\big) = \lambda(\widetilde{y^1},\ldots,\widetilde{y^n},\widetilde{z}) : (\mathcal{G}(S_1),\ldots,\mathcal{G}(S_n),\mathcal{G}(C))^{\leadsto}.\,N$$
$$\text{where:}\quad N = (\nu\,\widetilde{c}) \prod_{i\in|\widetilde{y}|} (c^{y_i}?(b).(b\,\widetilde{y}^i)) \mid \overline{c_k}!\langle\widetilde{x}\rangle \mid \mathcal{B}^k_{\tilde{x}}\big(P\{z_1/z\}\big)$$

We assume variables in $\widetilde{y}$ have recursive session types $\widetilde{S}$ and variable z has some non-recursive session type C; the general case involving different orders in variables and multiple variables with non-recursive types is as expected. Every variable y_i (with $y_i : S_i$)

is decomposed into $\widetilde{y}^i = (y_1, \ldots, y_{|\mathcal{G}(S_i)|})$. Variable z is decomposed as in Tab. 1. The breakdown is similar to the (monadic) shared value given in Tab. 1. In this case, for every $y_i \in \widetilde{y}$ there is a server $c^{y_i}?(b).(b\,\widetilde{y}^i)$ as a subprocess in the abstracted composition. The rationale for these servers is as described in previous cases.

To sum up, each trio using a name with a recursive session type first receives a sequence of names; then, it uses one of such names to mimic the appropriate action; finally, it propagates the entire sequence by reinstating a server defined as a control trio. Interestingly, this scheme for name propagation follows the implementation of the encoding of name-passing in HO.

▶ **Example 3.26** (Breakdown of Recursion Encoding). Consider the recursive process $P = \mu X.a?(m).a!\langle m\rangle.X$, which is not an HO process. P can be encoded into HO as follows [19]:

$$[\![P]\!] = a?(m).a!\langle m\rangle.(\nu\, s)(V\,(a,s) \mid s!\langle V\rangle.\mathbf{0})$$

where the value V is an abstraction that potentially reduces to $[\![P]\!]$:

$$V = \lambda(x_a, y_1).\, y_1?(z_x).x_a?(m).x_a!\langle m\rangle.(\nu\, s)(z_x\,(x_a, s) \mid \overline{s}!\langle z_x\rangle.\mathbf{0})$$

We compose $[\![P]\!]$ with an appropriate client process to illustrate the encoding of recursion:

$$\begin{aligned}
&[\![P]\!] \mid a!\langle W\rangle.a?(b).R\\
&\longrightarrow^2 (\nu\, s)(V\,(a,s) \mid s!\langle V\rangle.\mathbf{0}) \mid R\\
&\longrightarrow (\nu\, s)(s?(z_x).a?(m).a!\langle m\rangle.(\nu\, s')(z_x\,(a,s') \mid \overline{s'}!\langle z_x\rangle.\mathbf{0}) \mid s!\langle V\rangle.\mathbf{0}) \mid R\\
&\longrightarrow a?(m).a!\langle m\rangle.(\nu\, s')(V\,(a,s') \mid \overline{s'}!\langle V\rangle.\mathbf{0}) \mid R = [\![P]\!] \mid R
\end{aligned}$$

where R is some unspecified process such that $a \in \mathtt{rn}(R)$. We now analyze $\mathcal{D}([\![P]\!])$ and its reduction chain. By Def. 3.5, we have $|[\![P]\!]| = 7$, and $|V| = 0$. Then, we choose $k = 1$ and observe that $\sigma = \{{}^{a_1\overline{a_1}}/_{a\overline{a}}\}$. Following Def. 3.25, we get:

$$\begin{aligned}
\mathcal{D}([\![P]\!]) &= (\nu\, c_1,\ldots,c_7)(\nu\, c^a)(c^a?(b).b\,(a_1,a_2) \mid \overline{c_1}!\langle\rangle.\mathbf{0} \mid \mathcal{B}^1_\epsilon\big([\![P]\!]\sigma\big))\\
\mathcal{B}^1_\epsilon\big([\![P]\!]\big) &= c_1?().c^a!\langle\lambda(z_1,z_2).\, z_1?(m).\overline{c_2}!\langle m\rangle.c^a?(b).b\,(z_1,z_2)\rangle.\mathbf{0}\\
&\quad \mid c_2?(m).c^a!\langle\lambda(z_1,z_2).\, z_2!\langle m\rangle.\overline{c_3}!\langle\rangle.c^a?(b).b\,(z_1,z_2)\rangle.\mathbf{0}\\
&\quad \mid (\nu\, s_1)\big(c_3?().\overline{c_4}!\langle\rangle.\overline{c_5}!\langle\rangle.\mathbf{0} \mid c_4?().\overline{c^a}!\langle\lambda(z_1,z_2).\, \mathcal{V}^5_\epsilon\big(V\big)\,(z_1,z_2,s_1)\rangle.\mathbf{0}\\
&\qquad \mid c_5?().\overline{s}_1!\langle\mathcal{V}^6_\epsilon\big(V\big)\rangle.\overline{c_7}!\langle\rangle.\mathbf{0} \mid c_7?().\mathbf{0}\big)
\end{aligned}$$

The decomposition relies twice on the breakdown of value V, so we give $\mathcal{V}^k_\epsilon\big(V\big)$ here for arbitrary $k > 0$. For this, we observe that V is an abstraction of a process Q with $|Q| = 7$.

$$\begin{aligned}
\mathcal{V}^k_\epsilon\big(V\big) &= \lambda(x_{a_1},x_{a_2},y_1).\,(\nu\, c_k,\ldots,c_{k+6})(c^{x_a}?(b).b\,(x_{a_1},x_{a_2}) \mid \overline{c_k}!\langle\rangle.\mathbf{0} \mid \mathcal{B}^k_\epsilon\big(Q\big))\\
\mathcal{B}^k_\epsilon\big(Q\big) &= c_k?().y_1?(z_x).\overline{c_{k+1}}!\langle z_x\rangle.\mathbf{0}\\
&\mid c_{k+1}?(z_x).c^a_1!\langle\lambda(z_1,z_2).\, z_1?(m).\overline{c_{k+2}}!\langle z_x, m\rangle.c^a_2?(b).b\,(z_1,z_2)\rangle.\mathbf{0}\\
&\mid c_{k+2}?(z_x).c^a_2!\langle\lambda(z_1,z_2).\, z_2!\langle m\rangle.\overline{c_{k+3}}!\langle z_x\rangle.c^a_3?(b).b\,(z_1,z_2)\rangle.\mathbf{0}\\
&\mid (\nu\, s_1)\big(c_{k+3}?(x_z).\overline{c_{k+4}}!\langle z_x\rangle.\overline{c_{k+5}}!\langle z_x\rangle.\mathbf{0}\\
&\mid c_{k+4}?(z_x).c^a_3!\langle\lambda(z_1,z_2).\, z_x\,(z_1,z_2,s_1)\rangle.\mathbf{0} \mid c_{k+5}?(z_x).\overline{s_1}!\langle z_x\rangle.\overline{c_{k+6}}!\langle\rangle.\mathbf{0} \mid c_{k+6}?().\mathbf{0}\big)
\end{aligned}$$

We follow the reduction chain on $\mathcal{D}([\![P]\!])$ until it is ready to mimic the first action with channel a, which is an input. First, c_1 will synchronize, after which c^a sends the abstraction to which then (a_1, a_2) is applied. We obtain $\mathcal{D}(\llbracket P \rrbracket) \longrightarrow^3 (\nu\, c_2, \ldots, c_7, c^a)P'$, where

$$\begin{aligned} P' = {} & a_1?(m).\overline{c_2}!\langle m\rangle.c^a?(b).b\,(a_1, a_2) \\ & \mid c_2?(m).c^a!\langle \lambda(z_1, z_2).\, z_2!\langle m\rangle.\overline{c_3}!\langle\rangle.c^a?(b).b\,(z_1, z_2)\rangle.\mathbf{0} \\ & \mid (\nu\, s_1)\big(c_3?().\overline{c_4}!\langle\rangle.\overline{c_5}!\langle\rangle.\mathbf{0} \mid c_4?().\overline{c^a}!\langle \lambda(z_1, z_2).\, \mathcal{V}^5_\epsilon\big(V\big)\,(z_1, z_2, s_1)\rangle.\mathbf{0} \\ & \qquad \mid c_5?().\overline{s}_1!\langle \mathcal{V}^6_\epsilon\big(V\big)\rangle.\overline{c_7}!\langle\rangle.\mathbf{0} \mid c_7?().\mathbf{0}\big) \end{aligned}$$

Note that this process is awaiting an input on channel a_1, after which c_2 can synchronize with its dual. At that point, c^a is ready to receive another abstraction that mimics an input on a_1. This strongly suggests a tight operational correspondence between a process P and its decomposition in the case where P performs higher-order recursion. ⌟

Below we write Δ_μ to denote a session environment that concerns only recursive types. We state our main results:

▶ **Theorem 3.27** (Typability of Breakdown). *Let P be an initialized* HO *process and V be a value.*

1. *If $\Gamma; \Lambda; \Delta, \Delta_\mu \vdash P \triangleright \diamond$ then $\mathcal{G}(\Gamma_1), \Phi; \emptyset; \mathcal{G}(\Delta), \Theta \vdash \mathcal{B}^k_{\tilde{x}}\big(P\big) \triangleright \diamond$ where: $\widetilde{r} = dom(\Delta_\mu)$; $\Phi = \prod_{r\in\tilde{r}} c^r : \langle \mathcal{R}^\star(\Delta_\mu(r)) \multimap \diamond\rangle$; $\widetilde{x} = \mathtt{fv}(P)$; $k > 0$; $\Gamma_1 = \Gamma \setminus \widetilde{x}$; and* balanced$(\Theta)$ *with $dom(\Theta) = \{c_k, \ldots, c_{k+|P|-1}\} \cup \{\overline{c_{k+1}}, \ldots, \overline{c_{k+|P|-1}}\}$ such that $\Theta(c_k) = ?(\widetilde{M}); \mathtt{end}$, where $\widetilde{M} = (\mathcal{G}(\Gamma), \mathcal{G}(\Lambda))(\widetilde{x})$.*
2. *If $\Gamma; \Lambda; \Delta \vdash V \triangleright C \multimap \diamond$ then $\mathcal{G}(\Gamma); \mathcal{G}(\Lambda); \mathcal{G}(\Delta), \Theta \vdash \mathcal{V}^k_{\tilde{x}}\big(V\big) \triangleright \mathcal{G}(C) \multimap \diamond$, where: $\widetilde{x} = \mathtt{fv}(V)$; $k > 0$; and* balanced(Θ) *with $dom(\Theta) = \{c_k, \ldots, c_{k+|V|-1}\} \cup \{\overline{c_k}, \ldots, \overline{c_{k+|V|-1}}\}$ such that $\Theta(c_k) = ?(\widetilde{M}); \mathtt{end}$ and $\Theta(\overline{c_k}) = !\langle\widetilde{M}\rangle; \mathtt{end}$, where $\widetilde{M} = (\mathcal{G}(\Gamma), \mathcal{G}(\Lambda))(\widetilde{x})$.*
3. *If $\Gamma; \emptyset; \emptyset \vdash V \triangleright C \rightarrow \diamond$ then $\mathcal{G}(\Gamma); \emptyset; \emptyset \vdash \mathcal{V}^k_{\tilde{x}}\big(V\big) \triangleright \mathcal{G}(C) \rightarrow \diamond$ where $\widetilde{x} = \mathtt{fv}(V)$ and $k > 0$.*

Proof. By mutual induction on the structure of P and V. ◀

▶ **Theorem 3.28** (Typability of the Decomposition with Recursive Types). *Let P be a closed* HO *process with $\widetilde{u} = \mathtt{fn}(P)$ and $\widetilde{v} = \mathtt{rn}(P)$. If $\Gamma; \emptyset; \Delta, \Delta_\mu \vdash P \triangleright \diamond$, where Δ_μ only involves recursive session types, then $\mathcal{G}(\Gamma\sigma); \emptyset; \mathcal{G}(\Delta\sigma), \mathcal{G}(\Delta_\mu\sigma) \vdash \mathcal{D}(P) \triangleright \diamond$, where $\sigma = \{{}^{\mathsf{init}(\widetilde{u})}/\widetilde{u}\}$.*

Proof. Directly from the definitions, using Thm. 3.27. ◀

▶ Remark 3.29 (Non-Tail-Recursive Session Types). Our definitions and results apply to tail-recursive session types. We can accommodate the non-tail-recursive type $\mu \mathsf{t}.?((\widetilde{T}, \mathsf{t}) \rightarrow \diamond); \mathtt{end}$ into our approach: in Def. 3.18, we need to have $\mathcal{G}(\mu \mathsf{t}.S) = \mu \mathsf{t}.\mathcal{G}(S)$ if $\mu \mathsf{t}.S$ is non-tail-recursive. The decomposition functions for non-recursive session types suffice in this case.

4 Optimizations of the Decomposition

Here we briefly discuss two optimizations of the decompositions. They simplify the structure of trios and the underlying communication discipline. Interestingly, they are both enabled by the higher-order nature of HO. In fact, they hinge on *thunk processes*, i.e., inactive processes that can be activated upon reception. We write $\{\{P\}\}$ to stand for the thunk process $\lambda x : \langle \mathtt{end} \rightarrow \diamond\rangle.\, P$, with $x \notin \mathtt{fn}(P)$. We write $\mathtt{run}\,\{\{P\}\}$ to denote the application of a thunk to a (dummy) name of type $\mathtt{end} \rightarrow \diamond$. This way, we have $\mathtt{run}\,\{\{P\}\} \longrightarrow P$.

From Trios to Duos. We can simplify the breakdown functions by replacing trios with *duos*, i.e., processes with exactly two sequential prefixes. The idea is to transform trios such as $c_k?(\widetilde{x}).u!\langle V\rangle.c_{k+1}!\langle\widetilde{y}\rangle$ into the composition of a duo with a control trio:

$$c_k?(\widetilde{x}).c_{k+1}!\big\langle\{\{u!\langle V\rangle.c_{k+2}!\langle\widetilde{z}\rangle\}\}\big\rangle \mid c_{k+1}?(b).(\mathtt{run}\, b) \tag{4}$$

The first action is as before; the two remaining prefixes are encapsulated into a thunk. This thunk is sent via a propagator to the control trio that activates it upon reception. This transformation involves an additional propagator, denoted c_{k+2} above. This requires minor modifications in the definition of the degree function $|\cdot|$ (cf. Def. 3.5).

In some cases, the breakdown function in § 3.2 already produces duos. Breaking down input and output prefixes and parallel composition involves proper trios; following the scheme illustrated by (4), we can define a map $\{\!|\cdot|\!\}$ to transform these trios into duos:

$$\begin{aligned}
\{\!|c_k?(\widetilde{x}).u_i!\langle V\rangle.\overline{c_{k+1}}!\langle\widetilde{z}\rangle|\!\} &= c_k?(\widetilde{x}).\overline{c_{k+1}}!\big\langle\{\{u_i!\langle V\rangle.\overline{c_{k+2}}!\langle\widetilde{z}\rangle\}\}\big\rangle \mid c_{k+1}?(b).(\mathtt{run}\, b)\\
\{\!|c_k?(\widetilde{x}).u_i?(y).\overline{c_{k+1}}!\langle\widetilde{x}'\rangle|\!\} &= c_k?(\widetilde{x}).\overline{c_{k+1}}!\big\langle\{\{u_i?(y).\overline{c_{k+2}}!\langle\widetilde{x}'\rangle\}\}\big\rangle \mid c_{k+1}?(b).(\mathtt{run}\, b)\\
\{\!|c_k?(\widetilde{x}).\overline{c_{k+1}}!\langle\widetilde{y}\rangle.\overline{c_{k+l+1}}!\langle\widetilde{z}\rangle|\!\} &= c_k?(\widetilde{x}).\overline{c_{k+1}}!\big\langle\{\{\overline{c_{k+2}}!\langle\widetilde{y}\rangle.\overline{c_{k+l+2}}!\langle\widetilde{z}\rangle\}\}\big\rangle \mid c_{k+1}?(b).(\mathtt{run}\, b)
\end{aligned}$$

In the breakdown given in § 3.3 there is a proper trio, which can be transformed as follows:

$$\{\!|u_i \triangleleft l_j.u_i?(z).\overline{c_k}!\big\langle\widetilde{x}\big\rangle.(z\,\widetilde{y})|\!\} = u_i \triangleleft l_j.\overline{c_k}!\big\langle\{\{u_i?(z).\overline{c_{k+1}}!\langle\widetilde{x}\rangle.z\,\widetilde{y}\}\}\big\rangle \mid c_k?(b).(\mathtt{run}\, b)$$

Similarly, in the breakdown function extended with recursion (cf. § 3.4) there is only one trio pattern, which can be transformed into a duo following the very same idea.

From Polyadic to Monadic Communication. Since we consider *closed* HO processes, we can dispense with polyadic communication in the breakdown function. We can define a *monadic decomposition*, $\mathsf{D}(P)$, that simplifies Def. 3.8 as follows:

$$\mathsf{D}(P) = (\nu\,\widetilde{c})\big(c_k?(b).(\mathtt{run}\, b) \mid \mathsf{B}^k\big(P\sigma\big)\big)$$

where $k > 0$, $\widetilde{c} = (c_k, \ldots, c_{k+|P|-1})$, and σ is as in Def. 3.8. Process $c_k?(b).(\mathtt{run}\, b)$ activates a thunk received from $\mathsf{B}^k\big(\,\cdot\,\big)$, the *monadic* breakdown function that simplifies the one in Tab. 1 by using only one parameter, namely k:

$$\begin{aligned}
\mathsf{B}^k\big(u_i?(x).Q\big) &= (\nu\, c_x)\big(\overline{c_k}!\big\langle\{\{u_i?(x).c_{k+1}?(b).(c_x!\langle x\rangle \mid (\mathtt{run}\, b))\}\}\big\rangle \mid \mathsf{B}^{k+1}\big(Q\sigma\big)\big)\\
\mathsf{B}^k\big(u_i!\langle x\rangle.Q\big) &= \overline{c_k}!\big\langle\{\{c_x?(x).u_i!\langle x\rangle.c_{k+1}?(b).(\mathtt{run}\, b)\}\}\big\rangle \mid \mathsf{B}^{k+1}\big(Q\sigma\big)\\
\mathsf{B}^k\big(u_i!\langle V\rangle.Q\big) &= \overline{c_k}!\big\langle\{\{u_i!\langle\mathsf{V}^{k+1}\big(V\sigma\big)\rangle.c_{k+1}?(b).(\mathtt{run}\, b)\}\}\big\rangle \mid \mathsf{B}^{k+1}\big(Q\sigma\big)\\
\mathsf{B}^k\big(x\, u\big) &= \overline{c_k}!\big\langle\{\{c_x?(x).(x\,\widetilde{u})\}\}\big\rangle\\
\mathsf{B}^k\big(V\, u\big) &= \overline{c_k}!\big\langle\{\{\mathsf{V}^{k+1}\big(V\big)\,\widetilde{u})\}\}\big\rangle\\
\mathsf{B}^k\big((\nu\, s)P'\big) &= (\nu\,\widetilde{s})\mathsf{B}^k\big(P'\sigma\big)\\
\mathsf{B}^k\big(Q \mid R\big) &= \overline{c_k}!\langle\{\{c_{k+1}?(b).\mathtt{run}\, b \mid c_{k+|Q|+1}?(b).\mathtt{run}\, b\}\}\rangle \mid \mathsf{B}^{k+1}\big(Q\big) \mid \mathsf{B}^{k+|Q|+1}\big(R\big)
\end{aligned}$$

Above, σ is as in Tab. 1. $\mathsf{B}^k\big(\,\cdot\,\big)$ propagates values using thunks and a dedicated propagator c_x for each variable x. We describe only the definition of $\mathsf{B}^k\big(u_i?(x).Q\big)$: it illustrates key ideas common to all other cases. It consists of an output of a thunk on c_k composed in parallel with $\mathsf{B}^{k+1}\big(Q\sigma\big)$. The thunk will be activated by a process $c_k?(b).(\mathtt{run}\, b)$ at the top-level; this activation triggers the input action on u_i, and prepares the activation for the next thunk (exchanged on name c_{k+1}). Upon reception, such a thunk is activated in parallel with $c_x!\langle x\rangle$, which propagates the value received on u_i. The scope of c_x is restricted to include input actions on c_x in $\mathsf{B}^{k+1}\big(Q\sigma\big)$; such actions are the first in the thunks present in, e.g., $\mathsf{B}^{k}\big(u_i!\langle x\rangle.Q\big)$. We also need to revise the breakdown function for values $\mathcal{V}^{k}_{\tilde{x}}\big(\,\cdot\,\big)$. The breakdown functions given in § 3.3 and § 3.4 (cf. Tables 3 and 4) can be made monadic following similar lines.

These two optimizations can be combined by transforming the trios of the monadic breakdown into duos, following the key idea of the first optimization (cf. (4)).

5 Related Work

Our developments are related to results by Parrow [24], who showed that every process in the untyped, summation-free π-calculus with replication is weakly bisimilar to its decomposition into trios processes (i.e., $P \approx \mathcal{D}(P)$). We draw inspiration from insights developed in [24], but pursuing different goals in a different technical setting: our decomposition treats processes from a calculus without name-passing but with higher-order concurrency (abstraction-passing), supports labeled choices, and accommodates recursive types. Our goals are different than those in [24] because trios processes are relevant to our work in that they allow us to formally justify minimal session types; however, they are not an end in themselves. Still, we opted to retain the definitional style and terminology for trios from [24], which are elegant and clear.

Our main result connects the typability of a process with that of its decomposition; this is a *static guarantee*. Based on our examples, we conjecture that the *behavioral guarantee* given by $P \approx \mathcal{D}(P)$ in [24] holds in our setting too, under an appropriate *typed* weak bisimilarity. An obstacle here is that known notions of typed bisimilarity for session-typed processes, such as those given by Kouzapas et al. [20], are not adequate: they only relate processes typed under the *same* typing environments. We need a relaxed equivalence that (i) relates processes typable under different environments (e.g., Δ and $\mathcal{G}(\Delta)$) and (ii) admits that actions along s from P can be matched by $\mathcal{D}(P)$ using actions along s_k, for some k (and viceversa). Defining this notion precisely and studying its properties goes beyond the scope of this paper.

Our approach is broadly related to works that relate session types with other type systems for the π-calculus (cf. [17, 3, 4, 5, 10]). Kobayashi [17] encoded a finite session π-calculus into a π-calculus with linear types with usages (without sequencing); this encoding uses a continuation-passing style to codify a session name using multiple linear channels. Dardha et al. [3, 4] formalize and extend Kobayashi's approach. They use two separate encodings, one for processes and one for types. The former uses a freshly generated linear name to mimic each session action; this fresh name becomes an additional argument in communications. Polyadicity is thus an essential ingredient in [3, 4], whereas in our work it is convenient but not indispensable (cf. § 4). The encoding of types in [3, 4] codifies sequencing in session types by nesting payload types. In contrast, we "slice" the n actions occurring in a session s along indexed names $s_1, \ldots, s_n$ with minimal session types, i.e., slices of the type for s. All in all, an approach based on minimal session types appears simpler than that in [3, 4]. Works by Padovani [23] and Scalas et al. [25] is also related: they rely on [3, 4] to develop verification techniques based on session types for OCaml and Scala programs, respectively.

Gay et al. [10] formalize how to encode a monadic π-calculus, equipped with a finite variant of the binary session types of [9], into a polyadic π-calculus with an instance of the generic process types of [16]. The work of Demangeon and Honda [5] encodes a session π-calculus into a linear/affine π-calculus with subtyping based on choice and selection types. Our developments differ from these previous works in an important respect: we relate two formulations of session types, namely standard session types and minimal session types. Indeed, while [17, 3, 4, 5, 10] target the *relative expressiveness* of session-typed process languages, our work emerges as the first study of *absolute expressiveness* in this context.

Finally, we elaborate further on our choice of HO as source language. HO is a sub-calculus of HOπ, whose basic theory and expressivity were studied by Kouzapas et al. [19, 20] as a hierarchy of session-typed calculi based on relative expressiveness. Our developments enable us to include HO with minimal session types within this hierarchy. Still, our approach does not rely on having HO as source language, and can be adapted to other typed frameworks based on session types, such as the type discipline for first-order π-calculus processes in [26].

6 Concluding Remarks

Session types are a class of *behavioral types* for message-passing programs. We presented a *decomposition* of session-typed processes in [19, 20] using *minimal* session types, in which there is no sequencing. The decomposition of a process P, denoted $\mathcal{D}(P)$, is a collection of *trios processes* that trigger each other mimicking its sequencing. We prove that typability of P using standard session types implies the typability of $\mathcal{D}(P)$ with minimal session types. Our results hold for all session types constructs, including labeled choices and recursive types.

Our contributions can be interpreted in three ways. *First*, from a foundational standpoint, our study of minimal session types is a conceptual contribution to the theory of behavioral types, in that we precisely identify sequencing as a source of redundancy in all preceding session types theories. As remarked in § 1, there are many session types variants, and their expressivity often comes at the price of an involved underlying theory. Our work contributes in the opposite direction, as we identified a simple yet expressive fragment of an already minimal session-typed framework [19, 20], which allows us to justify session types in terms of themselves. Understanding further the underlying theory of minimal session types (e.g., notions such as type-based compatibility) is an exciting direction for future work.

Second, our work can be seen as a new twist on Parrow's decomposition results in the *untyped* setting [24]. While Parrow's work indeed does not consider types, in fairness we must observe that when Parrow's work appeared (1996) the study of types for the π-calculus was rather incipient (for instance, binary session types appeared in 1998 [12]). That said, we should stress that our results are not merely an extension of Parrow's with session types, for types in our setting drastically narrow down the range of conceivable decompositions. Also, we exploit features not supported in [24], most notably higher-order concurrency.

Last but not least, from a practical standpoint, we believe that our approach paves a new avenue to the integration of session types in programming languages whose type systems lack sequencing, such as Go. It is natural to envision program analysis tools which, given a message-passing program that should conform to protocols specified as session types, exploit our decomposition as an intermediate step in the verification of communication correctness. Remarkably, our decomposition lends itself naturally to an implementation – in fact, we generated our examples automatically using MISTY, an associated artifact written in Haskell.

References

1 Davide Ancona, Viviana Bono, Mario Bravetti, Joana Campos, Giuseppe Castagna, Pierre-Malo Deniélou, Simon J. Gay, Nils Gesbert, Elena Giachino, Raymond Hu, Einar Broch Johnsen, Francisco Martins, Viviana Mascardi, Fabrizio Montesi, Rumyana Neykova, Nicholas Ng, Luca Padovani, Vasco T. Vasconcelos, and Nobuko Yoshida. Behavioral Types in Programming Languages. *Foundations and Trends in Programming Languages*, 3(2-3):95–230, 2016. doi:10.1561/2500000031.

2 Stephanie Balzer and Frank Pfenning. Objects as session-typed processes. In Elisa Gonzalez Boix, Philipp Haller, Alessandro Ricci, and Carlos Varela, editors, *Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control, AGERE! 2015, Pittsburgh, PA, USA, October 26, 2015*, pages 13–24. ACM, 2015. doi:10.1145/2824815.2824817.

3 Ornela Dardha, Elena Giachino, and Davide Sangiorgi. Session types revisited. In *Proc. of PPDP 2012*, pages 139–150. ACM, 2012. doi:10.1145/2370776.2370794.

4 Ornela Dardha, Elena Giachino, and Davide Sangiorgi. Session types revisited. *Inf. Comput.*, 256:253–286, 2017. doi:10.1016/j.ic.2017.06.002.

5 Romain Demangeon and Kohei Honda. Full Abstraction in a Subtyped pi-Calculus with Linear Types. In *Proc. of CONCUR 2011*, volume 6901 of *LNCS*, pages 280–296. Springer, 2011. doi:10.1007/978-3-642-23217-6_19.

6 Mariangiola Dezani-Ciancaglini and Ugo de' Liguoro. Sessions and Session Types: an Overview. In *WS-FM'09*, volume 6194 of *LNCS*, pages 1–28. Springer, 2010. URL: http://www.di.unito.it/~dezani/papers/sto.pdf.

7 Mariangiola Dezani-Ciancaglini, Elena Giachino, Sophia Drossopoulou, and Nobuko Yoshida. Bounded Session Types for Object Oriented Languages. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *Formal Methods for Components and Objects, 5th International Symposium, FMCO 2006, Amsterdam, The Netherlands, November 7-10, 2006, Revised Lectures*, volume 4709 of *Lecture Notes in Computer Science*, pages 207–245. Springer, 2006. doi:10.1007/978-3-540-74792-5_10.

8 Mariangiola Dezani-Ciancaglini, Dimitris Mostrous, Nobuko Yoshida, and Sophia Drossopoulou. Session Types for Object-Oriented Languages. In Dave Thomas, editor, *ECOOP 2006 - Object-Oriented Programming, 20th European Conference, Nantes, France, July 3-7, 2006, Proceedings*, volume 4067 of *Lecture Notes in Computer Science*, pages 328–352. Springer, 2006. doi:10.1007/11785477_20.

9 Simon Gay and Malcolm Hole. Subtyping for session types in the pi calculus. *Acta Inf.*, 42:191–225, 2005. doi:10.1007/s00236-005-0177-z.

10 Simon J. Gay, Nils Gesbert, and António Ravara. Session Types as Generic Process Types. In Johannes Borgström and Silvia Crafa, editors, *Proceedings Combined 21st International Workshop on Expressiveness in Concurrency and 11th Workshop on Structural Operational Semantics, EXPRESS 2014, and 11th Workshop on Structural Operational Semantics, SOS 2014, Rome, Italy, 1st September 2014.*, volume 160 of *EPTCS*, pages 94–110, 2014. doi:10.4204/EPTCS.160.9.

11 Simon J. Gay, Vasco Thudichum Vasconcelos, António Ravara, Nils Gesbert, and Alexandre Z. Caldeira. Modular session types for distributed object-oriented programming. In Manuel V. Hermenegildo and Jens Palsberg, editors, *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, pages 299–312. ACM, 2010. doi:10.1145/1706299.1706335.

12 Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language Primitives and Type Disciplines for Structured Communication-based Programming. In *ESOP'98*, volume 1381 of *LNCS*, pages 22–138. Springer, 1998.

13 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty Asynchronous Session Types. In *POPL'08*, pages 273–284. ACM, 2008.

14 Raymond Hu, Nobuko Yoshida, and Kohei Honda. Session-Based Distributed Programming in Java. In Jan Vitek, editor, *ECOOP 2008 - Object-Oriented Programming, 22nd European Conference, Paphos, Cyprus, July 7-11, 2008, Proceedings*, volume 5142 of *Lecture Notes in Computer Science*, pages 516–541. Springer, 2008. doi:10.1007/978-3-540-70592-5_22.

15 Hans Hüttel, Ivan Lanese, Vasco T. Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Deniélou, Dimitris Mostrous, Luca Padovani, António Ravara, Emilio Tuosto, Hugo Torres Vieira, and Gianluigi Zavattaro. Foundations of Session Types and Behavioural Contracts. *ACM Comput. Surv.*, 49(1):3, 2016. doi:10.1145/2873052.

16 Atsushi Igarashi and Naoki Kobayashi. A generic type system for the Pi-calculus. *Theor. Comput. Sci.*, 311(1-3):121–163, 2004. doi:10.1016/S0304-3975(03)00325-6.

17 Naoki Kobayashi. Type Systems for Concurrent Programs. In *Formal Methods at the Crossroads*, volume 2757 of *LNCS*, pages 439–453. Springer, 2003. doi:10.1007/978-3-540-40007-3_26.

18 Dimitrios Kouzapas, Ornela Dardha, Roly Perera, and Simon J. Gay. Typechecking protocols with Mungo and StMungo. In James Cheney and Germán Vidal, editors, *Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming, Edinburgh, United Kingdom, September 5-7, 2016*, pages 146–159. ACM, 2016. doi:10.1145/2967973.2968595.

19 Dimitrios Kouzapas, Jorge A. Pérez, and Nobuko Yoshida. On the Relative Expressiveness of Higher-Order Session Processes. In Peter Thiemann, editor, *Programming Languages and Systems - 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, volume 9632 of *Lecture Notes in Computer Science*, pages 446–475. Springer, 2016. Extended version to appear in Information and Computation (Elsevier). doi:10.1007/978-3-662-49498-1_18.

20 Dimitrios Kouzapas, Jorge A. Pérez, and Nobuko Yoshida. Characteristic bisimulation for higher-order session processes. *Acta Inf.*, 54(3):271–341, 2017. doi:10.1007/s00236-016-0289-7.

21 Julien Lange, Nicholas Ng, Bernardo Toninho, and Nobuko Yoshida. A static verification framework for message passing in Go using behavioural types. In Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman, editors, *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 1137–1148. ACM, 2018. doi:10.1145/3180155.3180157.

22 Nicholas Ng and Nobuko Yoshida. Static deadlock detection for concurrent go by global session graph synthesis. In Ayal Zaks and Manuel V. Hermenegildo, editors, *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*, pages 174–184. ACM, 2016. doi:10.1145/2892208.2892232.

23 Luca Padovani. A Simple Library Implementation of Binary Sessions. *Journal of Functional Programming*, 27, 2017. doi:10.1017/S0956796816000289.

24 Joachim Parrow. Trios in concert. In Gordon D. Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language, and Interaction, Essays in Honour of Robin Milner*, pages 623–638. The MIT Press, 2000. Online version, dated July 22, 1996, available at http://user.it.uu.se/~joachim/trios.pdf.

25 Alceste Scalas and Nobuko Yoshida. Lightweight Session Programming in Scala. In Shriram Krishnamurthi and Benjamin S. Lerner, editors, *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy*, volume 56 of *LIPIcs*, pages 21:1–21:28. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016. doi:10.4230/LIPIcs.ECOOP.2016.21.

26 Vasco T. Vasconcelos. Fundamentals of session types. *Inf. Comput.*, 217:52–70, 2012. doi:10.1016/j.ic.2012.05.002.

Julia's Efficient Algorithm for Subtyping Unions and Covariant Tuples

Benjamin Chung
Northeastern University, Boston, MA, USA
bchung@ccs.neu.edu

Francesco Zappa Nardelli
Inria of Paris, Paris, France
francesco.zappa_nardelli@inria.fr

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

Abstract

The Julia programming language supports multiple dispatch and provides a rich type annotation language to specify method applicability. When multiple methods are applicable for a given call, Julia relies on subtyping between method signatures to pick the correct method to invoke. Julia's subtyping algorithm is surprisingly complex, and determining whether it is correct remains an open question. In this paper, we focus on one piece of this problem: the interaction between union types and covariant tuples. Previous work normalized unions inside tuples to disjunctive normal form. However, this strategy has two drawbacks: complex type signatures induce space explosion, and interference between normalization and other features of Julia's type system. In this paper, we describe the algorithm that Julia uses to compute subtyping between tuples and unions – an algorithm that is immune to space explosion and plays well with other features of the language. We prove this algorithm correct and complete against a semantic-subtyping denotational model in Coq.

2012 ACM Subject Classification Theory of computation → Type theory

Keywords and phrases Type systems, Subtyping, Union types

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2019.24

Category Pearl

Supplement Material ECOOP 2019 Artifact Evaluation approved artifact available at `https://dx.doi.org/10.4230/DARTS.5.2.8`

Acknowledgements The authors thank Jiahao Chen for starting us down the path of understanding Julia, and Jeff Bezanson for coming up with Julia's subtyping algorithm. We would also like to thank Ming-Ho Yee, Celeste Hollenbeck, and Julia Belyakova for their help in preparing this paper. This work received funding from the European Research Council under the European Union's Horizon 2020 research and innovation programme (grant agreement 695412), the NSF (award 1544542 and award 1518844), the ONR (grant 503353), and the Czech Ministry of Education, Youth and Sports (grant agreement CZ.02.1.01/0.0/0.0/15_003/0000421).

1 Introduction

Union types, originally introduced by Barbanera and Dezani-Ciancaglini [2], are being adopted in mainstream languages. In some cases, such as Julia [5] or TypeScript [11], they are exposed at the source level. In others, such as Hack [8], they are only used internally as part of type inference. As a result, subtyping algorithms between union types are of increasing practical import. The standard subtyping algorithm for this combination of features has, for some time, been exponential in both time and space. An alternative algorithm, linear in space

33rd European Conference on Object-Oriented Programming (ECOOP 2019).
Editor: Alastair F. Donaldson; Article No. 24; pp. 24:1–24:15
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

but still exponential in time, has been tribal knowledge in the subtyping community [15]. In this paper, we describe and prove correct an implementation of that algorithm.

We observed the algorithm in our prior work formalizing the Julia subtyping relation [17]. There, we described Julia's subtyping relation as it arose from its decision procedure but were unable to prove it correct. Indeed, we found bugs in the Julia implementation and identified unresolved correctness issues. Contemporary work addresses some correctness concerns [3] but leaves algorithmic correctness open.

Julia's subtyping algorithm [4] is used for method dispatch. While Julia is dynamically typed, method arguments can have type annotations. These annotations allow one method to be implemented by multiple functions. At run time, Julia searches for the most specific applicable function for a given invocation. Consider these declarations of multiplication:

```
*(x::Number, r::Range)  = range(x*first(r),...)
*(x::Number, y::Number) = *(promote(x,y)...)
*(x::T, y::T) where T <: Union{Signed,Unsigned} =  mul_int(x,y)
```

The first two methods implement, respectively, multiplicaton of a range by a number and generic numeric multiplication. The third method invokes native multiplication when both arguments are either signed or unsigned integers (but not a mix of the two). Julia uses subtyping to decide which of the methods to call at any specific site. The call `1*(1:4)` dispatches to the first, `1*1.1` the second, and `1*1` the third.

Julia offers programmers a rich type language to express complex relationships in type signatures. The type language includes nominal primitive types, union types, existential types, covariant tuples, invariant parametric datatypes, and singletons. Intuitively, subtyping between types is based on semantic subtyping: the subtyping relation between types holds when the sets of values they denote are subsets of one another [5]. We write the set of values represented by a type t as $[\![t]\!]$. Under semantic subtyping, the types t_1 and t_2 are subtypes iff $[\![t_1]\!] \subseteq [\![t_2]\!]$. From this, we derive a *forall-exists* intuition for subtyping: for every value denoted on the left-hand side, there must exist some value on the right-hand side to match it, thereby establishing the subset relation. This simple intuition is, however, complicated to check algorithmically.

In this paper, we focus on the interaction of two features: covariant tuples and union types. These two kinds of type are important to Julia's semantics. Julia does not record return types, so a function's signature consists solely of the tuple of its argument types. These tuples are covariant, as a function with more specific arguments is preferred to a more generic one. Union types are widely used as shorthand to avoid writing multiple functions with the same body. As a consequence, Julia library developers write many functions with union typed arguments, functions whose relative specificity must be decided using subtyping. To prove the correctness of the subtyping algorithm, we first examine typical approaches in the presence of union types. Based on Vouillon [16], the following is a typical deductive system for subtyping union types:

$$\textsc{allexist}\ \frac{ft' <: t \qquad t'' <: t}{\texttt{Union}\{t', t''\} <: t} \qquad \textsc{existL}\ \frac{t <: t'}{t <: \texttt{Union}\{t', t''\}} \qquad \textsc{existR}\ \frac{t <: t''}{t <: \texttt{Union}\{t', t''\}} \qquad \textsc{tuple}\ \frac{t_1 <: t_1' \qquad t_2 <: t_2'}{\texttt{Tuple}\{t_1, t_2\} <: \texttt{Tuple}\{t_1', t_2'\}}$$

While this rule system might seem to make intuitive sense, it does not match the semantic intuition for subtyping. For instance, consider the following judgment:

$$\texttt{Tuple}\{\texttt{Union}\{t', t''\}, t\} \quad <: \quad \texttt{Union}\{\texttt{Tuple}\{t', t\}, \texttt{Tuple}\{t'', t\}\}$$

Using semantic subtyping, the judgment should hold. The set of values denoted by a union $[\![\texttt{Union}\{t_1, t_2\}]\!]$ is just the union of the set of values denoted by each of its members

$\llbracket t_1 \rrbracket \cup \llbracket t_2 \rrbracket$. A tuple $\texttt{Tuple}\{t_1, t_2\}$'s denotation is the set of tuples of the respective values $\{\texttt{Tuple}\{v_1, v_2\} \mid v_1 \in \llbracket t_1 \rrbracket \wedge v_2 \in \llbracket t_2 \rrbracket\}$. Therefore, the left-hand side denotes the values $\{\texttt{Tuple}\{v', v''\} \mid v' \in \llbracket t' \rrbracket \cup \llbracket t'' \rrbracket \wedge v'' \in \llbracket t \rrbracket\}$, while the right-hand side denotes $\llbracket \texttt{Tuple}\{t', t\} \rrbracket \cup \llbracket \texttt{Tuple}\{t'', t\} \rrbracket$ or equivalently $\{\texttt{Tuple}\{v', v''\} \mid v' \in \llbracket t' \rrbracket \cup \llbracket t'' \rrbracket \wedge v'' \in \llbracket t \rrbracket\}$. These sets are the same, and therefore subtyping should hold in either direction between the left- and right-hand types. However, we cannot derive this relation from the above rules. According to them, we must pick either t' or t'' on the right-hand side using ExistL or ExistR, respectively, ending up with either $\texttt{Tuple}\{\texttt{Union}\{t', t''\}, t\} <: \texttt{Tuple}\{t', t\}$ or $\texttt{Tuple}\{\texttt{Union}\{t', t''\}, t\} <: \texttt{Tuple}\{t'', t\}$. In either case, the judgment does not hold. How can this problem be solved?

Most prior work addresses this problem by normalization [2, 14, 1], rewriting all types into their disjunctive normal form, as unions of union-free types, *before* building the derivation. Now all choices are made at the top level, avoiding the structural entanglements that cause difficulties. The correctness of this rewriting step comes from the semantic denotational model, and the resulting subtyping algorithm can be proved both correct and complete. Other proposals, such as Vouillon [16] and Dunfield [7], do not handle distributivity. Normalization is used by Frisch et al.'s [9], by Pearce's flow-typing algorithm [13], and by Muehlboeck and Tate in their general framework for union and intersection types [12]. Few alternatives have been proposed, with one example being Damm's reduction of subtyping to regular tree expression inclusion [6].

However, a normalization-based algorithm has two major drawbacks: it is not space efficient, and other features of Julia render it incorrect. The first drawback is caused because normalization can create exponentially large types. Real-world Julia code [17] has types like the following whose normal form has 32,768 constituent union-free types:

```
Tuple{Tuple{Union{Int64, Bool}, Union{String, Bool}, Union{String, Bool},
            Union{String, Bool}, Union{Int64, Bool}, Union{String, Bool},
            Union{String, Bool}, Union{String, Bool}, Union{String, Bool},
            Union{String, Bool}, Union{String, Bool}, Union{String, Bool},
            Union{String, Bool}, Union{String, Bool}, Union{String, Bool}}, Int64}
```

The second drawback arises because of type-invariant constructors. For example, `Array{Int}` is an array of integers, and is not a subtype of `Array{Any}`. In conjunction with type variables, this makes normalization ineffective. Consider $\texttt{Array}\{\texttt{Union}\{t', t''\}\}$, the set of arrays whose elements are either t' or t''. It wrong to rewrite it as $\texttt{Union}\{\texttt{Array}\{t'\}, \texttt{Array}\{t''\}\}$, as this denotes the set of arrays whose elements are either all t' or t''. A weaker disjunctive normal form, only lifting union types inside each invariant constructor, is a partial solution. However, this reveals a deeper problem caused by existential types. Consider the judgment:

$$\texttt{Array}\{\texttt{Union}\{\texttt{Tuple}\{t\}, \texttt{Tuple}\{t'\}\}\} \quad <: \quad \exists T\,.\,\texttt{Array}\{\texttt{Tuple}\{T\}\}$$

It holds if the existential variable T is instantiated with $\texttt{Union}\{t, t'\}$. If types are in invariant-constructor weak normal form, an algorithm would strip off the array type constructors and proceed. However, since type constructors are invariant, the algorithm must test that both $\texttt{Union}\{\texttt{Tuple}\{t\}, \texttt{Tuple}\{t\}\} <: \texttt{Tuple}\{T\}$ and $\texttt{Tuple}\{T\} <: \texttt{Union}\{\texttt{Tuple}\{t\}, \texttt{Tuple}\{t'\}\}$ hold. The first of these can be concluded without issue, producing the constraint $\texttt{Union}\{t, t'\} <: T$. However, this constraint on T is retained for checking the reverse direction, which is where problems arise. When checks the reverse direction, the algorithm has to prove that $\texttt{Tuple}\{T\} <: \texttt{Union}\{\texttt{Tuple}\{t\}, \texttt{Tuple}\{t'\}\}$, and in turn either $T <: t$ or $T <: t'$. All of these are unprovable under the assumption that $\texttt{Union}\{t, t'\} <: T$. The key to deriving a successful judgment for this relation is to rewrite the right-to-left check into $\texttt{Tuple}\{T\} <: \texttt{Tuple}\{\texttt{Union}\{t, t'\}\}$, which is provable. This *anti-normalization* rewriting must be performed on sub-judgments of the derivation; to the best of our knowledge it is not part of any subtyping algorithm based on ahead-of-time disjunctive normalization.

Julia's subtyping algorithm avoids these problems, but it is difficult to determine how: the complete subtyping algorithm is implemented in close to two thousand lines of highly optimized C code. In this paper, we describe and prove correct only one part of that algorithm: the technique used to avoid space explosion while dealing with union types and covariant tuples. This is done by defining an iteration strategy over type terms, keeping a string of bits as its state. The space requirement of the algorithm is bounded by the number of unions in the type terms being checked.

We use a minimal type language with union, tuples, and primitive types to avoid being drawn into the vast complexity of Julia's type language. This tiny language is expressive enough to highlight the decision strategy and illustrate the structure of the algorithm. Empirical evidence from Julia's implementation suggests that this technique extends to invariant constructors and existential types [17], among others. We expect that the algorithm we describe can be leveraged in other modern language designs.

Our mechanized proof is available at: `benchung.github.io/subtype-artifact`.

2 A space-efficient subtyping algorithm

Formally, our core type language consists of binary unions, binary tuples, and primitive types ranged over by $p_1 \ldots p_n$, as shown below:

```
type typ =   Prim of int  | Tuple of typ * typ  | Union of typ * typ
```

We define subtyping for primitives as the identity, so $p_i <: p_i$.

2.1 Normalization

To explain the operation of the space-efficient algorithm, we first describe how normalization can be used as part of subtyping. Normalization rewrites types to move all internal unions to the top level. The resultant term consists of a union of union-free terms. Consider the following relation:

$$\mathtt{Union\{Tuple\{}p_1, p_2\mathtt{\}}, \mathtt{Tuple\{}p_2, p_3\mathtt{\}\}} \quad <: \quad \mathtt{Tuple\{Union\{}p_2, p_1\mathtt{\}}, \mathtt{Union\{}p_3, p_2\mathtt{\}\}}.$$

The term on the left is in normal form, but the right term needs to be rewritten as follows:

$$\mathtt{Union\{Tuple\{}p_2, p_3\mathtt{\}}, \mathtt{Union\{Tuple\{}p_2, p_2\mathtt{\}}, \mathtt{Union\{Tuple\{}p_1, p_3\mathtt{\}}, \mathtt{Tuple\{}p_1, p_2\mathtt{\}\}\}\}}$$

The top level unions can then be viewed as sets of union-free-types equivalent to each side,

$$\ell_1 = \{\mathtt{Tuple\{}p_1, p_2\mathtt{\}}, \mathtt{Tuple\{}p_2, p_3\mathtt{\}}\}$$

and

$$\ell_2 = \{\mathtt{Tuple\{}p_2, p_3\mathtt{\}}, \mathtt{Tuple\{}p_2, p_2\mathtt{\}}, \mathtt{Tuple\{}p_1, p_3\mathtt{\}}, \mathtt{Tuple\{}p_1, p_2\mathtt{\}}\}.$$

Determining whether $\ell_1 <: \ell_2$ is equivalent to checking that for each tuple component t_1 in ℓ_1, there should be an element t_2 in ℓ_2 such that $t_1 <: t_2$. Checking this final relation is straightforward, as neither t_1 nor t_2 may contain unions. Intuitively, this mirrors the rules ([AllExist], [ExistL/R], [Tuple]).

A possible implementation of normalization-based subtyping can be written compactly, as shown in the code below. The `subtype` function takes two types and returns true if they are related by subtyping. It delegates its work to `allexist` to check that all normalized terms in its first argument have a supertype, and to `exist` to check that there is at least one supertype in the second argument. The `norm` function takes a type term and returns a list of union-free terms.

```
let subtype(a:typ)(b:typ) = allexist (norm a) (norm b)

let allexist(a:list typ)(b:list typ) =
  foldl (fun acc a' => acc && exist a' b) true a

let exist(a:typ)(b:list typ) =
  foldl (fun acc b' => acc ||  a==b') false b

let rec norm = function
  | Prim i -> [Prim i]
  | Tuple t t' ->
      map_pair Tuple (cartesian_product (norm t) (norm t'))
  | Union t t' -> (norm t) @ (norm t')
```

However, as previously described, this expansion is space-inefficient. Julia's algorithm is more complicated, but avoids having to pre-compute the set of normalized types as `norm` does.

2.2 Iteration with choice strings

Given a type term such as the following,

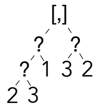
$$\texttt{Tuple\{Union\{Union\{}p_2,p_3\texttt{\}},p_1\texttt{\}},\texttt{Union\{}p_3,p_2\texttt{\}\}}$$

we want an algorithm that checks the following tuples,

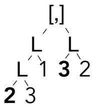
$$\texttt{Tuple\{}p_2,p_3\texttt{\}},\ \texttt{Tuple\{}p_2,p_2\texttt{\}},\ \texttt{Tuple\{}p_1,p_3\texttt{\}},\ \texttt{Tuple\{}p_1,p_2\texttt{\}},\ \texttt{Tuple\{}p_3,p_3\texttt{\}},\ \texttt{Tuple\{}p_3,p_2\texttt{\}}$$

without having to compute and store all of them ahead-of-time. This algorithm should be able to generate each tuple on-demand while still being guaranteed to explore every tuple of the original type's normal form.

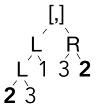
To illustrate the process that the algorithm uses to generate each tuple, consider the type term being subtyped. An alternative representation for the term is a tree, where each occurrence of a union node is a *choice point*. The following tree thus has three choice points, each represented as a ? symbol:



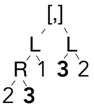
At each choice point we can go either left or right; making such a decision at each point leads to visiting one particular tuple.



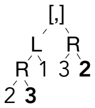
$= \texttt{Tuple\{}p_2, p_3\texttt{\}}$



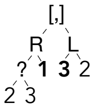
$= \texttt{Tuple\{}p_2, p_2\texttt{\}}$



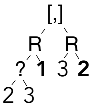
$= \texttt{Tuple\{}p_3, p_3\texttt{\}}$



$= \texttt{Tuple\{}p_3, p_2\texttt{\}}$



$= \texttt{Tuple\{}p_1, p_3\texttt{\}}$



$= \texttt{Tuple\{}p_1, p_2\texttt{\}}$

Each tuple is uniquely determined by the original type term t and a choice string c. In the above example, the result of iteration through the normalized, union-free, type terms is defined by the strings `L L L`, `L L R`, `L R L`, `L R R`, `R L`, `R R`. The length of each string is bounded by the number of unions in a term.

The iteration sequence in the above example is thus $\texttt{L L}\underline{\texttt{L}} \rightarrow \texttt{L}\underline{\texttt{L}}\texttt{ R} \rightarrow \texttt{L R}\underline{\texttt{L}} \rightarrow \underline{\texttt{L}}\texttt{ R R} \rightarrow \texttt{R}\underline{\texttt{L}} \rightarrow \texttt{R R}$, where the underlined choice is next one to be toggled in that step. Stepping from a choice string c to the next string consists of splitting c in three, $c'\,\texttt{L}\,c''$, where c' can be empty and c'' is a possibly empty sequence of `R`s. The next string is $c'\,\texttt{R}\,c_{pad}$, that is to say it retains the prefix c', toggles the `L` to an `R`, and is padded by a sequence of `L`s. The leftover tail c'' is discarded. If there is no `L` in c, iteration terminates.

One step of iteration is performed by calling the `next` function with a type term and a choice string (encoded as a `list` of `choices`); `next` either returns the next string in the sequence or `None`. Internally, it calls `step` to toggle the last `L` and shorten the string (constructing $c'\,\texttt{R}$). Then it calls on `pad` to add the trailing sequence of `L`s (constructing $c'\,\texttt{R}\,c_{pad}$).

```
type choice = L | R

let rec next(a:typ)(l:choice list) =
  match step l with
   | None -> None
   | Some(l') -> Some(fst (pad a l'))
```

The `step` function delegates the job of flipping the last occurrence of `L` to `toggle`. For ease of programming, it reverses the string so that `toggle` can be a simple recursion without an accumulator. If the given string has no `L`, then `toggle` returns empty and `step` returns `None`.

```
let step(l:choice list) =
  match rev (toggle (rev l)) with
  | [] -> None
  | hd::tl -> Some(hd::tl)

let rec toggle = function
  | [] -> []
  | L::tl -> R::tl
  | R::tl -> toggle tl
```

The `pad` function takes a type term and a choice string to be padded. It returns a pair, whose first element is the padded string and second element is the string left over from the current type. Each union encountered by `pad` in its traversal of the type consumes a character from the input string. Unions explored after the exhaustion of the original choice string are treated as if there was an `L` remaining in the choice string. The first component of the returned value is the original choice string extended with an `L` for every union encountered after exhaustion of the original.

```
let rec pad t l =
   match t,l with
   | (Prim i,l) -> ([],l)
   | (Tuple(t,t'),l) ->
      let (h,tl) = pad t l in
      let (h',tl') = pad t' tl in (h @ h',tl')
   | (Union(t,_),L::r) ->
      let (h,tl) = pad t r in (L::h,tl)
   | (Union(_,t),R::r) ->
      let (h,tl) = pad t r in (R::h,tl)
   | (Union(t,_),[]) -> (L::(fst(pad t [])),[])
```

To obtain the initial choice string, the string composed solely of Ls, it suffices to call pad with the type term under consideration and an empty list. The first element of the returned tuple is the initial choice string. For convenience, we define the function initial for this.

```
let initial(t:typ) = fst (pad t [])
```

2.3 Subtyping with iteration

Julia's subtyping algorithm visits union-free type terms using choice strings to iterate over types. The subtype function takes two type terms, a and b, and returns true if they are related by subtyping. It does so by iterating over all union-free type terms t_a in a, and checking that for each of them, there exists a union-free type term t_b in b such that $t_a <: t_b$.

```
let subtype(a:typ)(b:typ) = allexist a b (initial a)
```

The allexist function takes two type terms, a and b, and a choice string f, and returns true if a is a subtype of b for the iteration sequence starting at f. This is achieved by recursively testing that for each union-free type term in a (induced by a and the current value of f), there exists a union-free super-type in b.

```
let rec allexist(a:typ)(b:typ)(f:choice list) =
  match exist a b f (initial b) with
  | true -> (match next a f with
             | Some ns -> allexist a b ns
             | None -> true)
  | false -> false
```

Similarly, the exist function takes two type terms, a and b, and choice strings, f and e. It returns true if there exists in b, a union-free super-type of the type specified by f in a. This is done by recursively iterating through e. The determination if two terms are related is delegated to the sub function.

```
type res = NotSub | IsSub of choice list * choice list

let rec exist(a:typ)(b:typ)(f:choice list)(e:choice list) =
  match sub a b f e with
  | IsSub(_,_) -> true
  | NotSub ->
     (match next b e with
      | Some ns -> exist a b f ns
      | None -> false)
```

Finally, the sub function takes two type terms and choice strings and returns a value of type res. A res can be either NotSub, indicating that the types are not subtypes, or IsSub(_,_) when they are subtypes. If the two types are primitives, then they are only subtypes if they are equal. If the types are tuples, they are subtypes if each of their respective elements are subtypes. Note that the return type of `sub`, when successful, holds the unused choice strings for both type arguments. When encountering a union, `sub` follows the choice strings to decide which branch to take. Consider, for instance, the case when the first type term is `Union(t1,t2)` and the second is type `t`. If the first element of the choice string is an `L`, then `t1` and `t` are checked, otherwise `sub` checks `t2` and `t`.

```
let rec sub t1 t2 f e =
  match t1,t2,f,e with
  | (Prim i,Prim j,f,e) -> if i==j then IsSub(f,e) else NotSub
  | (Tuple(a1,a2), Tuple(b1,b2),f,e) ->
     (match sub a1 b1 f e with
      | IsSub(f', e') -> sub a2 b2 f' e'
      | NotSub -> NotSub)
  | (Union(a,_),b,L::f,e) -> sub a b f e
  | (Union(_,a),b,R::f,e) -> sub a b f e
  | (a,Union(b,_),f,L::e) -> sub a b f e
  | (a,Union(_,b),f,R::e) -> sub a b f e
```

2.4 Further optimization

This implementation represents choice strings as linked lists, but this design requires allocation and reversals when stepping. However, the implementation can be made more efficient by using a mutable bit vector instead of a linked list. Additionally, the maximum length of the bit vector is bounded by the number of unions in the type, so it need only be allocated once. Julia's implementation uses this efficient representation.

3 Correctness and completeness of subtyping

To prove the correctness of Julia's subtyping, we take the following general approach. We start by giving a denotational semantics for types from which we derive a definition of semantic subtyping. Then we easily prove that a normalization-based subtyping algorithm is correct and complete. This provides the general framework for which we prove two iterator-based algorithms correct. The first iterator-based algorithm explicitly includes the structure of the type in its state to guide iteration; the second is identical to that of the prior section.

The order in which choice strings iterate through a type term is determined by both the choice string and the type term being iterated over. Rather than directly working with choice strings as iterators over types, we start with a simpler structure, namely that of iterators over the trees induced by type terms. We prove correct and complete a subtyping algorithm that uses these simpler iterators. Finally, we establish a correspondence between tree iterators and choice string iterators. This concludes our proof of correctness and completeness, and details can be found in the Coq mechanization.

The denotational semantics we use for types is as follows:

$$\begin{aligned}
[\![p_i]\!] &= \{p_i\} \\
[\![\texttt{Union}\{t_1, t_2\}]\!] &= [\![t_1]\!] \cup [\![t_2]\!] \\
[\![\texttt{Tuple}\{t_1, t_2\}]\!] &= \{\texttt{Tuple}\{t_1', t_2'\} \,|\, t_1' \in [\![t_1]\!], t_2' \in [\![t_2']\!]\}
\end{aligned}$$

We define subtyping as follows: if $[\![t]\!] \subseteq [\![t']\!]$, then $t <: t'$. This leads to the definition of subtyping in our restricted language.

▶ **Definition 1.** *The subtyping relation* $t_1 <: t_2$ *holds iff* $\forall t'_1 \in [\![t_1]\!], \exists t'_2 \in [\![t_2]\!], t'_1 = t'_2$.

The use of equality for relating types is a simplification afforded by the structure of primitives.

3.1 Subtyping with normalization

The correctness and completeness of the normalization-based subtyping algorithm requires proving that the `norm` function returns all union-free type terms.

▶ **Lemma 2** (NF Equivalence). $t' \in [\![t]\!]$ *iff* $t' \in$ `norm` t.

Theorem 3 states that the `subtype` relation of Section 2.1 abides by Definition 1 because it uses `norm` to compute the set of union-free type terms for both argument types, and directly checks subtyping.

▶ **Theorem 3** (NF Subtyping). *For all* a *and* b, `subtype` a b *iff* $a <: b$.

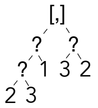
Therefore, normalization-based subtyping is correct against our definition.

3.2 Subtyping with tree iterators

Reasoning about iterators that use choice strings, as described in Section 2.2, is tricky as it requires simultaneously reasoning about the structure of the type term and the validity of the choice string that represents the iterator's state. Instead, we propose to use an intermediate data structure, called a tree iterator, to guarantee consistency of iterator state with type structure.

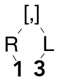
A tree iterator is a representation of the iteration state embedded in a type term. Thus a tree iterator yields a union-free tuple and can either step to a successor state or a final state. Recalling the graphical notation of Section 2.2, we can represent the state of iteration as a combination of type term and a choice or, equivalently, as a tree iterator.

Choice string:



, R L = `Tuple{`p_1, p_3`}`

Tree iterator:



= `Tuple{`p_1, p_3`}`

This structure-dependent construction makes tree iterators less efficient than choice strings. A tree iterator must have a node for each structural element of the type being iterated over, and is thus less space efficient than the simple choices-only strings. However, it is easier to prove subtyping correct for tree iterators first.

Tree iterators depend on the type term they iterate over. The possible states are `IPrim` at primitives, `ITuple` at tuples, and for unions either `ILeft` or `IRight`.

```
Inductive iter: Typ -> Set :=
| IPrim : forall i, iter (Prim i)
| ITuple : forall t1 t2, iter t1 -> iter t2 -> iter (Tuple t1 t2)
| ILeft : forall t1 t2, iter t1 -> iter (Union t1 t2)
| IRight : forall t1 t2, iter t2 -> iter (Union t1 t2).
```

The `next` function for tree iterators steps in depth-first, right-to-left order. There are four cases to consider:

- A primitive has no successor.
- A tuple steps its second child; if that has no successor step, then it steps its first child and resets the second child.

- An `ILeft` tries to step its child. If it has no successor, then the `ILeft` becomes an `IRight` with a newly initialized child corresponding to the right child of the union.
- An `IRight` also tries to step its child, but is final if its child has no successor.

```
Fixpoint next(t:Typ)(i:iter t): option(iter t) := match i with
  | IPrim _ => None
  | ITuple t1 t2 i1 i2 =>
    match (next t2 i2) with
    | Some i' => Some(ITuple t1 t2 i1 i')
    | None =>
      match (next t1 i1) with
      | Some i' => Some(ITuple t1 t2 i' (start t2))
      | None => None
      end
    end
  | ILeft t1 t2 i1 =>
    match (next t1 i1) with
    | Some(i') => Some(ILeft t1 t2 i')
    | None => Some(IRight t1 t2 (start t2))
    end
  | IRight t1 t2 i2 =>
    match (next t2 i2) with
    | Some(i') => Some(IRight t1 t2 i')
    | None => None
    end
  end.
```

An induction principle for tree iterators is needed to reason about all iterator states for a given type. First, we show that iterators eventually reach a final state. This is done with a function `inum`, which assigns natural numbers to each state. It simply counts the number of remaining steps in the iterator. To count the total number of union-free types denoted by a type, we use the `tnum` helper function.

```
Fixpoint tnum(t:Typ):nat :=
  match t with
  | Prim i => 1
  | Tuple t1 t2 => tnum t1 * tnum t2
  | Union t1 t2 => tnum t1 + tnum t2
  end.

Fixpoint inum(t:Typ)(ti:iter t):nat :=
  match ti with
  | IPrim i => 0
  | ITuple t1 t2 i1 i2 => inum t1 i1 * tnum t2 + inum t2 i2
  | IUnionL t1 t2 i1 => inum t1 i1 + tnum t2
  | IUnionR t1 t2 i2 => inum t2 i2
  end.
```

This function then lets us define the key theorem needed for the induction principle. At each step, the value of `inum` decreases by 1, and since it cannot be negative, the iterator must therefore reach a final state.

▶ **Lemma 4** (Monotonicity). *If* $\mathtt{next}\ t\ it = it'$ *then* $\mathtt{inum}\ t\ it = 1 + \mathtt{inum}\ t\ it'$.

It is now possible to define an induction principle over `next`. By monotonicity, `next` eventually reaches a final state. For any property of interest, if we prove that it holds for the final state and for the induction step, we can prove it holds for every state for that type.

▶ **Theorem 5** (Tree Iterator Induction)**.** *Let P be any property of tree iterators for some type t. Suppose P holds for the final state, and whenever P holds for a successor state it then it holds for its precursor it' where* `next` *t $it' = it$. Then P holds for every iterator state over t.*

Now, we can prove correctness of the subtyping algorithm with tree iterators. We implement subtyping with respect to choice strings in the Coq implementation in a two-stage process. First, we compute the union-free types induced by the iterators over their original types using `here`. Second, we decide subtyping between the two union-free types in `ufsub`. The function `here` walks the given iterator, producing a union-free type mirroring its state. To decide subtyping between the resulting union-free types, `ufsub` checks equality between `Prims` and recurses on the elements of `Tuples`, while returning false for all other types. Since `here` will never produce a union type, the case of `ufsub` for them is irrelevant, and is false by default.

```
Fixpoint here(t:Typ)(i:iter t):Typ:=
  match i with
  | IPrim i => Prim i
  | ITuple t1 t2 p1 p2 =>
    Tuple (here t1 p1) (here t2 p2)
  | ILeft t1 t2 pl => (here t1 pl)
  | IRight t1 t2 pr => (here t2 pr)
  end.
```

```
Fixpoint ufsub(t1 t2:Typ) :=
  match (t1, t2) with
  | (Prim p, Prim p') => p==p'
  | (Tuple a a', Tuple b b') =>
     ufsub a b && ufsub a' b'
  | (_, _) => false
  end.
```

```
Definition sub (a b:Typ) (ai:iter a) (bi:iter b) :=
    ufsub (here a ai) (here b bi).
```

This version of `sub` differs from the algorithmic implementation to ensure that recursion is well founded. The previous version of `sub` was, in the case of unions, decreasing on alternating arguments when unions were found on either of the sides. In contrast, the proof's version of `sub` applies the choice string to each side first using `here`, a strictly decreasing function that recurs structurally on the given type. This computes the union-free type induced by the iterator applied to the current type. The algorithm then checks subtyping between the resultant union-free types, which is entirely structural. These implementations are equivalent, as they both apply the given choice strings at the same places while computing subtyping; however, the proof version separates choice string application while the implementation intertwines it with the actual subtyping decision.

Versions of `exist` and `allexist` that use tree iterators are given next. They are similar to the string iterator functions of Section 2.2. `exist` tests if the subtyping relation holds in the context of the current iterator states for both sides. If not, it recurs on the next state. Similarly, `allexist` uses its iterator for a in conjunction with `exist` to ensure that the current left-hand iterator state has a matching right-hand state. We prove termination of both using Lemma 4.

```
Definition subtype(a b:Typ) = allexist a b (initial a)

Program Fixpoint allexist (a b:typ)(ia:iter a) {measure(inum ia)} =
   exists a b ia (initial b) &&
      (match next a ia with
       | Some(ia') => allexist a b ia'
       | None => true).

Program Fixpoint exist(a b:typ)(ia:iter a)(ib:iter b)
                                          {measure(inum ib)} =
   subtype a b ia ib  ||
      (match next b ib with
        | Some(ib') => exist a b ia ib'
        | None => false).
```

The denotation of a tree iterator state $\mathcal{R}(i)$ is the set of states that can be reached using `next` from i. Let $a(i)$ indicate the union-free type produced from the type a at i, and $|i|_a$ is the set $\{a(i') \mid i' \in \mathcal{R}(i)\}$, the union-free types that result from states in the type a reachable by i. This lets us prove that the set of types corresponding to states reachable from the initial state of an iterator is equal to the set of states denoted by the type itself.

▶ **Lemma 6** (Initial equivalence). $|\mathtt{initial}\ a|_a = \llbracket a \rrbracket$.

Next, Theorem 5 allows us to show that `exists` of a, b, with i_a and i_b tries to find an iterator state i'_b starting from i_b such that $b(i'_b) = a(i_a)$. The desired property trivially holds when $|i_b|_b = \emptyset$, and if the iterator can step then either the current union-free type is satisfying or we defer to the induction hypothesis.

▶ **Theorem 7.** *$\mathtt{exist}\, a\ b\ i_a\ i_b$ holds iff $\exists t \in |i_b|_b, a(i_a) = t$.*

We can then appeal to both Theorem 7 and Lemma 6 to show that `exist` $a\ b\ i_a$ (`initial` b) finds a satisfying union-free type on the right-hand side if it exists in $\llbracket b \rrbracket$. Using this, we can then use Theorem 5 in an analogous way to `exist` to show that `allexist` is correct up to the current iterator state.

▶ **Theorem 8.** *$\mathtt{allexist}\, a\ b\ i_a$ holds iff $\forall a' \in |i_a|_a, \exists b' \in \llbracket b \rrbracket, a' = b'$.*

Finally, we can appeal to Theorem 8 and Lemma 6 again to show correctness of the algorithm.

▶ **Theorem 9.** *$\mathtt{subtype}\, a\ b$ holds iff $\forall a' \in \llbracket a \rrbracket, \exists b' \in \llbracket b \rrbracket, a' = b'$.*

3.3 Subtyping with choice strings

We prove the subtyping algorithm using choice strings correct and complete. We start by showing that iterators over choice strings simulate tree iterators. This lets us prove that the choice string based subtyping algorithm is correct by showing that the iterators at each step are equivalent. To relate tree iterators to choice string iterators, we use the `itp` function, which traverses a tree iterator state and linearizes it, producing a choice string using depth-first search.

```
Fixpoint itp{t:Typ}(it:iter t):choice list :=
   match it with
   | IPrim _ => nil
   | ITuple t1 t2 it1 it2 => (itp t1 it1)++(itp t2 it2)
   | ILeft t1 _ it1 => Left::(itp t1 it1)
   | IRight _ t2 it1 => Right::(itp t2 it1)
   end.
```

Next, we define an induction principle over choice strings by way of linearized tree iterators. The `next` function in Section 2.2 works by finding the last `L` in the choice string, turning it into an `R`, and replacing the rest with `L`s until the type is valid. If we use `itp` to translate both the initial and final states for a valid `next` step of a tree iterator, we see the same structure.

▶ **Lemma 10** (Linearized Iteration). *For some type t and tree iterators $it\ it'$, if $\mathtt{next}\, t\ it = it'$, there exists some prefix c', an initial suffix c'' made up of `R`s, and a final suffix c''' consisting of `L`s such that $\mathtt{itp}\ t\ it = c'\, \mathtt{Left}\, c''$ and $\mathtt{itp}\ t\ it' = c'\, \mathtt{Right}\, c'''$.*

We can then prove that stepping a tree iterator state is equivalent to stepping the linearized versions of the state using the choice string `next` function.

▶ **Lemma 11** (Step Equivalence). *If it and it' are tree iterator states and $\mathtt{next}\ it = it'$, then $\mathtt{next}(\mathtt{itp}\ it) = (\mathtt{itp}\ it')$.*

The initial state of a tree iterator linearizes to the initial state of a choice string iterator.

▶ **Lemma 12** (Initial Equivalence). *$\mathtt{itp}(\mathtt{initial}\ t) = \mathtt{pad}\ t\ \mathtt{[]}$.*

The functions `exist` and `allexist` for choice string based iterators are identical to those for tree iterators (though using choice string iterators internally), and `sub` is as described in Section 2.2. The correctness proofs for the choice string subtype decision functions use the tree iterator induction principle (Theorem 5), and are thus in terms of tree iterators. By Lemma 11, however, each step that the tree iterator takes will be mirrored precisely by `itp` into choice strings. Similarly, the initial states are identical by Lemma 12. As a result, the sequence of states checked by each of the iterators is equivalent with `itp`.

▶ **Lemma 13.** *$\mathtt{exist}\ a\ b\ (\mathtt{itp}\ i_a)\ (\mathtt{itp}\ i_b)$ holds iff $\exists t \in |i_b|_b, a(ia) = t$.*

With the correctness of `exist` following from the tree iterator definition, we can apply the same proof methodology to show that `allexist` is correct. In order to do so, we instantiate Lemma 13 with Lemma 6 and Lemma 12 to show that if `exist a b` (`itp ia`) (`pad t []`) then $\exists \mathtt{t} \in [\![\mathtt{b}]\!], \mathtt{a}(\mathtt{ia}) = \mathtt{t}$, allowing us to check each of the exists cases while establishing the forall-exists relationship.

▶ **Lemma 14.** *$\mathtt{allexist}\ a\ b\ (\mathtt{itp}\ i_a)$ holds iff $\forall a' \in |i_a|_a, \exists b' \in [\![b]\!], a' = b'$.*

We can then instantiate Lemma 14 with Lemma 12 and Lemma 6 to show that `allexist` for choice strings ensures that the forall-exists relation holds.

▶ **Theorem 15.** *$\mathtt{allexist}\ a\ b\ (\mathtt{pad}\ t\ \mathtt{[]})$ holds iff $\forall a' \in [\![a]\!], \exists b' \in [\![b]\!], a' = b'$.*

Finally, we can prove that subtyping is correct using the choice string algorithm.

▶ **Theorem 16.** *$\mathtt{subtype}\ a\ b$ holds iff $\forall a' \in [\![a]\!], \exists b' \in [\![b]\!], a' = b'$.*

Thus, we can correctly decide subtyping with distributive unions and tuples using the choice string based implementation of iterators.

4 Complexity

The worst-case time complexity of Julia's subtyping algorithm and normalization-based approaches is determined by the number of terms that could exist in the normalized type. In the worst case, there are 2^n union-free tuples in the fully normalized version of a type that has n unions. Each of those tuples must always be explored. As a result, both algorithms have worst-case $O(2^n)$ time complexity. The approaches differ, however, in space complexity. The normalization approach computes and stores each of the exponentially many alternatives, so it also has $O(2^n)$ space complexity. However, Julia need only store the choice made at each union, thereby offering $O(n)$ space complexity.

Julia's algorithm improves best-case time performance. Normalization always experiences worst-case time and space behavior as it has to precompute the entire normalized type. Julia's iteration-based algorithm can discover the relation between types early. In practice, many queries are of the form $\mathit{uft} <: \mathit{union}(t_1 ... t_n)$, where uft is an already union-free tuple. As a result, all that Julia needs to do is find one matching tuple in $t_1 ... t_n$, which can be done sequentially without needing explicit enumeration.

5 Future work

We plan to handle additional features of Julia. Our next steps will be subtyping for primitive types, existential type variables, and invariant constructors. Adding subtyping to primitive types would be the simplest change. The challenge is how to retain completeness, as a primitive subtype heirarchy and semantic subtyping have undesirable interactions. For example, if the primitive subtype hierarchy contains only the relations $p_2 <: p_1$ and $p_3 <: p_1$, then is p_1 a subtype of `Union`$\{p_2, p_3\}$? In a semantic subtyping system, they are, but this requires changes both to the denotational framework and the search space of the iterators. Existential type variables create substantial new complexities in the state of the algorithm. No longer is the state solely restricted to that of the iterators being attempted; now, the state includes variable bounds that are accumulated as the algorithm compares types to type variables. As a result, correctness becomes a much more complex contextually linked property to prove. Finally, invariant type constructors induce contravariant subtyping, which when combined with existential variables may create cycles within the subtyping relation.

6 Conclusion

It is likely that subtyping with unions and tuples is always going to be exponential time, as subtyping of regular expression types have been proven to be EXPTIME-complete [10]. However, it need not take exponential space to decide subtyping: we have described and proven correct a subtyping algorithm for covariant tuples and unions that uses iterators instead of normalization. This algorithm uses linear space and allows common patterns, such as testing if a tuple of primitives is a subtype of a tuple of unions, to be handled as a special case of the subtyping algorithm. Finally, based on Julia's experience with the algorithm, we think that it can generalize to rich type languages; Julia supports bounded polymorphism and invariant constructors enabled in part by its use of this algorithm.

References

1 Alexander Aiken and Brian R. Murphy. Implementing regular tree expressions. In *Functional Programming Languages and Computer Architecture FPCA*, 1991. doi:10.1007/3540543961_21.

2 Franco Barbanera and Mariangiola Dezani-Ciancaglini. Intersection and union types. In *Theoretical Aspects of Computer Software TACS*, 1991. doi:10.1007/3-540-54415-1_69.

3 Julia Belyakova. Decidable Tag-Based Semantic Subtyping for Nominal Types, Tuples, and Unions. In *Proceedings of the 21st Workshop on Formal Techniques for Java-like Programs FTFJP*, 2019.

4 Jeff Bezanson. *Abstraction in technical computing.* PhD thesis, Massachusetts Institute of Technology, 2015. URL: http://dspace.mit.edu/handle/1721.1/7582.

5 Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. Julia: A Fresh Approach to Numerical Computing. *SIAM Review*, 59(1), 2017. doi:10.1137/141000671.

6 Flemming M. Damm. Subtyping with Union Types, Intersection Types and Recursive Types. In *Theoretical Aspects of Computer Software TACS*, 1994. doi:10.1007/3-540-57887-0_121.

7 Joshua Dunfield. Elaborating intersection and union types. *J. Funct. Program.*, 2014. doi:10.1017/S0956796813000270.

8 Facebook. Hack. https://hacklang.org/.

9 Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. Semantic subtyping: Dealing set-theoretically with function, union, intersection, and negation types. *J. ACM*, 55(4), 2008. doi:10.1145/1391289.1391293.

10 Haruo Hosoya, Jérôme Vouillon, and Benjamin C. Pierce. Regular Expression Types for XML. *ACM Trans. Program. Lang. Syst.*, 2005.

11 Microsoft. Typescript Language Specification. URL: `https://github.com/Microsoft/TypeScript/blob/master/doc/spec.md`.

12 Fabian Muehlboeck and Ross Tate. Empowering Union and Intersection Types with Integrated Subtyping. *Proc. ACM Program. Lang.*, 2(OOPSLA), 2018.

13 David J. Pearce. Sound and Complete Flow Typing with Unions, Intersections and Negations. In *Verification, Model Checking, and Abstract Interpretation VMCAI*, 2013. `doi:10.1007/978-3-642-35873-9_21`.

14 Benjamin Pierce. Programming with intersection types, union types, and polymorphism. Technical Report CMU-CS-91-106, Carnegie Mellon University, 1991.

15 Ross Tate. personal communication.

16 Jerome Vouillon. Subtyping Union Types. In *Computer Science Logic (CSL)*, 2004. `doi:10.1007/978-3-540-30124-0_32`.

17 Francesco Zappa Nardelli, Julia Belyakova, Artem Pelenitsyn, Benjamin Chung, Jeff Bezanson, and Jan Vitek. Julia subtyping: a rational reconstruction. *Proc. ACM Program. Lang.*, 2(OOPSLA), 2018. `doi:10.1145/3276483`.

Finally, a Polymorphic Linear Algebra Language

Amir Shaikhha
Department of Computer Science, University of Oxford, UK
amir.shaikhha@cs.ox.ac.uk

Lionel Parreaux
DATA Lab, EPFL, Lausanne, Switzerland
lionel.parreaux@epfl.ch

Abstract

Many different data analytics tasks boil down to linear algebra primitives. In practice, for each different type of workload, data scientists use a particular specialised library. In this paper, we present PILATUS, a polymorphic iterative linear algebra language, applicable to various types of data analytics workloads. The design of this domain-specific language (DSL) is inspired by both mathematics and programming languages: its basic constructs are borrowed from abstract algebra, whereas the key technology behind its polymorphic design uses the *tagless final* approach (a.k.a. polymorphic embedding/object algebras). This design enables us to change the behaviour of arithmetic operations to express matrix algebra, graph algorithms, logical probabilistic programs, and differentiable programs. Crucially, the polymorphic design of PILATUS allows us to use multi-stage programming and rewrite-based optimisation to recover the performance of specialised code, supporting fixed sized matrices, algebraic optimisations, and fusion.

2012 ACM Subject Classification Software and its engineering → Domain specific languages; Computing methodologies → Linear algebra algorithms; Mathematics of computing → Automatic differentiation; Mathematics of computing → Graph algorithms; Theory of computation → Probabilistic computation

Keywords and phrases Linear Algebra, Domain-Specific Languages, Tagless Final, Polymorphic Embedding, Object Algebra, Multi-Stage Programming, Graph Processing, Probabilistic Programming, Automatic Differentiation

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2019.25

Category Pearl

Acknowledgements The authors would like to thank Jeremy Gibbons and Oleg Kiselyov for their helpful comments on draft versions of this paper. The first author was partially supported by EPFL during the preparation of this paper.

1 Introduction

It is well-known that many problems can be formulated using linear algebra primitives. These problems come from various data analytics domains including machine learning, statistical data analytics, signal processing, graph processing, computer vision, and robotics.

Despite the fact that all these workloads could use a standard unified linear algebra library, in practice many different specialised libraries are developed and used for each of these workload types [9]. This is mainly due to the performance-critical nature of such data analytics workloads: in order to satisfy their performance requirement, such workloads use hand-tuned specialised libraries implemented using either general-purpose or specialised domain-specific programming languages.

In this paper, we demonstrate the PILATUS language (Polymorphic Iterative Linear Algebra, Typed, Universal, and Staged). PILATUS is a *polymorphic* domain-specific language (DSL), in the sense that it can support various workloads, such as standard iterative linear algebra tasks, graph processing algorithms, logical probabilistic programs, and linear algebra

33rd European Conference on Object-Oriented Programming (ECOOP 2019).
Editor: Alastair F. Donaldson; Article No. 25; pp. 25:1–25:29
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

programs relying on automatic differentiation. By default, this polymorphic nature causes a significant performance overhead. We demonstrate how to remove this overhead by using safe high-level meta-programming and compilation techniques, and more specifically multi-stage programming (MSP, or *staging*) [66, 65].

This paper uses the tagless final approach [36, 10] (also known as polymorphic embedding [30] and object algebras [48]) in order to embed [32] the PILATUS DSL in the Scala programming language. This technique allows embedding an *object* language in a *host* language in a type-safe manner. In addition, this approach allows multiple semantics for the embedded DSL (EDSL). Based on this feature and by carefully choosing the abstractions involved in defining PILATUS (such as semi-ring/ring, module, and linear map structures), we provide several evaluation semantics. More specifically, we allow several variants of a linear algebra language, such as: a standard matrix algebra language, a graph language for expressing all-pairs reachability and shortest path problems, a logical probabilistic programming language, and a differentiable programming language. The polymorphic aspect of PILATUS is also essential for the seamless application of staging, and to express different optimised staged variants: fixed size matrices, deforestation [67, 25, 63, 13], and algebraic optimisations.[1]

Next, we motivate the need for PILATUS (Section 2), and we make the following contributions:

- We present PILATUS, a polymorphic EDSL in Section 3. This DSL uses the notion of semi-rings and rings (Section 3.2) in order to define operations on each individual element of a vector and a matrix. Furthermore, PILATUS uses the notion of pull arrays (Section 3.5) for defining a collection (or array) of elements.
- We present four different languages that are implemented by providing a concrete interpreter for PILATUS in Section 5: (1) a standard matrix algebra language (Section 5.1); (2) a graph DSL (Section 5.2); (3) a logical probabilistic linear algebra language (Section 5.3); and (4) a differentiable linear algebra language (Section 5.4).
- We present our use of multi-stage programming to improve the performance of PILATUS programs by creating a staged language (Section 6.2) for fixed size matrices (Section 6.5), performing algebraic optimisations (Section 6.4), and performing fusion (Section 6.6).
- We show the impact of using multi-stage programming on the performance of applications written using PILATUS in Section 7. Overall, the implementation of PILATUS consists of around 400 LoC supporting all the features presented in this paper. PILATUS uses the Squid [51] type-safe meta-programming framework for its multi-stage programming facilities, which is the only external library dependency.

Finally, we present the related work in Section 8 and conclude the paper in Section 9.

2 Motivation

Apart from standard matrix algebra tasks, many numerical workloads in various domains can be expressed using linear algebra primitives [17]. Among such examples are various graph

[1] We used Scala as the implementation language for PILATUS, but other programming languages with support for lambda expressions and multi-stage programming could be used as well; most of the techniques presented in this paper can also be implemented in Haskell, OCaml, and Java for example. For expressing rewrite-based optimisations, either the multi-stage programming framework should support code inspection (as is the case with Squid [51], which we use), or the developer is responsible for implementing/extending the intermediate representations (as with frameworks like LMS [57]).

problems such as reachability and shortest path. Figure 1 shows as example the reachability problem on both deterministic and probabilistic graphs.

Despite the expressiveness of linear algebra, there are many different libraries specialized for each particular data analytics task. This is because of two main reasons. First, most existing linear algebra libraries do not define the interfaces for extending their usage for the problems in other domains. Second, despite some efforts on providing abstract and extensible linear algebra libraries [17], such analytical tasks are performance critical. As a result, there should be hand-tuned and specialized libraries for each particular task. As an example, for graph problems, rather than having the linear-algebra-based solutions presented in Figure 1, the library developers prefer to provide specialized graph libraries for performance reasons.

This paper aims to solve both these issues by combining ideas from mathematics and programming languages. The first issue is tackled by defining a polymorphic linear algebra language by using abstractions from abstract algebra, including the ring, module, and linear map structures for expressing scalar values, a vector of values, and a matrix of values, respectively. Furthermore, for implementing these abstract interfaces, we use the tagless-final approach [10, 36], a well-known technique from the programming language community.

The examples of Figure 1 show matrices of elements of various types, for which the addition and multiplication operations can be assigned various meanings. Figure 1a shows the usage of linear algebra primitives for expressing graph reachability problems. To do so, the addition and multiplication operators are instantiated to boolean disjunction and conjunction, respectively. For expressing the reachability problems on probabilistic graphs, these two operators are instantiated with the disjunction and conjunction on boolean distributions, as shown in Figure 1b. Finally, Figure 1c shows the process of computing the derivative of an example matrix expression with respect to a given variable. To do so, each element of the matrix should be represented as a pair of numbers, known as *dual numbers*, where the first component is the actual value of that expression and the second component is the value of its derivative with respect to the given variable. As an example, the dual number representation for the element of the 2^{nd} row and 3^{rd} column is represented as $3 \triangleright 2$, meaning that the actual value of this element at $x = 2$ is $2x - 1 = 3$, whereas its derivative value is $(2x - 1)' = 2$. Similarly, the addition and multiplication operators are instantiated with the corresponding ones operating on dual numbers, which implement the derivative rules.

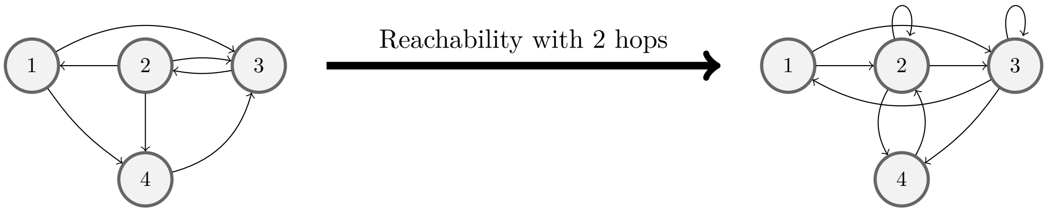
All of these use cases can be easily represented as Pilatus programs, parameterized over the meaning one wants to use for a particular domain.

The second issue is the performance overhead caused by the polymorphic nature of the language, due to the abstractions introduced in order to solve the first issue. We use multi-stage programming (also known as *staging*) to compile away the overhead corresponding to these abstractions. Moreover, by using a staging framework with support for rewriting, we can also implement algebraic optimization rules for further improving performances.

Next, we give more details on the design of Pilatus.

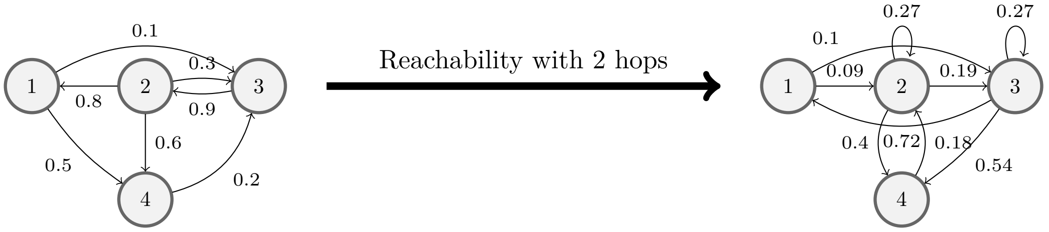
3 Pilatus Design

In this section, we first give an overview of the tagless final approach. Then, we define the polymorphic interface for the semi-ring and ring structures. Afterwards, we show an abstract interface for vectors and matrices using the mathematical notions of modules and linear maps. Finally, we define the interface for a functional encoding of an array of elements and control-flow constructs.



$$\begin{bmatrix}0&0&1&1\\1&0&1&1\\0&1&0&0\\0&0&1&0\end{bmatrix}\otimes\begin{bmatrix}0&0&1&1\\1&0&1&1\\0&1&0&0\\0&0&1&0\end{bmatrix}=\begin{bmatrix}0&1&1&0\\0&1&1&1\\1&0&1&1\\0&1&0&0\end{bmatrix}\quad M_{2,2}=(1\wedge0)\vee(0\wedge0)\vee(1\wedge1)\vee(1\wedge0)=1$$

(a) The reachability problem in a graph can be expressed using matrix-matrix multiplication of the adjacency matrix of a graph. Instead of using the standard addition operator, here we use the boolean disjunction, and instead of the multiplication operator, we use the boolean conjunction.



$$\begin{bmatrix}0&0&0.1&0.5\\0.8&0&0.3&0.6\\0&0.9&0&0\\0&0&0.2&0\end{bmatrix}\otimes\begin{bmatrix}0&0&0.1&0.5\\0.8&0&0.3&0.6\\0&0.9&0&0\\0&0&0.2&0\end{bmatrix}=\begin{bmatrix}0&0.09&0.1&0\\0&0.27&0.19&0.4\\0.72&0&0.27&0.54\\0&0.18&0&0\end{bmatrix}$$

(b) The reachability problem in a probabilistic graph can also be expressed using matrix-matrix multiplication of its adjacency matrix. Each element of the adjacency matrix represents the presence of a node with probability p. The addition and multiplication operators correspond to disjunction and conjunction of two boolean distributions, respectively.

$$f(x)=\begin{bmatrix}0&0&1&x+3\\8&0&2x-1&6\\0&3x+3&0&0\\0&0&x&0\end{bmatrix}^2=\begin{bmatrix}0&3x+3&x^2+3x&0\\0&6x^2+3x-3&6x+8&8x+24\\24x+24&0&6x^2+3x-3&18x+18\\0&3x^2+3x&0&0\end{bmatrix}$$

$$f'(x)=\begin{bmatrix}0&3&2x+3&0\\0&12x+3&6&8\\24&0&12x+3&18\\0&6x+3&0&0\end{bmatrix}\quad f(2)=\begin{bmatrix}0&9&10&0\\0&27&20&40\\72&0&27&54\\0&18&0&0\end{bmatrix}\quad f'(2)=\begin{bmatrix}0&3&7&0\\0&27&6&8\\24&0&27&18\\0&15&0&0\end{bmatrix}$$

$$\begin{bmatrix}0\triangleright0&0\triangleright0&1\triangleright0&5\triangleright1\\8\triangleright0&0\triangleright0&3\triangleright2&6\triangleright0\\0\triangleright0&9\triangleright3&0\triangleright0&0\triangleright0\\0\triangleright0&0\triangleright0&2\triangleright1&0\triangleright0\end{bmatrix}\otimes\begin{bmatrix}0\triangleright0&0\triangleright0&1\triangleright0&5\triangleright1\\8\triangleright0&0\triangleright0&3\triangleright2&6\triangleright0\\0\triangleright0&9\triangleright3&0\triangleright0&0\triangleright0\\0\triangleright0&0\triangleright0&2\triangleright1&0\triangleright0\end{bmatrix}=\begin{bmatrix}0\triangleright0&9\triangleright3&10\triangleright7&0\triangleright0\\0\triangleright0&27\triangleright27&20\triangleright6&40\triangleright8\\72\triangleright24&0\triangleright0&27\triangleright27&54\triangleright18\\0\triangleright0&18\triangleright15&0\triangleright0&0\triangleright0\end{bmatrix}$$

(c) The derivative of a matrix with respect to the variable x can also be expressed using linear algebra operations. The dual number technique represents each element of a matrix as the pair $v \triangleright d$ of the actual value v and the value of its derivative d. Accordingly, the addition and multiplication operators are the corresponding ones on dual numbers.

Figure 1 Example of problems expressed using different interpretations of linear algebra primitives.

3.1 Tagless Final

Tagless final [10, 36] (also known as polymorphic embedding [30] and object algebras [48] in the context of object-oriented programming languages) is a type-safe approach for *embedding* [32] domain-specific languages. This approach solves the expression problem [68] by encoding each DSL construct as a separate function, and leaving their interpretation abstract.

There are different ways of implementing this approach: (1) in languages like Haskell, one can use type classes [10, 36]; (2) in OCaml, one can use the module system [10, 37]; (3) in languages like Java, one can use the object-oriented features [48]; and (4) in Scala one can use either type classes or mixin composition (also known as the cake pattern) [30, 57].

In this paper, we follow the approach based on type classes. Consider a DSL with two constructs, one for creating an integer literal, and the other for adding two terms. The tagless final interface for this DSL is as follows:

```
trait SimpleDSL[Repr] {
  def lit(i: Int): Repr
  def add(a: Repr, b: Repr): Repr
}
```

The code above defines a *trait* (similar to an *interface* in Java or a *module signature* in ML). The `SimpleDSL` trait is parameterised with a `Repr` type, which is the type of the objects manipulated by the DSL. This trait contains one abstract method for each constructs of the DSL, here `lit` and `add`.

For convenience, we also typically define free-standing functions for writing programs in the DSL while omitting the particular DSL implementation used:

```
def lit[Repr](i: Int)          (implicit dsl: SimpleDSL[Repr]): Repr = dsl.lit(i)
def add[Repr](a: Repr, b: Repr)(implicit dsl: SimpleDSL[Repr]): Repr = dsl.add(a, b)
```

These functions require an *implicit* instance of the `SimpleDSL` trait, and redirect to the implementations of the corresponding methods in that instance. In Scala, implicit parameters need not be specified by users at each call site; indeed, they can be *filled in* automatically by the compiler, based on their expected type. Implicits are the mechanism used to implement type classes in Scala [49].

One can then define generic programs in the DSL, as follows:

```
def myProgram[Repr](implicit dsl: SimpleDSL) = add(lit(2), lit(3))
```

Which can also be written using the following shorthand syntax:

```
def myProgram[Repr: SimpleDSL] = add(lit(2), lit(3))
```

Then, one can specify a particular evaluation semantic for this program. As an example, the following type class instance defines an evaluator/interpreter for `SimpleDSL`:

```
implicit object SimpleDSLInter extends SimpleDSL[Int] {
  def lit(i: Int): Int = i
  def add(a: Int, b: Int): Int = a + b
}
```

Evaluating the example program above in the REPL with this evaluation semantics, which is automatically picked up by the compiler based on the requested type `Int`, results in:

```
scala> myProgram[Int]
result: Int = 5
```

```
trait SemiRing[R] {
  def add(a: R, b: R): R
  def mult(a: R, b: R): R
  def one: R
  def zero: R
}
trait Ring[R] extends SemiRing[R] {
  def neg(a: R): R
  def sub(a: R, b: R): R = add(a, neg(b))
}

object Pilatus {
  def add[R](a: R, b: R)(implicit sr: SemiRing[R]): R = sr.add(a, b)
  // ... other boilerplate methods elided for brevity
}
```

Figure 2 The tagless final interface for semi-rings and rings.

Rather than directly *evaluating* DSL programs, one can also represent the programs as strings. Below is a type class instance that *stringifies* programs in our DSL:[2]

```
  implicit object SimpleDSLStringify extends SimpleDSL[String] {
    def lit(i: Int): String = i.toString
    def add(a: String, b: String): String = s"$a + $b"
  }
```

Evaluating the same program with the stringification evaluation semantic results in:

```
  scala> myProgram[String]
  result: String = "2 + 3"
```

Pilatus defines a separate type class for each category of the language constructs (e.g., semi-rings, rings, modules, linear maps, etc.), as we show next. We will introduce different evaluation semantics for this DSL by providing type class instances. These evaluation semantics are both interpretation-based (cf. Section 5) and compilation-based (cf. Section 6).

3.2 Semi-Ring and Ring

A semi-ring is defined as a set of numerical values R, with two binary operators $+$ and $\times$, and two elements 0 (additive identity) and 1 (multiplicative identity), such that for all elements a, b, and c in R the following properties hold:

- $a + 0 = a$
- $a + b = b + a$
- $(a + b) + c = a + (b + c)$
- $a \times 1 = 1 \times a = a$
- $a \times 0 = 0 \times a = 0$
- $(a \times b) \times c = a \times (b \times c)$
- $a \times (b + c) = (a \times b) + (a \times c)$
- $(a + b) \times c = (a \times c) + (b \times c)$

[2] String interpolation syntax `s"...$x..."` is equivalent to `"..."+ x + "..."`.

A ring is a semi-ring with an additional additive inverse operator $(-)$ such that for all elements a in R, $a + (-a) = 0$. The binary operator for subtraction can be easily defined as $a - b = a + (-b)$.

The tagless final encoding of semi-rings and rings is shown in Figure 2. There are six DSL constructs corresponding to addition, multiplication, negation, subtraction, one, and zero. These methods are redirected to the implementation of the corresponding operations of the `SemiRing` and `Ring` type classes. The implementation of the methods of these type classes are left abstract. These definitions will be given by each concrete semantics, which should make sure that the aforementioned properties hold for the elements of type `R`.

3.3 Module

A mathematical module is a generalization of the notion of a vector space. A module over a particular semi-ring is realised using an addition operator for two modules (similar to vector addition), and a multiplication between a semi-ring element and the module (similar to scalar-vector multiplication). For all elements a and b in a semi-ring R with the multiplicative identity 1_R, and the elements u and v in a (left-)module M, the following properties hold:

- $a \cdot (u + v) = a \cdot u + a \cdot v$
- $(a + b) \cdot u = a \cdot u + b \cdot u$
- $(a \times b) \cdot u = a \cdot (b \cdot u)$
- $1_R \cdot u = u$

Additionally, the dimension of a finite module generalises the notion of the number of basis vectors[3] representing a vector.

Figure 3 shows the tagless final interface for modules. The `Module` type class has three type parameters: (1) `V` specifies the type of the underlying vector representation; (2) `R` specifies the type of each element of the vector; and (3) `D` specifies the type of the dimension of the underlying vector. Note that all the elements of type `R` and `D` support semi-ring operations, thanks to the two type class instances `sr` and `dr`. Furthermore, the `Module` type class supports the following operations: (1) the `dim` method returns the dimension of the given module; (2) the `add` method computes the result of the addition of two module elements; and (3) the `smult` method computes the multiplication of a semi-ring element and a given module.

3.4 Linear Map

A linear map is a transformation between two modules, which preserves the addition and the scalar multiplication operations of the given module. Assume the linear map M transforming module V to module W, and both modules are over the semi-ring R. Then for all elements f in the linear map M, u and v from module V, and a from the semi-ring R, the following properties hold:

- $f(u + v) = f(u) + f(v)$
- $f(a \cdot u) = a \cdot f(u)$

Similar to functions, linear maps have two operations. First, a linear map can be applied to a module returning a transformed module, behaving similar to the function application.

[3] The basis vectors are *linearly independent* vectors (none of them can be expressed as a linear combination of the other ones) that can be used to express every vector as a *unique linear combination* of them.

```
trait Module[V, R, D] {
  implicit val sr: SemiRing[R]
  implicit val dr: SemiRing[D]
  def dim(a: V): D
  def add(a: V, b: V): V
  def smult(s: R, a: V): V
}

object Pilatus {
  // ...
  def dim[V, D](a: V)(implicit m: Module[V, _, D]): D = m.dim(a)
  // ... other boilerplate methods elided for brevity
}
```

Figure 3 The tagless final interface for modules.

```
trait LinearMap[M, V, R, D] {
  implicit val rowModule: Module[V, R, D]
  implicit val sr: SemiRing[R]
  implicit val dr: SemiRing[D]
  def apply(m: M, v: V): V
  def compose(m1: M, m2: M): M
  def add(m1: M, m2: M): M
  def dims(mat: M): (D, D)
}

object Pilatus {
  // ...
  def apply[M, V](m: M, v: V)(implicit lm: LinearMap[M, V, _, _]): V = lm.apply(m, v)
  // ... other boilerplate methods elided for brevity
}
```

Figure 4 The tagless final interface for linear maps.

Second, a linear map can be composed with another linear map resulting in another linear map, behaving similarly to function composition.

Figure 4 shows the tagless final encoding of linear maps. Here, we only consider finite linear maps transforming two finite modules, and we assume that both modules are over the same semi-ring (represented with type `R`, and the `sr` type class instance) with the same module type representation (represented with the type `V`). From a vector/matrix point of view, the `compose` and `apply` methods correspond to the matrix-matrix and matrix-vector multiplication, respectively. The `add` method corresponds to the matrix addition operator, and the `dims` construct returns the dimension of the input and output modules, which is represented as a tuple.

3.5 Pull Array and Control-Flow Constructs

Using a pull array is a well-known approach in the high-performance functional programming community for a functional encoding of arrays [64, 2, 12]. In this representation, an array is defined using two components: (1) the length of the array; and (2) a function mapping an index to the value of the corresponding element in that array.

Figure 5 demonstrates the tagless final encoding of pull arrays and looping constructs. The `build` method is responsible for constructing a pull array of size `len`, in which the i^{th}

```
trait PullArrayOps[A, E, L] {
  def build(len: L)(f: L => E): A
  def get(arr: A)(i: L): E
  def length(arr: A): L
}
trait Looping[L] {
  def forloop[S](z: S)(n: L)(f: (S, L) => S): S
}
object Pilatus {
  // ...
  def build[A, E, L](len: L)(f: L => E)(implicit p: PullArrayOps[A, E, L]): A =
    p.build(len)(f)
  // ... other boilerplate methods elided for brevity
}
```

Figure 5 The tagless final interface for pull arrays and control-flow constructs.

element is `f(i)`, indexed from `0` to `len - 1`. The `get` method returns the i^{th} element of the array `arr`, whereas the `length` method returns the size of the given array. Finally, the `forloop` method is meant for implementing recursion and iteration. More specifically, this function starts from the state `z`, iterates `n` times (from `0` to `n - 1`), and at the i^{th} step, updates the state `s` with `f(s, i)`.

4 Matrix Algebra

In this section, we build the constructs of matrix algebra based on the mathematical notions explained in the previous section. First, we show the construction of vector constructs using modules and pull arrays. Then, we demonstrate the matrix constructs by using linear maps and vectors.

4.1 Vector: Module + Pull Array

A vector (more specifically, a dense vector where most elements are non-zero) can be seen as a module the elements of which are stored as a pull array. Given that each element of a vector form a semi-ring, we can define the addition, element-wise multiplication, and dot product of two vectors.

The implementation for the tagless final encoding of a vector, as well as the mentioned methods are given in Figure 6. The `V` type parameter specifies the underlying vector type representation, the `R` type parameter specifies the type of each element of the vector, and the `D` type parameter is the type of the dimension of the underlying vector.

The `zipMap` method, receives two vectors `v1` and `v2` as input and creates a vector of the same size,[4] for which each element is constructed by applying the binary operator `op` on the corresponding elements from `v1` and `v2`. The `add` and `elemMult` are constructed by passing the addition and multiplication functions of the underlying semi-ring of elements to the `zipMap` method. The `map` method applies a given function to each element of the input vector and produces a vector of the same size with the transformed elements as output. The `smult` method is implemented using this method. Finally, the `dot` method computes the dot

[4] We assume that the input vectors have the same size for the sake of simplicity. In practice, this property can be enforced statically using Scala's powerful implicit programming capabilities, and singleton types.

```
trait Vector[V, R, D] extends Module[V, R, D] {
  implicit val pa: PullArrayOps[V, R, D]
  implicit val looping: Looping[D]
  def dim(a: V): D = pa.length(a)
  def add(v1: V, v2: V): V = zipMap(v1, v2, sr.add)
  def smult(s: R, a: V): V = map(a, e => sr.mult(s, e))
  def map(v: V, op: R => R): V = pa.build(pa.length(v))(i => op(pa.get(v)(i)))
  def zipMap(v1: V, v2: V, op: (R, R) => R): V =
    pa.build(pa.length(v1))(i => op(pa.get(v1)(i), pa.get(v2)(i)))
  def elemMult(v1: V, v2: V): V = zipMap(v1, v2, sr.mult)
  def dot(v1: V, v2: V): R =
    looping.forloop(sr.zero)(pa.length(v1))((acc, i) =>
      sr.add(acc, sr.mult(pa.get(v1)(i), pa.get(v2)(i))))
  /* sum and norm are omitted for brevity */
}
```

Figure 6 The tagless final implementation for (dense) vectors.

product of two vectors `v1` and `v2` by first computing the element-wise multiplication of these vectors, and then adding the elements of this intermediate vector.

Next, we use the mentioned vector data structure together with linear maps in order to define a matrix data-structure.

4.2 Matrix: Linear Map + Vector

Figure 7 shows the implementation of matrices (more specifically, dense matrices) using linear maps and vectors. The `M` type parameter specifies the type of the underlying matrix representation, `V` represents the type of each row-vector and column-vector of the matrix, `R` denotes the type of each element of the matrix, and `D` specifies the type of the dimension of each row and each column of the matrix.

In order to facilitate usages of the generic library, we have implemented several helper methods. The `get` method returns the corresponding element in the r^{th} row and c^{th} column of the matrix.[5] The `numRows` and `numCols` methods return the number of rows and columns of a matrix, respectively. The `getRow` method returns the vector representing the r^{th} row of the given matrix, whereas `getCol` returns a vector containing the elements in the c^{th} column of the given matrix. Finally, the `zipMap` and `map` methods have similar behaviour to the methods with the same name from the vector data type.

The `add` method returns the result of the addition of two matrices, which is implemented using the `zipMap` method. The `mult` method returns the matrix-matrix multiplication of two matrices. This method is implemented by performing a vector dot-product of each row of the first matrix with each column of the second matrix. Finally, the `transpose` method returns the transpose of the given matrix.

4.3 Putting It All Together

Before showing different evaluation semantics in the upcoming sections, we need a way to print the result values. To do so, we define the `Printable` type class which converts the value of a particular type into a string. Figure 8 shows the corresponding tagless final definition.

[5] As we will see in Section 6, `r` and `c` can have types other than `Int`.

```
trait Matrix[M, V, R, D] extends LinearMap[M, V, R, D] {
  implicit val paMat: PullArrayOps[M, V, D]
  implicit val vector: Vector[V, R, D]
  /* Other implicit values: paRow, rowModule, looping, dr, sr */
  /* apply and compose methods use the mult method, elided for brevity. */
  def get(mat: M, r: D, c: D): R =
    paRow.get(paMat.get(mat)(r))(c)
  def numRows(mat: M) = dims(mat)._1
  def numCols(mat: M) = dims(mat)._2
  def getRow(mat: M, i: D): V = paMat.get(mat)(i)
  def getCol(mat: M, j: D): V = getRow(transpose(mat), j)
  def zipMap(m1: M, m2: M, bop: (R, R) => R): M =
    paMat.build(numRows(m1))(i =>
      paRow.build(numCols(m1))(j =>
        bop(get(m1, i, j), get(m2, i, j))))
  def add(m1: M, m2: M): M = zipMap(m1, m2, sr.add)
  def mult(m1: M, m2: M): M =
    paMat.build(numRows(m1))(i =>
      paRow.build(numCols(m2))(j =>
        vector.dot(getRow(m1, i), getCol(m2, j))))
  def transpose(mat: M): M =
    paMat.build(numCols(mat))(i =>
      paRow.build(numRows(mat))(j => get(mat, j, i)))
  /* map, eye, fill, and zeros are omitted for brevity */
}
```

Figure 7 The tagless final implementation for (dense) matrices.

```
trait Printable[T] {
  def string(e: T): String
}
object Pilatus {
  // ...
  def getString[T](e: T)(implicit p: Printable[T]) = p.string(e)
}
```

Figure 8 The tagless interface for the stringification of the values of different evaluation semantics.

▶ **Example 1.** Throughout this paper, we use the following example matrix program, where we change the values for matrix `m` based on the evaluation semantic that we are interested in:

```
  def example[M, D](m: M)(implicit mev: Matrix[M,_,_,D], pev: Printable[M]): Unit = {
    import mev._
    val I = eye(numRows(m))
    val m2 = mult(m, m)
    val res = add(I, add(m, m2))
    println(getString(res))
  }
```

This program accepts the matrix `m`, the value of which differs based on the evaluation semantic that we would like to use. The result of this program is the addition of the identity matrix (represented using the `eye` method), the given input matrix, and the second power of it.

In the next sections, we give several concrete interpretations for Pilatus, and we show the output of the example program above for each of the interpretations.

```
implicit object RingInt extends Ring[Int] {
  def add(a: Int, b: Int) = a + b
  def mult(a: Int, b: Int) = a * b
  def one: Int = 1
  def zero: Int = 0
  def neg(a: Int): Int = -a
}
implicit object RingDouble extends Ring[Double] {
  /* Similar to RingInt */
}
```

Figure 9 The tagless final interpreter (a.k.a. type class instances) for a ring of integer and double values.

```
class PullArrayArrayOps[E: ClassTag] extends PullArrayOps[Array[E], E, Int] {
  def build(len: Int)(f: Int => E): Array[E] =
    Array.tabulate(len)(f)
  def get(arr: Array[E])(i: Int): E =
    arr(i)
  def length(arr: Array[E]): Int =
    arr.length
}
class LoopingInt extends Looping[Int] {
  def forloop[S](f: (S, Int) => S)(z: S)(n: Int): S =
    (0 until n).foldLeft(z)(f)
}
object Semantics {
  implicit def pullArrayArrayOps[E: ClassTag] = new PullArrayArrayOps[E]
  implicit val loopingInt = new Looping[Int]
}
```

Figure 10 The tagless final interpreter for a pull array, represented as a list of elements, and the control-flow constructs.

5 Interpreted Languages

In this section, we first show an evaluation strategy which results in a standard matrix algebra library. Then, we show how we can define an alternative interpretation which leads to treating Pilatus as a graph library. Afterwards, we show a linear algebra library for logical probabilistic programming. Finally, we demonstrate how Pilatus can behave as a library for differentiable programming.

5.1 Standard Matrix Algebra

In order to define a standard matrix algebra library for Pilatus, we start by defining a normal interpreter for rings. Figure 9 shows the interpretation for a ring of integer and double values. In both cases, the addition and multiplication operations are defined using the primitive operations provided by the Scala language.

Figure 10 shows an interpreter for pull arrays, where every constructed pull array is materialised into an array of elements. Hence, retrieving an element and returning the size of the pull array is achieved by returning the corresponding element in the materialised array and the length of the array, respectively. Finally, the implementation of `forloop` is achieved by performing a `foldLeft` on the range of elements from `0` to `n-1`, and passing the initial state and the accumulator function.

```
case class PullArrayInter[E](len: Int, f: Int => E)

class PullArrayInterOps[E] extends PullArrayOps[PullArrayInter[E], E, Int] {
  def build(len: Int)(f: Int => E): PullArrayInter[E] =
    PullArrayInter(len, f)
  def get(arr: PullArrayInter[E])(i: Int): E =
    arr.f(i)
  def length(arr: PullArrayInter[E]): Int =
    arr.len
}
object Semantics {
  // ...
  implicit def pullArrayInterOps[E] = new PullArrayInterOps[E]
}
```

Figure 11 The tagless final interpreter for a pull array, represented as a pair of length and the element constructor function.

An alternative way of interpretation for pull arrays, which avoids the materialisation of the intermediate arrays into a sequence, keeps a data structure which holds the length and the constructor function of each element. This representation is given in Figure 11.

▶ **Example 2.** The standard matrix algebra interpreter evaluates the example program as follows:

```
  import Semantics.{ pullArrayInterOps, loopingInt, ringInt }
  val adj = Array(Array(0, 0, 1, 5),
                  Array(8, 0, 3, 6),
                  Array(0, 9, 0, 0),
                  Array(0, 0, 2, 0))
  val m = build(4)(i => build(4)(j => adj(i)(j)))
  example(m)
  // output:
  [ [ 1, 9, 11, 5 ]
  , [ 8, 28, 23, 46 ]
  , [ 72, 9, 28, 54 ]
  , [ 0, 18, 2, 1 ] ]
```

Note that the `build` method is redirected to the `build` method of the `PullArrayOps` type class (cf. Figure 5).

5.2 Graph DSL for Reachability and Shortest Path

A directed graph can be represented using its adjacency matrix. More specifically, a graph with n vertices can be represented using a matrix of size $n \times n$, in which all elements are Boolean. If the element in the i^{th} row and j^{th} column is `true`, this means that there is an edge between the i^{th} and j^{th} vertices in the graph.

In order to support such adjacency matrices, we need to use the Boolean semi-ring for the matrix elements. Figure 12 shows the implementation of the Boolean semi-ring, in which addition performs disjunction, and multiplication performs conjunction.

Using the Boolean semi-ring for the elements of a matrix leads to a graph library. This instantiation of PILATUS is appropriate for expressing reachability computations among all vertices of a graph: given an adjacency matrix M, each element of $M \times M$ shows the existence of a path of length 2 between the two vertices in the corresponding graph.

```
class SemiRingBoolean extends SemiRing[Boolean] {
  def add(a: Boolean, b: Boolean) = a || b
  def mult(a: Boolean, b: Boolean) = a && b
  def one: Boolean = true
  def zero: Boolean = false
}
object Semantics {
  // ...
  implicit val semiRingBoolean = new SemiRingBoolean
}
```

Figure 12 The tagless final interpreter for a semi-ring of Boolean values, used for expressing graph reachability problems.

The graph algorithms that can be implemented on top of Pilatus are not limited to reachability ones. By adding other types of semi-rings, one can express other graph computation problems. As an example, Tropical semi-rings can express shortest-path graph problems [45, 17]. Figure 13 shows the tagless final encoding of Tropical semi-rings. The `ShortestPath` data type represents the path between two nodes of a graph, where `Unreachable` specifies a path of length $+\infty$, and `Distance(v)` specifies a path of length `v`. The Tropical semi-ring computes the minimum length of two paths as the addition operator of the semi-ring, and adds the length of two paths as the multiplication operator. We omit the definition for other semi-rings for graph and other similar problems (e.g., linear equations, data-flow analysis, petri nets, etc., which are already explored in the literature [17]).

▶ **Example 3.** When one uses the Boolean semi-ring, the example program is actually computing the existence of paths with maximum length two among all the nodes. When we provide the adjacency matrix of the graph of Figure 1a, the example program evaluates to:

```
  import Semantics.{ pullArrayInterOps, loopingInt, ringInt, semiRingBoolean }
  val adj = Array(Array(false, false, true, true),
                  Array(true, false, true, true),
                  Array(false, true, false, false),
                  Array(false, false, true, false))
  val m = build(4)(i => build(4)(j => adj(i)(j)))
  example(m)
  // output:
  [ [ T, T, T, T ]
  , [ T, T, T, T ]
  , [ T, T, T, T ]
  , [ F, T, T, T ] ]
```

5.3 Probabilistic Linear Algebra Language

Probabilistic models are used in many applications including artificial intelligence, machine learning, cryptography, and economics. Probabilistic programming languages have proven to be successful for expressing such stochastic models in a declarative style without worrying about computational aspects [11, 28, 27, 39]. As an example, an important computer vision application was recently expressed in only 50 lines of code in the Picture probabilistic programming language [40].

In this paper, our aim is not to make Pilatus a full-fledged probabilistic programming language. Instead, we show how we can encode Boolean probability distributions in Pilatus in the form of a semi-ring. This means that we support the conjunction and disjunction

```
sealed trait ShortestPath {
  def add(o: ShortestPath): ShortestPath = (this, o) match {
    case (Unreachable, x) => Unreachable
    case (x, Unreachable) => Unreachable
    case (Distance(v1), Distance(v2)) => Distance(v1 + v2)
  }
  def min(o: ShortestPath): ShortestPath = (this, o) match {
    case (Unreachable, x) => x
    case (x, Unreachable) => x
    case (Distance(v1), Distance(v2)) => Distance(math.min(v1, v2))
  }
}
case class Distance(v: Int) extends ShortestPath
case object Unreachable extends ShortestPath

class SemiRingTropical extends SemiRing[ShortestPath] {
  def add(a: ShortestPath, b: ShortestPath) = a.min(b)
  def mult(a: ShortestPath, b: ShortestPath) = a.add(b)
  def one: ShortestPath = Distance(0)
  def zero: ShortestPath = Unreachable
}

object Semantics {
  // ...
  implicit val semiRingTropical = new SemiRingTropical
}
```

Figure 13 The tagless final interpreter for a semi-ring of Boolean values, used for expressing graph shortest-path problems.

between two Boolean distributions. Also, the zero and one elements of the semi-ring correspond to the distribution with the probability of one for false and true, respectively. As a side effect of the compositional design of PILATUS, we can support vectors and matrices of such distributions as well, virtually for free. Thus, PILATUS supports probabilistic graphs and the associated path queries, similar to systems such as ProbLog [15].

Figure 14 shows the tagless final implementation for Boolean distributions. The `BoolProb` data type has a list of probabilities assigned to each Boolean value. This data type is actually a *probability monad* [26, 19]. As is customary with monad implementation in Scala, the `flatMap` method represents the bind operator of the monad, and the `apply` method of the companion object represents the unit operator. The `normalise` method makes sure that the list of probabilities associated to each Boolean value has distinct Boolean values, and that the probabilities sum up to one.

There are many alternative implementations for the probability monad such as lazy trees [39] with the possibility to support distributions for values other than Booleans. Furthermore, in this context one can use various optimisations such as variable elimination [16]. Finally, it is possible to explore other inference mechanisms [42]. All these aspects are orthogonal to the purposes of this work, and PILATUS can be extended to support all these features, which we leave as exercises to the reader.

```
case class BoolProb(l: List[(Boolean, Double)]) {
  def flatMap(f: Boolean => BoolProb): BoolProb = {
    val ll = for(x <- l; y <- f(x._1).l) yield { y._1 -> (y._2 * x._2) }
    BoolProb(ll).normalise()
  }
  def normalise(): BoolProb = {
    val sum = l.map(_._2).sum
    val nl = l.groupBy(_._1).mapValues(_.map(_._2).sum / sum)
    BoolProb(nl.toList)
  }
}
object BoolProb {
  def apply(v: Boolean): BoolProb = BoolProb(List(v -> 1.0))
}

class SemiRingBoolProb extends SemiRing[BoolProb] {
  def add(a: BoolProb, b: BoolProb) = a.flatMap(x => if(x) one else b)
  def mult(a: BoolProb, b: BoolProb) = a.flatMap(x => if(x) b else zero)
  def one: BoolProb = BoolProb(true)
  def zero: BoolProb = BoolProb(false)
}

object Semantics {
  // ...
  implicit val semiRingBoolProb = new SemiRingBoolProb
}
```

Figure 14 The tagless final implementation using the Boolean probability monad for semi-ring operations.

▶ **Example 4.** When we give the adjacency matrix of the probabilistic graph of Figure 1b as the input to the example program, the evaluation is as follows:

```
  import Semantics.{ pullArrayInterOps, loopingInt, ringInt, semiRingBoolProb }
  def flip(p: Double): BoolProb = BoolProb(List(true -> p, false -> (1 - p)))
  val adj = Array(Array(flip(0), flip(0), flip(0.1), flip(0.5)),
                  Array(flip(0.8), flip(0), flip(0.3), flip(0.6)),
                  Array(flip(0), flip(0.9), flip(0), flip(0)),
                  Array(flip(0), flip(0), flip(0.2), flip(0)))
  val m = build(4)(i => build(4)(j => adj(i)(j)))
  example(m)
  // output:
  [ [ 1, 0.1, 0.2, 0.5 ]
  , [ 0.8, 1, 0.4, 0.8 ]
  , [ 0.7, 0.9, 1, 0.5 ]
  , [ 0, 0.2, 0.2, 1 ] ]
```

As in the previous section, the example program computes the all-pairs path with maximum length of two. Hence, the result matrix is the probability of the existence of a path by traversing at most one intermediate node.

```
class DualSemiRing[R](implicit val sr: SemiRing[R])
extends SemiRing[(R, R)] {
  type Dual = (R, R)
  def add(a: Dual, b: Dual) = (sr.add(a._1, b._1), sr.add(a._2, b._2))
  def mult(a: Dual, b: Dual) =
    (sr.mult(a._1, b._1), sr.add(sr.mult(a._1, b._2), sr.mult(a._2, b._1)))
  def one: Dual = (sr.one, sr.zero)
  def zero: Dual = (sr.zero, sr.zero)
}
class DualRing[R](implicit val ring: Ring[R])
extends DualSemiRing[R] with Ring[(R, R)] {
  def neg(a: Dual) = (ring.neg(a._1), ring.neg(a._2))
}
object Semantics {
  // ...
  implicit def dualSemiRing[R: SemiRing] = new DualSemiRing[R]
  implicit def dualRing[R: Ring] = new DualRing[R]
}
```

Figure 15 The tagless final implementation using dual numbers for ring operations.

5.4 Differentiable Linear Algebra DSL

Many applications in machine learning such as training artificial neural networks require computing the derivate of an objective function. In many cases, the manual derivation of analytical derivatives is not a practical solution, as it is error prone and time consuming. Hence, several techniques were developed for automating the derivation process.

Automatic differentiation (or algorithmic differentiation) is one of the most well-known techniques to systematically compute the derivative of a program. This technique systematically applies the chain rule, and evaluates the derivatives for the primitive arithmetic operations (such as addition, multiplication, etc.) [4].

Among different implementations of automatic differentiation, here we show the forward mode technique using *dual numbers*. In this implementation, every number is augmented with an additional component, which maintains the computed derivative value. Correspondingly, all primitive operations should be augmented with the appropriate derivation computation.

Figure 15 demonstrates the generic tagless final interface for the dual number representation of a ring. This interface uses the pair representation for dual numbers, in which the first component is the normal value, whereas the second component is the derivative value. The second component in the implementation of the addition operator reflects the addition rule of derivation ($d(a + b) = da + db$), whereas the one in multiplication reflects the multiplication rule ($d(a \times b) = da \times b + a \times db$).

▶ **Example 5.** Let us consider again the example matrix given in Figure 1c. By representing this input matrix using dual numbers, our running example is evaluated as follows:

```
  import Semantics.{ pullArrayInterOps, loopingInt, ringInt, dualRing }
  val adj = Array(Array(0 -> 0, 0 -> 0, 1 -> 0, 5 -> 1),
                  Array(8 -> 0, 0 -> 0, 3 -> 2, 6 -> 0),
                  Array(0 -> 0, 9 -> 3, 0 -> 0, 0 -> 0),
                  Array(0 -> 0, 0 -> 0, 2 -> 1, 0 -> 0))
  val m = build(4)(i => build(4)(j => adj(i)(j)))
  example(m)
  // output:
```

```
[ [ 1 -> 0, 9 -> 3, 11 -> 7, 5 -> 1 ]
, [ 8 -> 0, 28 -> 27, 23 -> 8, 46 -> 8 ]
, [ 72 -> 24, 9 -> 3, 28 -> 27, 54 -> 18 ]
, [ 0 -> 0, 18 -> 15, 2 -> 1, 1 -> 0 ] ]
```

More specifically, computing the square of this matrix results in:

```
val m2 = compose(m, m)
println(getString(m2))
// output:
[ [ 0 -> 0, 9 -> 3, 10 -> 7, 0 -> 0 ]
, [ 0 -> 0, 27 -> 27, 20 -> 6, 40 -> 8 ]
, [ 72 -> 24, 0 -> 0, 27 -> 27, 54 -> 18 ]
, [ 0 -> 0, 18 -> 15, 0 -> 0, 0 -> 0 ] ]
```

This output is the same as what we have observed in Figure 1c.

6 Staging and Optimisation

In this section, we show how to use multi-stage programming (MSP, or just *staging*) to improve the performance of PILATUS programs, by removing the abstraction overhead incurred by the high-level programming features we use to make our DSL polymorphic. We use Squid [51, 52], a type-safe meta-programming framework that supports MSP. Squid is implemented in Scala as a macro library, making its usage straightforward and user-friendly.

6.1 Preliminaries on Squid and Multi-Stage Programming

Squid makes use of *quasi-quotes* to manipulate program fragments; this way, one can both compose programs together (multi-stage programming) and pattern-match on them to perform rewritings, thereby achieving *quoted staged rewriting* [50].

Quasi-quotes. Given a Scala expression `e` of type `T`, the quasi-quote expression `code"e"` has type `Code[T]` and represents a program fragment whose representation can be manipulated programmatically. Crucially, quasi-quotes may contain *holes*, delineated by the `${...}` escape syntax.[6] When constructing a program fragment, code inside of a hole is evaluated and inserted in place of the hole. For example, `val part = code"List(1,2,3)"; code"2 * $part.length"`, which has type `Code[Int]`, evaluates to `code"2 * List(1,2,3).length"`.

Runtime Compilation. After composing a program at run time using quasi-quotes, one can then either dump a stringified version of the code inside a file to be compiled and run later, or runtime-compile it on the fly, using the `.compile` method, which will produce bytecode that can then be run efficiently. After the one-off cost of runtime compilation, a definition such as `val f = code"(x: Int)\[ => x + 1".compile` will be as efficient as `val f = (x: Int)\] => x + 1`.

Multi-Stage Programming (MSP). The goal of MSP is to turn a program which contains abstractions and indirections into a *code generator*: instead of producing the program's result directly, this staged program will produce *code* that is straightforward and free of

[6] When the escaped expression is a simple identifier, one can leave out the curly braces.

abstractions, to compute the program's result more efficiently. To achieve this, one annotates the non-static parts of the program (those that should be executed later) using quasi-quotes and Code types. Thanks to runtime compilation, the staged program effectively partially evaluates the fixed parts of a program even if they depend on values obtained at runtime.

Code Pattern Matching and Rewriting. Squid extends the classical MSP ability with code pattern matching and rewriting (quasi-quotes are allowed in patterns), which lets programmers inspect already-composed code fragments in a type-safe way. As we will see in Section 6.4, in practice this saves programmers the trouble of having to define their own inspectable program representations before turning them into code.

6.2 Staging Pilatus

Thanks to the polymorphic nature of Pilatus, it is quite straightforward to turn a given semantics into a multi-stage program. All we need to do is to provide an evaluation semantics which manipulates program fragments instead of normal values, and which composes these fragments together instead of directly evaluating the results of each operation.

Figure 16 shows the staged versions of some of the Pilatus interfaces. A `RingCode[T]` is a ring implementation[7] that manipulates `Code[T]` ring elements (the `RingCode[T]` class extends `Ring[Code[T]]`). The `CodeType[T]` type class is used to automatically infer runtime type representations, which is necessary for Squid program manipulation. Notice that the implicit `ringCode` definition takes an implicit argument of type `Code[Ring[T]]`. This works out of the box, because Squid can turn an implicit `Ring[T]` into a `Code[Ring[T]]` automatically, lifting the code used for generating the implicit.

As an example, consider the following polymorphic Pilatus program:

```
def polymorphicProgram[R: Ring](a: R, b: R): R = mult(add(a, one), b)
```

And the following two usages, one with a direct `R = Int` interpretation, and one with a staged `R = Code[Int]` one:

```
import Semantics.{ ringInt }, StagedSemantics.{ ringCode }
Console.print("Enter an integer number: ")
val k = Console.readInt
val f_slow = (x: Int) => polymorphicProgram(x, k)
val f_code = (x: Code[Int]) => polymorphicProgram(x, Const(k))
val f_fast = code"(x: Int) => ${f_code}(x)".compile
```

The `Const` constructor turns a primitive value (here an `Int`) into a code value (here a `Code[Int]`). Notice that we insert `f_code` into a quasi-quote even though it is not a code value, but a function from code to code; in fact, it is implicitly lifted by Squid [51].

Assuming the user enters the number `27` on the console, the code generated at runtime for `f_fast` will be equivalent to `(x: Int)\[ => Semantics.ringInt.mult(Semantics.ringInt.add(x, 1), 27)` which, after inlining of the statically-dispatched `ringInt` methods, corresponds to `(x: Int)\] => (x + 1)* 27`. To understand why this is much more efficient than the `f_slow` version, consider that the evaluation of `f_slow` has to go through virtual dispatch of all the ring operations; moreover, it also has to use boxed representations of the manipulated

[7] Strictly speaking, this implementation does not form a ring, because for example `code"2+1"` is not the same as `code"1+2"` – though they are "morally" equivalent as they represent equivalent programs.

integer values due to the generic context in which ring operations are defined, which requires repeated allocations and unwrapping of boxed integers. As a result, in a realistic workload, even the just-in-time compiler will typically not manage to make that code as fast as the straightforward primitive operations performed by `f_fast`.[8]

This kind of overhead easily compounds as we introduce more abstractions, to the point where non-staged abstract programs end up being *orders of magnitude* slower than the staged versions [72], as we will see in Section 7.

6.3 Staged Representation Optimisations

An interesting aspect of MSP is that it lets us define data structures made of partially-staged data. For example, if we want to partially evaluate the allocation of pairs and the selection of their components, we can use representations of type `(Code[A],Code[B])` instead of `Code[(A,B)]`.

This comes in useful when representing dual numbers in our staged interpreter. We can implement an alternative to `DualRing` that is specialised for handling code values, and define its operations accordingly, for example:

```
def mult(a: (Code[R], Code[R]), b: (Code[R], Code[R])) =
  (code"$sr.mult(${a._1}, ${b._1})",
   code"$sr.add($sr.mult(${a._1}, ${b._2}), $sr.mult(${a._2}, ${b._1}))")
```

Note that in the code above, we use program fragments `a._1` and `b._1` *several times*. This is fine, because the default intermediate representation that Squid uses to encode program fragment is based on the A-normal form [20], which let-binds every subexpression to a local variable, and thus avoids code duplication [51, 50]; in other words, by inserting a given code value in several places, we only duplicate variable references.

6.4 Algebraic Optimisations

Thanks to the staged interpretations of Pilatus, which allows us to manipulate program fragments as first-class values, we can leverage the algebraic properties of ring structures to perform optimisations. To do so, we can *extend* the staged ring implementation, so that we use the normal staged method implementations by default, and *override* those methods where there is a potential for algebraic optimisations. The goal of the overridden methods is to return simplified program fragments based on the shape of their inputs.

This technique is similar to the original tagless final [10] and polymorphic embedding [30] approaches to algebraic optimisation. The main difference is that thanks to Squid's analytic capabilities, we do not need to create our own intermediate symbolic representation of programs, and instead we can pattern-match on code values directly.

An implementation of this optimised staged semantics for rings is given in Figure 17. When used in pattern position, traditional quasi-quote escapes `${...}`, which *insert* code values into bigger expressions, are written `$${...}` instead.

[8] Runtime systems like the CLR for C# avoid boxing by performing runtime specialisation of generic code, but that only achieves a small part of all the optimisation and partial evaluation we are interested in here. C++ templates can perform advanced compile-time specialisation, which could get us closer to our goal (though this means specialisation could not rely on runtime values), but they are difficult and heavyweight, yet much less flexible because they do not allow for first-class manipulation of code values.

```
import squid.IR.Predef._ // import the ‘Code’, ‘CodeType’ and ‘code’ functionalities

class SemiRingCode[T: CodeType](val sr: Code[SemiRing[T]]) extends SemiRing[Code[T]] {
  def add(a: Code[T], b: Code[T]) = code"$sr.add($a, $b)"
  def mult(a: Code[T], b: Code[T]) = code"$sr.mult($a, $b)"
  def one: Code[T] = code"$sr.one"
  def zero: Code[T] = code"$sr.zero"
}
class RingCode[T: CodeType](val ring: Code[Ring[T]])
extends SemiRingCode[T](ring) with Ring[Code[T]] {
  def neg(a: Code[T]) = code"$ring.neg($a)"
}
class PullArrayCodeOps[E: CodeType]
extends PullArrayOps[Code[PullArrayInter[E]], Code[E], Code[Int]] {
  def build(len: Code[Int])(f: Code[Int] => Code[E]): Code[PullArrayInter[E]] =
    code"PullArrayInter($len, $f)"
  def get(arr: Code[PullArrayInter[E]])(i: Code[Int]): Code[E] = code"$arr.f($i)"
  def length(arr: Code[PullArrayInter[E]]): Code[Int] = code"$arr.len"
}

object StagedSemantics {
  implicit def semiRingCode[T: CodeType]
    (implicit cde: Code[SemiRing[T]]): SemiRing[Code[T]] = new SemiRingCode(cde)
  implicit def ringCode[T: CodeType]
    (implicit cde: Code[Ring[T]]): Ring[Code[T]] = new RingCode(cde)
  // other similar definitions elided...
}
```

Figure 16 The tagless final encoding of compiled rings, pull arrays, and control-flow constructs.

Many more algebraic rewritings can be added to perform partial evaluation and normalization of program fragments. We have omitted them for the sake of brevity. Furthermore, one can encode the algebraic properties of modules (cf. Section 3.3) and linear maps (cf. Section 3.4) as rewrite rules, which we leave for the future.

6.5 Fixed-Size Matrix DSL

In some applications, such as computer vision, the matrices or vectors have a small size and sometimes their size are statically known (e.g. a vector of size 3 to show a point in the 3D space). In these cases the necessary memory for the corresponding arrays can be allocated at compile time (or even stack allocated), leading to better performance and memory consumption at run time.

Pilatus can be instantiated with an evaluator that makes sure that the length of arrays is known during the compilation time. In this case, the representation of a pull array is a sequence of the symbolic representation for each element. Furthermore, the representation for its length is an integer, instead of a symbolic representation. Interestingly, this representation is the same as the one shown in Figure 10, but with the `E` type instantiated to multi-stage code types.

```
class SemiRingOptCode[T: CodeType](sr: Code[SemiRing[T]]) extends SemiRingCode[T](sr) {
  override def add(a: Code[T], b: Code[T]) = (a, b) match {
    case (_, code"$$sr.zero") => a
    case (code"$$sr.zero", _) => b
    case _ => super.add(a, b)
  }
  override def mult(a: Code[T], b: Code[T]) = (a, b) match {
    case (_, code"$$sr.zero") => code"$sr.zero"
    case (code"$$sr.zero", _) => code"$sr.zero"
    case (_, code"$$sr.one") => a
    case (code"$$sr.one", _) => b
    case _ => super.mult(a, b)
  }
}

class RingOptCode[T: CodeType](ring: Code[Ring[T]]) extends RingCode[T](ring) {
  override def neg(a: Code[T]) = a match {
    case code"$$ring.zero" => code"$ring.zero"
    case _ => super.neg(a)
  }
}
```

Figure 17 The tagless final encoding of the compiled library of Pilatus, which applies algebraic optimisations for the elements of matrices.

6.6 Fused DSL

Deforestation [67, 25, 63, 13] is a well-known technique used in functional languages in order to remove the unnecessary intermediate data structures. This removal has a positive effect on both memory consumption and run-time performance, thanks to the removal of unnecessary memory allocations and avoidance of unnecessary computations.

One of the key advantages of using pull arrays is providing deforestation. However, to benefit from this feature, one should provide an appropriate representation for pull arrays which avoids materialisation. This can be achieved by symbolically maintaining the length and the constructor function. Whenever the array is indexed or the length of array is needed, instead of creating a symbolic representation for them, we can use the maintained length and constructor function.

Figure 18 represents the implementation of fused pull array, and a compiler allowing deforestation for Pilatus.

7 Evaluation

In this section, we show how multi-stage programming and rewriting can make Pilatus faster than the high-level implementation, while being competitive with a handwritten low-level implementation. We use several micro benchmarks consisting of a pipeline of vector operations such as addition, dot product, and norm. Each benchmark is tested with five different approaches:

- Pilatus by using a native array without optimisation
- Pilatus by using a pull array without optimisation
- Pilatus by using a pull array with staging
- Pilatus by using a pull array with staging and fusion
- A handwritten low-level optimised implementation

```
case class PullArrayCode[E](len: Code[Int], f: Code[Int] => Code[E])

class PullArrayCodeFusedOps[E]
      extends PullArrayOps[PullArrayCode[E], Code[E], Code[Int]] {
  def build(len: Code[Int])(f: Code[Int] => Code[E]): PullArrayCode[E] =
    PullArrayCode(len, f)
  def get(arr: PullArrayCode[E])(i: Code[Int]): Code[E] =
    arr.f(i)
  def length(arr: PullArrayCode[E]): Code[Int] =
    arr.len
}
```

Figure 18 The tagless final encoding of the compiled library of Pilatus, which removes all unnecessary intermediate arrays.

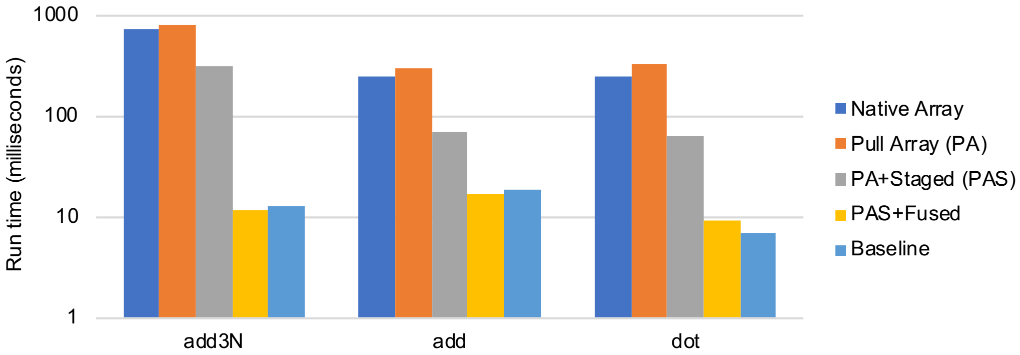


Figure 19 Performance comparison between Pilatus with different configurations and a baseline low-level implementation.

The experiments are performed on a six-core Intel Xeon E5-2620 v2 processor with 256GB of DDR3 RAM (1600Mhz), with Scala version 2.12.8 running on the OpenJDK 64-Bit Server VM (build 24.95-b01) with Java 1.7.0_101.

Figure 19 shows the performance results. For these experiments, the input vectors are all stored in a native JVM array, consisting of one million integer elements. Based on these experiments we make the following observations. First, changing the usage of native array representation to a pull array causes a minor performance overhead. This is because the JIT of JVM is unable to remove the overhead caused by the lambdas used in a pull array. Second, the overhead of lambdas as well as several other overheads are removed by using staging. This performance improvement is between 2.5x to 5x. Finally, the intermediate arrays are successfully removed by benefiting from the fusion of pull arrays (cf. Section 6.6). The improvement varies between 4x to 26x depending on the number of removed intermediate arrays. This makes the staged and fused Pilatus competitive with the baseline low-level implementation.

8 Related Work

8.1 Linear Algebra Languages and Libraries

The R programming language [56] is widely used by statisticians and data miners. It provides a standard language for statistical computing that includes arithmetic, array manipulation, object oriented programming and system calls. It is a Turing-complete language. By contrast, with our language we chose to focus solely on linear algebra operations. This minimalistic approach results in a language that is not Turing Complete, but is nevertheless polymorphic in various dimensions.

The Spiral [55] project introduces the languages SPL [71], OL [22], and more recently LL [62] which mainly captures the non-iterative matrix operations of Pilatus. Furthermore, the intermediate languages Σ-SPL [23] and Σ-LL [62] expose opportunities to perform loop fusion. One interesting direction is to use the search-based techniques to perform global optimisations offered by Spiral. Furthermore, Spiral in Scala [47] supports loop unrolling and fixed size matrices by using the staging facilities offered by LMS [57] and abstracting over data layout. Kiselyov [37] has used the tagless final approach and staging facilities of MetaOCaml in order to implement a linear algebra DSL based on rings and pull arrays (but not modules and linear maps) and has implemented many of the optimisations that we have presented in this paper. However, to the best of our knowledge, none of these projects consider graph algorithms, probabilistic programming, and automatic differentiation of linear algebra.

In the Haskell programming language, Dolan [17] implements a linear algebra library which uses different semi-ring configurations for expressing graph algorithms, as well as several other algorithms. We can easily extend Pilatus in order to support the additional semi-ring configurations used in that work. However, even though Elliot [18] implements a library for forward automatic differentiation in Haskell, Dolan [17] does not consider automatic differentiation. In addition, as both these two libraries are implemented without using any multi-stage programming facilities, none of them can support the staged libraries provided by Pilatus.

8.2 Deforestation and Array Fusion

Deforestation [67] and the corresponding short cut techniques [25, 63, 13] were introduced for functional languages for removing the unnecessary intermediate collections. Recently, these techniques have been implemented as a library using multi-stage programming [33, 38, 59].

On the other hand, in the high-performance functional array programming there are the two well-known array representations, which also achieve deforestation: pull arrays and push arrays [2, 12]. Each one of these two array representations comes with its own benefits, for which [64] combines the benefits of these two complementary representations. Pull arrays have been used for various DSLs [3, 37, 60] to produce efficient low-level code from the high-level specification of linear algebra programs. However, none of these systems consider other domains presented in this paper.

8.3 Automatic Differentiation and Differentiable Programming

Many techniques for finding optima of a given objective function (such as gradient-descent-based techniques) require the derivative of that function. Automatic differentiation (AD) [35] is one of the key techniques for automatically computing the derivative of a given program. Thus, these tools are an essential component of many machine learning frameworks. There is a large body of work on AD frameworks for imperative programming languages such as Tapenade [29] for C and Fortran, ADIFOR [7] for Fortran, and Adept [31] and ADIC [46] for C++, ADiMat [8], ADiGator [70], and Mad [21] performs AD for MATLAB programs, whereas AutoGrad [41], Theano [6], Tensorflow [1] performs AD for a subset of Python programs. There are also AD tools developed for functional languages such as DiffSharp [5] for F#, d$\tilde{\text{F}}$ [61] for a subset of F#, Stalingrad [53] for a dialect of Scheme, as well as the work by Karczmarczuk [34] and Elliott [18] for Haskell. The most similar work to ours is Lantern [69], which uses the multi-stage programming features provided by LMS [57] in order to perform AD for numerical programs written in a subset of Scala. A key feature provided by Lantern is supporting reverse-mode AD by using *delimited continuations* [14], which can also be supported by Pilatus, which we leave it for the future. All the presented frameworks are only for differentiable programming, whereas Pilatus supports graph computations and probabilistic programming, while providing algebraic-based and compiler optimisations for improving performance.

8.4 Probabilistic Programming

Probabilistic programming languages (PPL) can express stochastic models in a productive manner without worrying about the low-level details [28]. Infer.NET [44], Picture [40], and probabilistic C are examples of imperative PPLs, whereas, BUGS [24], STAN [11], and Church [27] are functional PPLs. Figaro [54] is an object-oriented probabilistic programming language which is defined as an EDSL on top of Scala. PRISM [58], BLOG [43], and ProbLog [15] are examples of Logical PPLs. Pilatus is inspired by both logical PPLs and the embedding of functional PPLs [39]. Although Pilatus has a more limited expressivity power in comparison with other logical probabilistic programs, it supports various other domains such as differentiable programming for linear algebra workloads.

9 Conclusions

In this paper, we have presented Pilatus, a polymorphic linear algebra language. This language is embedded in Scala and can have several interpretations supporting various domains such as standard matrix algebra, all-pairs reachability and shortest-path computations for graphs, logical probabilistic programming, and differentiable programming. In order to compensate the performance penalty caused by the abstraction overheads, Pilatus uses multi-stage programming. Furthermore, thanks to the mathematical nature of Pilatus, we use algebraic rewrite rules to further improve the performance.

References

1 Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. TensorFlow: A System for Large-Scale Machine Learning. In *OSDI*, volume 16, pages 265–283, 2016.

2 Johan Anker and Josef Svenningsson. An EDSL approach to high performance Haskell programming. In *ACM Haskell Symposium*, pages 1–12, 2013.

3 Emil Axelsson, Koen Claessen, Mary Sheeran, Josef Svenningsson, David Engdal, and Anders Persson. The Design and Implementation of Feldspar an Embedded Language for Digital Signal Processing. In *Proceedings of the 22Nd International Conference on Implementation and Application of Functional Languages*, IFL'10, pages 121–136, Berlin, Heidelberg, 2011. Springer-Verlag.

4 Atilim Gunes Baydin, Barak A Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. *arXiv preprint*, 2015. arXiv:1502.05767.

5 Atilim Gunes Baydin, Barak A Pearlmutter, and Jeffrey Mark Siskind. DiffSharp: Automatic Differentiation Library. *arXiv preprint*, 2015. arXiv:1511.07727.

6 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 compiler in Python. In *Proc. 9th Python in Science Conf*, pages 1–7, 2010.

7 Christian Bischof, Peyvand Khademi, Andrew Mauer, and Alan Carle. ADIFOR 2.0: Automatic differentiation of Fortran 77 programs. *IEEE Computational Science and Engineering*, 3(3):18–32, 1996.

8 Christian H Bischof, HM Bucker, Bruno Lang, Arno Rasch, and Andre Vehreschild. Combining source transformation and operator overloading techniques to compute derivatives for MATLAB programs. In *Source Code Analysis and Manipulation, 2002. Proceedings. Second IEEE International Workshop on*, pages 65–72. IEEE, 2002.

9 Jacques Carette and Oleg Kiselyov. Multi-stage Programming with Functors and Monads: Eliminating Abstraction Overhead from Generic Code. *Sci. Comput. Program.*, 76(5):349–375, May 2011.

10 Jacques Carette, Oleg Kiselyov, and Chung-Chieh Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *Journal of Functional Programming*, 19(05):509–543, 2009.

11 Bob Carpenter, Andrew Gelman, Matthew D Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. Stan: A probabilistic programming language. *Journal of statistical software*, 76(1), 2017.

12 Koen Claessen, Mary Sheeran, and Bo Joel Svensson. Expressive Array Constructs in an Embedded GPU Kernel Programming Language. In *Proceedings of the 7th Workshop on Declarative Aspects and Applications of Multicore Programming*, DAMP '12, pages 21–30, NY, USA, 2012. ACM.

13 Duncan Coutts, Roman Leshchinskiy, and Don Stewart. Stream Fusion: From Lists to Streams to Nothing at All. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, ICFP '07, pages 315–326, New York, NY, USA, 2007. ACM.

14 Olivier Danvy and Andrzej Filinski. Abstracting control. In *Proceedings of the 1990 ACM conference on LISP and functional programming*, pages 151–160. ACM, 1990.

15 Luc De Raedt, Angelika Kimmig, and Hannu Toivonen. ProbLog: A Probabilistic Prolog and Its Application in Link Discovery. In *Proceedings of the 20th International Joint Conference on Artifical Intelligence*, IJCAI'07, pages 2468–2473, San Francisco, CA, USA, 2007. Morgan Kaufmann Publishers Inc.

16 Rina Dechter. Bucket elimination: A unifying framework for probabilistic inference. In *Learning in graphical models*, pages 75–104. Springer, 1998.

17 Stephen Dolan. Fun with Semirings: A Functional Pearl on the Abuse of Linear Algebra. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 101–110, New York, NY, USA, 2013. ACM.

18 Conal M. Elliott. Beautiful Differentiation. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, ICFP '09, pages 191–202, New York, NY, USA, 2009. ACM.

19 Martin Erwig and Steve Kollmansberger. Functional pearls: Probabilistic functional programming in Haskell. *Journal of Functional Programming*, 16(1):21–34, 2006.

20 Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The Essence of Compiling with Continuations. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, PLDI '93, pages 237–247, New York, NY, USA, 1993. ACM.

21 Shaun A Forth. An efficient overloaded implementation of forward mode automatic differentiation in MATLAB. *ACM Transactions on Mathematical Software (TOMS)*, 32(2):195–222, 2006.

22 Franz Franchetti, Frédéric de Mesmay, Daniel McFarlin, and Markus Püschel. Operator language: A program generation framework for fast kernels. In *Domain-Specific Languages*, pages 385–409. Springer, 2009.

23 Franz Franchetti, Yevgen Voronenko, and Markus Püschel. Formal Loop Merging for Signal Transforms. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 315–326, 2005.

24 Wally R Gilks, Andrew Thomas, and David J Spiegelhalter. A language and program for complex Bayesian modelling. *The Statistician*, pages 169–177, 1994.

25 Andrew Gill, John Launchbury, and Simon L Peyton Jones. A short cut to deforestation. In *Proceedings of the conference on Functional programming languages and computer architecture*, FPCA, pages 223–232. ACM, 1993.

26 Michele Giry. A categorical approach to probability theory. In *Categorical aspects of topology and analysis*, pages 68–85. Springer, 1982.

27 Noah Goodman, Vikash Mansinghka, Daniel M Roy, Keith Bonawitz, and Joshua B Tenenbaum. Church: a language for generative models. *arXiv preprint*, 2012. arXiv:1206.3255.

28 Andrew D Gordon, Thomas A Henzinger, Aditya V Nori, and Sriram K Rajamani. Probabilistic programming. In *Proceedings of the on Future of Software Engineering*, pages 167–181. ACM, 2014.

29 Laurent Hascoet and Valérie Pascual. The Tapenade Automatic Differentiation Tool: Principles, Model, and Specification. *ACM Trans. Math. Softw.*, 39(3):20:1–20:43, May 2013.

30 Christian Hofer, Klaus Ostermann, Tillmann Rendel, and Adriaan Moors. Polymorphic embedding of DSLs. In *Proceedings of the 7th international conference on Generative programming and component engineering*, pages 137–148. ACM, 2008.

31 Robin J. Hogan. Fast Reverse-Mode Automatic Differentiation Using Expression Templates in C++. *ACM Trans. Math. Softw.*, 40(4):26:1–26:16, July 2014.

32 Paul Hudak. Building Domain-specific Embedded Languages. *ACM Comput. Surv.*, 28(4es), December 1996.

33 Manohar Jonnalagedda and Sandro Stucki. Fold-based Fusion As a Library: A Generative Programming Pearl. In *Proceedings of the 6th ACM SIGPLAN Symposium on Scala*, pages 41–50. ACM, 2015.

34 Jerzy Karczmarczuk. Functional differentiation of computer programs. *ACM SIGPLAN Notices*, 34(1):195–203, 1999.

35 Gershon Kedem. Automatic Differentiation of Computer Programs. *ACM Trans. Math. Softw.*, 6(2):150–165, June 1980.

36 Oleg Kiselyov. Typed tagless final interpreters. In *Generic and Indexed Programming*, pages 130–174. Springer, 2012.

37 Oleg Kiselyov. Reconciling Abstraction with High Performance: A MetaOCaml approach. *Foundations and Trends in Programming Languages*, 5(1):1–101, 2018.

38 Oleg Kiselyov, Aggelos Biboudis, Nick Palladinos, and Yannis Smaragdakis. Stream Fusion, to Completeness. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 285–299, New York, NY, USA, 2017. ACM.

39 Oleg Kiselyov and Chung-Chieh Shan. Embedded probabilistic programming. In *Domain-Specific Languages*, pages 360–384. Springer, 2009.

40 Tejas D Kulkarni, Pushmeet Kohli, Joshua B Tenenbaum, and Vikash Mansinghka. Picture: A probabilistic programming language for scene perception. In *Proceedings of the ieee conference on computer vision and pattern recognition*, pages 4390–4399, 2015.

41 Dougal Maclaurin, David Duvenaud, and Ryan P Adams. Autograd: Effortless Gradients in Numpy. In *ICML 2015 AutoML Workshop*, 2015.

42 Vikash K Mansinghka, Ulrich Schaechtle, Shivam Handa, Alexey Radul, Yutian Chen, and Martin Rinard. Probabilistic programming with programmable inference. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 603–616. ACM, 2018.

43 Brian Milch, Bhaskara Marthi, Stuart Russell, David Sontag, Daniel L Ong, and Andrey Kolobov. BLOG: Probabilistic Models with Unknown Objects. *Statistical relational learning*, page 373, 2007.

44 Tom Minka, John Winn, John Guiver, and David Knowles. Infer.NET 2.4, 2010. Microsoft Research Cambridge, 2014.

45 Mehryar Mohri. Semiring frameworks and algorithms for shortest-distance problems. *Journal of Automata, Languages and Combinatorics*, 7(3):321–350, 2002.

46 Sri Hari Krishna Narayanan, Boyana Norris, and Beata Winnicka. ADIC2: Development of a component source transformation system for differentiating C and C++. *Procedia Computer Science*, 1(1):1845–1853, 2010.

47 Georg Ofenbeck, Tiark Rompf, Alen Stojanov, Martin Odersky, and Markus Püschel. Spiral in Scala: Towards the Systematic Construction of Generators for Performance Libraries. In *Proceedings of the 12th International Conference on Generative Programming: Concepts and Experiences*, GPCE '13, pages 125–134, New York, NY, USA, 2013. ACM.

48 Bruno C.d.S Oliveira and William R Cook. Extensibility for the Masses. In *European Conference on Object-Oriented Programming*, pages 2–27. Springer, 2012.

49 Bruno C.d.S. Oliveira, Adriaan Moors, and Martin Odersky. Type Classes As Objects and Implicits. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 341–360, New York, NY, USA, 2010. ACM.

50 Lionel Parreaux, Amir Shaikhha, and Christoph E. Koch. Quoted Staged Rewriting: A Practical Approach to Library-defined Optimizations. In *Proceedings of the 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, GPCE 2017, pages 131–145, New York, NY, USA, 2017. ACM.

51 Lionel Parreaux, Amir Shaikhha, and Christoph E. Koch. Squid: Type-safe, Hygienic, and Reusable Quasiquotes. In *Proceedings of the 8th ACM SIGPLAN International Symposium on Scala*, SCALA 2017, pages 56–66, New York, NY, USA, 2017. ACM.

52 Lionel Parreaux, Antoine Voizard, Amir Shaikhha, and Christoph E. Koch. Unifying Analytic and Statically-typed Quasiquotes. *Proc. ACM Program. Lang.*, 2(POPL):13:1–13:33, December 2017.

53 Barak A Pearlmutter and Jeffrey Mark Siskind. Reverse-mode AD in a functional framework: Lambda the ultimate backpropagator. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 30(2):7, 2008.

54 Avi Pfeffer. Figaro: An object-oriented probabilistic programming language. Technical Report 137, Charles River Analytics, 2009.

55 Markus Puschel, José MF Moura, Jeremy R Johnson, David Padua, Manuela M Veloso, Bryan W Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, et al. SPIRAL: code generation for DSP transforms. *Proceedings of the IEEE*, 93(2):232–275, 2005.

56 R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2014. URL: `http://www.R-project.org/`.

57 Tiark Rompf and Martin Odersky. Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs. In *the ninth international conference on Generative programming and component engineering*, GPCE '10, pages 127–136, New York, NY, USA, 2010. ACM.

58 Taisuke Sato. A Glimpse of Symbolic-Statistical Modeling by PRISM. *Journal of Intelligent Information Systems*, 31(2):161–176, October 2008.

59 Amir Shaikhha, Mohammad Dashti, and Christoph Koch. Push versus Pull-Based Loop Fusion in Query Engines. *Journal of Functional Programming*, 28:e10, 2018.

60 Amir Shaikhha, Andrew Fitzgibbon, Simon Peyton Jones, and Dimitrios Vytiniotis. Destination-passing Style for Efficient Memory Management. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Functional High-Performance Computing*, FHPC 2017, pages 12–23, New York, NY, USA, 2017. ACM.

61 Amir Shaikhha, Andrew Fitzgibbon, Dimitrios Vytiniotis, Simon Peyton Jones, and Christoph Koch. Efficient Differentiable Programming in a Functional Array-Processing Language. *arXiv preprint*, 2018. `arXiv:1806.02136`.

62 Daniele G. Spampinato and Markus Püschel. A Basic Linear Algebra Compiler. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 23:23–23:32. ACM, 2014.

63 Josef Svenningsson. Shortcut Fusion for Accumulating Parameters & Zip-like Functions. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, ICFP '02, pages 124–132. ACM, 2002.

64 Bo Joel Svensson and Josef Svenningsson. Defunctionalizing Push Arrays. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Functional High-performance Computing*, FHPC '14, pages 43–52, NY, USA, 2014. ACM.

65 Walid Taha. A gentle introduction to multi-stage programming. In *Domain-Specific Program Generation*, pages 30–50. Springer, 2004.

66 Walid Taha and Tim Sheard. MetaML and multi-stage programming with explicit annotations. *Theor. Comput. Sci.*, 248(1-2):211–242, 2000.

67 Philip Wadler. Deforestation: Transforming programs to eliminate trees. In *ESOP'88*, pages 344–358. Springer, 1988.

68 Philip Wadler. The expression problem. *Java-genericity mailing list*, 1998.

69 Fei Wang, James Decker, Xilun Wu, Gregory Essertel, and Tiark Rompf. Backpropagation with Callbacks: Foundations for Efficient and Expressive Differentiable Programming. In *Advances in Neural Information Processing Systems*, pages 10200–10211, 2018.

70 Matthew J Weinstein and Anil V Rao. Algorithm 984: ADiGator, a toolbox for the algorithmic differentiation of mathematical functions in MATLAB using source transformation via operator overloading. *ACM Trans. Math. Softw*, 2016.

71 Jianxin Xiong, Jeremy Johnson, Robert Johnson, and David Padua. SPL: A Language and Compiler for DSP Algorithms. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, pages 298–308, New York, NY, USA, 2001. ACM.

72 Jeremy Yallop. Staged Generic Programming. *Proc. ACM Program. Lang.*, 1(ICFP):29:1–29:29, August 2017.

Towards Language-Parametric Semantic Editor Services Based on Declarative Type System Specifications

Daniel A. A. Pelsmaeker
Delft University of Technology, Delft, The Netherlands
d.a.a.pelsmaeker@tudelft.nl

Hendrik van Antwerpen
Delft University of Technology, Delft, The Netherlands
h.vanantwerpen@tudelft.nl

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

Abstract

Editor services assist programmers to more effectively write and comprehend code. Implementing editor services correctly is not trivial. This paper focuses on the specification of semantic editor services, those that use the semantic model of a program. The specification of refactorings is a common subject of study, but many other semantic editor services have received little attention. We propose a language-parametric approach to the definition of semantic editor services, using a declarative specification of the static semantics of the programming language, and constraint solving. Editor services are specified as constraint problems, and language specifications are used to ensure correctness. We describe our approach for the following semantic editor services: reference resolution, find usages, goto subclasses, code completion, and the extract definition refactoring. We do this in the context of Statix, a constraint language for the specification of type systems. We investigate the specification of editor services in terms of Statix constraints, and the requirements these impose on a suitable solver.

2012 ACM Subject Classification Software and its engineering → Semantics

Keywords and phrases semantics, constraint solving, Statix, name binding, editor services, reference resolution, code completion, refactoring

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2019.26

Category Brave New Idea Paper

Funding This research was partially funded by the NWO VICI *Language Designer's Workbench* project (639.023.206).

Acknowledgements We thank the anonymous reviewers for their feedback on previous versions of this paper. We also thank Arjen Rouvoet for his comments and his work on Ministatix, an implementation of the core Statix language we use for prototyping.

1 Introduction

Editor services, such as syntax highlighting, reference navigation, and variable renaming, are an important tool for programmers. For example, code navigation is important for effective comprehension of code [13], and refactoring approaches rely heavily on good tool support [8]. It is therefore no surprise that such services are regularly used by users of IDEs [9].

33rd European Conference on Object-Oriented Programming (ECOOP 2019).
Editor: Alastair F. Donaldson; Article No. 26; pp. 26:1–26:18
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Editor services can be classified into syntactic and semantic editor services. The former, such as syntax highlighting, rely only on the (abstract) syntax of a program. The latter depend on a semantic model of the program, and use type or name binding information. Semantic editor services can be further divided in two groups: services that *inform* about the program, and services that *transform* the program. Informing services depend on the program model that is the result of type checking. The program model contains information on the types of variables, the declarations that references refer to, etc. Transforming services are guided by the program model (e.g., to rename a declaration and all its usages), but may also rely on the typing rules to ensure the resulting, transformed program is well-formed.

Implementing semantic editor services and ensuring their correctness is not a trivial task (see, e.g., the difficulties around correctly implementing Java refactorings [14]). Language workbenches are tools to aid the development of programming languages and programming environments [5] by means of declarative formalisms and reusable tools that support correctness and reduce development effort. A good example of this is the use of a context-free grammar to specify syntax. This specification can be used to drive a parser, but also for unparsing, or to provide syntactic code completion. The language developer writes a declarative specification, which helps with the correctness of the syntax, while existing parsing, unparsing, and code completion algorithms can be reused, reducing development time.

However, even though "editor support is a central pillar of language workbenches" [3], and many language workbenches do indeed support many common editor services, there is little literature on reusable formalisms and algorithms for their definition [12]. An important exception is the extensive work on defining correct refactorings (e.g., [14, 19, 16]). However, many editor services common to modern IDEs, such as reference resolution, finding declaration usages, or semantic code completion, have received little attention.

In this paper we argue that a range of semantic editor services, beyond those that have already appeared in the literature, can be specified as constraint problems. Constraints separate the declarative specification of a problem from the operational interpretations necessary to solve it. This separates concerns, but also allows reuse of constraint-based specifications for different purposes. For example, in addition to verifying the correctness of the static semantics of a program, constraint-based typing rules have also been successfully used in the implementation of semantically correct refactorings [16].

Many editor services rely on name binding information, where complex scoping and name binding rules can be a challenge for the correct implementation of editor services (e.g., correct Java refactorings involving names [15]). Although constraint-based formulations of typing rules are pervasive, constraint-based formulations of the scoping and name binding rules are rare. Name binding introduces complexities, such as avoiding accidental name capture when refactorings introduce new names. We believe that treating name binding and name resolution as an integral part of the constraint problem increases the applicability of a constraint-based approach to editor services, and can improve existing specifications from the literature in this regard.

As the basis for our investigations we use Statix, a constraint language developed for the specification of type systems [21]. Statix is built around scope graphs, a language-independent model for name binding and name resolution [20]. We argue that Statix is a suitable basis for the definition of editor services by expressing them in terms of Statix constraints and Statix type system specifications. Although Statix constraints are suitable for a declarative specification of editor services, the current deterministic solver algorithm of Statix, suitable for type checking and code navigation, is not capable of solving the editor scenarios we discuss. We identify requirements for an alternative solver for Statix that does support the interpretation and solving algorithms required for our proposed editor service definitions.

Specifically, we have the following contributions:

- We express several common editor services in terms of Statix constraint problems.
- We identify requirements on an operational semantics of Statix that is able to solve these problems.

This paper is organized as follows. Section 2 discusses the characteristics of semantic editor services and motivates our choice of editor services. In Section 3, we introduce Statix, and Statix type specifications using an example. In Section 4 we express several informing editor services in terms of the resulting program model. In Section 5 and Section 6, we do the same for the semantic code completion and extract definition refactoring editor services, respectively. In Section 7 we discuss related work. We conclude and discuss future work necessary to fully realize our proposed approach in Section 8.

2 Characterizing Editor Services

Editor services can be characterized as syntactic, those that only need the syntactic model of the program to work, and semantic, those that require the semantic model of the program [3]. We can further distinguish the semantic editor services by whether they transform the program, or merely inform the user. The informing services include editor services such as goto declaration, finding and highlighting usages, navigating to the supertype, and listing all overriding methods. The transforming editor services include quick fixes, static semantics-preserving refactorings, and (semantic) code completion.

In this section we discuss aspects that distinguish the various semantic editor services, and motivate our choice for the editor services we discuss.

Completeness

Some editor services have to be able to work on syntactically and/or semantically incomplete programs. For example, as code completion can be invoked while the user is typing, it must be able to deal with a program that is both syntactically and semantically incomplete. Similarly, the *fix import* quick fix that adds an import statement to a program to make a reference resolve, must be able to deal with programs that have incomplete semantic information; namely the program with the reference that initially does not resolve. Other editor services could provide a better user experience if they can deal with syntactically or semantically incomplete programs, but this is not a requirement.

Preserving Static Semantics

The transforming editor services all need to preserve the existing semantics of the program up to some degree. Refactorings such as *rename refactoring* and *extract definition* tend to have very strict semantic preservation requirements, including that all existing references need to resolve to the same declarations before and after the refactoring. Quick fixes and code completion, by their nature, introduce new syntax that may change certain local semantics of the code, but should not have an impact outside the area of influence.

Concrete Name Generation

Often, transforming editor services add new declarations to the program as part of their refactoring or fixing behavior. These declarations need a concrete name, one which is syntactically valid and does not clash with existing names in the program. That is, the new name should not overlap with existing names, or cause inadvertent variable capture.

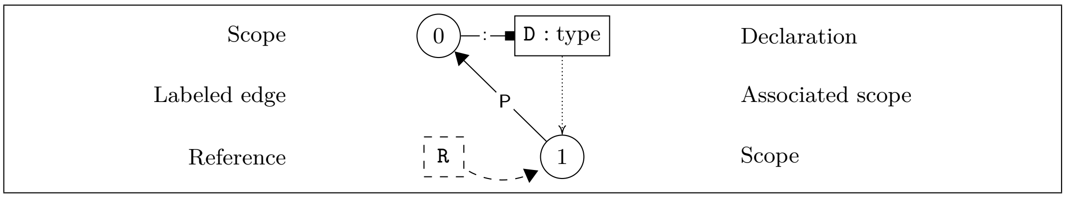


Figure 1 Overview of the notation used for scope graphs in this paper.

What We Study

Given the characterization above, we picked five editor services for which we describe the ideas of this paper. As informing editor services we choose *reference resolution*, *find usages*, and *list subclasses*, because they show how scope graphs can be used to answer these queries, where the last one requires language-specific knowledge. The last two also explore how flexible the solver must be to be able to answer such inverse queries.

We discuss two transforming editor services: *code completion*, which will have to deal with syntactically incomplete programs, and the *extract definition* refactoring, which is interesting because it introduces new syntax for which we want to use the solver to find the concrete, semantically correct, values to fill in.

We do not claim that these editor services cover all issues, or that the resulting requirements cover all editor services. However, we think that they exhibit a sufficient range of features to show the range of possibilities, and expose important requirements that need to be fulfilled to realize our approach.

3 Introduction to Statix

Statix is a recently introduced meta-language for the specification of static semantics [21], based on scope graphs and constraints [10, 20]. We chose Statix because it allows us to declare semantic editor services in terms of constraints and type system specifications.

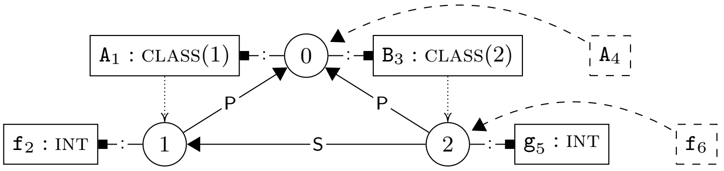
First, we explain scope graphs, a language-independent model for name binding and name resolution. Then, we introduce the rules for static semantics, and their (declarative) meaning. Finally, we explain how type checking based on these rules is implemented. We use the Java program in Figure 2 as a running example. The subscripts on program identifiers are a notational convention we use to distinguish different occurrences of the same name.

Name Binding with Scope Graphs

In Statix the name binding and resolution is part of the constraint problem, to allow complex interactions between type checking and name resolution. The name binding structure of a program is represented as a language-independent model called a *scope graph* [10, 20, 21], which is a graph of scopes and declarations in those scopes. As shown in Figure 1, the scopes are connected by labeled, directed edges. Name resolution corresponds to a query finding a path in the graph to a matching declaration.

Consider our example program and the corresponding scope graph in Figure 2. The global scope of the whole program is represented by the circled node 0. The definition of class `A` corresponds to a declaration $\mathtt{A}_1$. Declarations contain both the name and its type, and therefore use the '*is of type*'-symbol ":" to label these edges. Class types are represented by the class scope. For example, scope 1 is the scope of class `A`, and its type is CLASS(1). The class scopes are lexical sub-scopes of the global scope, which is modeled by the P-labeled

```
class A1 {
    int f2 = -1;
}
class B3 extends A4 {
    int g5 = f6;
}
```



$$\text{(J-ClassDec)}\ \frac{\begin{array}{c}\nabla s_c \quad s_c \xrightarrow{\mathsf{P}} s \quad s \xrightarrow{:} C_i : \mathrm{CLASS}(s_c) \quad s_c \vdash \overline{d}\ \mathrm{OK} \\ \mathsf{query}\ s \xmapsto{\mathsf{P}^*:} \mathrm{DECL}(D_j)\ \mathsf{as}\ D_k : \mathrm{CLASS}(s_d) \quad s_c \xrightarrow{\mathsf{S}} s_d\end{array}}{s \vdash \texttt{class}\ C_i\ \texttt{extends}\ D_j\ \{\ \overline{d}\ \}\ \mathrm{OK}}$$

$$\text{(J-FieldDec)}\ \frac{\begin{array}{cc} s \vdash [\![t]\!] \Rightarrow T & s \xrightarrow{:} f_j : T \\ s \vdash e : T' & \vdash T' <: T\end{array}}{s \vdash t\ f_j = e;\ \mathrm{OK}} \qquad \text{(J-Var)}\ \frac{\mathsf{query}\ s \xmapsto{\mathsf{P}^*\mathsf{S}^*:} \mathrm{DECL}(x_i)\ \mathsf{as}\ x_j : T}{s \vdash x_i : T}$$

$$\text{(T-Int)}\ \frac{}{s \vdash [\![\texttt{int}]\!] \Rightarrow \mathrm{INT}} \qquad \text{(T-Class)}\ \frac{\mathsf{query}\ s \xmapsto{\mathsf{P}^*:} \mathrm{DECL}(C_i)\ \mathsf{as}\ C_j : T}{s \vdash [\![C_i]\!] \Rightarrow T}$$

$$\text{(<:-Int)}\ \frac{}{\vdash \mathrm{INT} <: \mathrm{INT}} \qquad \text{(<:-Class)}\ \frac{\mathsf{query}\ s_1 \xmapsto{\mathsf{S}^*} \mathrm{SCOPE}(s_2)\ \mathsf{as}\ p}{\vdash \mathrm{CLASS}(s_1) <: \mathrm{CLASS}(s_2)}$$

Figure 2 Example Java program with two classes, its corresponding scope graph, and the relevant Statix typing rules.

(parent) edges. The fact that class `B` extends class `A` is represented by the edge labeled S (supertype). This edge makes the fields from the super class visible in the subclass, but is also used to decide subtyping between class types. The field declarations are similar to the class declarations, but in the class scopes.

Resolving a name corresponds to querying the scope graph for a matching declaration. Resolution queries are parameterized by a regular expression that determines which declaration can be reached, a predicate determining which declarations match. An additional order on labels is used to disambiguate multiple matching declarations. For example, the class reference $\mathtt{A}_4$ is resolved in the global scope 0. Class references are resolved in the lexical context, and the regular expression that encodes this is $\mathsf{P}^*:$, which matches any path to a declaration via any number of P-steps to lexical parents. The declaration itself should match the reference, which is specified with the predicate $\mathrm{DECL}(\mathtt{A}_4)$, which holds for any x_i where $x = \mathtt{A}$. In this case the reference resolves directly to declaration $\mathtt{A}_1$ in scope 0.

Resolving the variable reference $\mathtt{f}_6$ follows the same pattern. However, it should be possible to resolve not just to variables in the lexical context, but also to fields in the super class. This is achieved by using the regular expression $\mathsf{P}^*\mathsf{S}^*:$. This allows the reference to be resolved to declaration $\mathtt{f}_2$, by following the S-edge to scope 1.

Type Specifications

The rules of a Statix specification formally describe the scope graph that corresponds to a program, as well as constraints on references and types, in terms of syntax-directed rules. Figure 2 shows some of the rules that apply to our example program. For example, the rule (J-ClassDec) specifies that a class definition c is well-formed in scope s, written as $s \vdash c \text{ OK}$, if the scope graph has the correct structure, and the definitions in the class are well-formed as well ($s_c \vdash \overline{d} \text{ OK}$). The first three premises state that the scope graph contains a scope s_c that is unique to this class (∇s_c), that this scope has a P-edge to its lexical parent ($s_c \xrightarrow{\mathsf{P}} s$), and that there is a declaration C_i for the class in the lexical scope s, typed by the class scope s_c ($s \xrightarrow{:}\!\blacksquare\; C_i : \textsc{class}(s_c)$). The last two premises say that the reference to the super class resolves to a declaration D_k, which is typed by a class scope s_d (query $s \xmapsto{\mathsf{P}^*:} \textsc{decl}(D_j)$ as $D_k : \textsc{class}(s_d)$), and that an inheritance edge exists from the scope of this class to the scope of the super class ($s_c \xrightarrow{\mathsf{S}} s_d$).

The rule (J-FieldDec) specifies that a field declaration is well-formed if a declaration for the field exists in the scope graph ($s \xrightarrow{:}\!\blacksquare\; f_j : T$), if the assigned expression is well-typed for some type T' ($s \vdash e : T'$), and the expression type T' is a subtype ($\vdash T' <: T$) of the semantic type T corresponding to the type annotation ($s \vdash [\![t]\!] \Rightarrow T$). The relations for semantic typing, subtyping, and expression typing are also defined with Statix rules. The only built-in constraints are constraints to define the scope graph, constrains to query the scope graph, and term equality. All other relations are completely determined by the rules from the specification.

Type Checking

The specification is declarative, and only gives a logical description of what well-formed programs are with respect to a scope graph. We made no assumptions yet on how to operationalize it. One possible interpretation is to use the specification to type check programs. Checking that a program p is well-formed corresponds to checking if the constraint $s \vdash p \text{ OK}$ is satisfiable. Van Antwerpen et al. describe an algorithm to solve such constraints, given a specification and a program p as input [21]. The algorithm uses the rules from the specification to simplify constraints until only built-in constraints remain. These are solved using unification and scope graph resolution algorithms. This solver is deterministic: it does not use back-tracking, and only applies rules if they match the given program construct. The result of solving a constraint such as $s \vdash p \text{ OK}$ is a solution consisting of a variable assignment V and a scope graph G, or no solution if the constraint cannot be satisfied. A resulting program model would also include the types assigned to all expressions, and the resolution R of all references in the program.

4 Informing Editor Services

Many editors have editor services through which the user can navigate their program. The simplest of these involve clicking a reference and jumping to the corresponding declaration, or listing all usages of a declaration, but there are also more sophisticated editor services such as those that list the subclasses of a particular class. All these services have in common that they can be expressed as queries on the program model that resulted from type checking. Even though these queries themselves do not change the program, they may be part of the implementation of other editor services that do change the program. For example, a refactoring that renames a variable first needs to find all usages of the variable to ensure they are all renamed.

Reference resolution and finding declaration usages can easily be derived from the program model, which contains the resolution relation R, which consists of pairs of references and their declaration. Consider the example in Figure 2 again. Finding the declaration corresponding to reference $\mathtt{A}_4$, involves finding the entry for the reference in R. Conversely, finding all usages of declaration $\mathtt{f}_2$ corresponds to a reverse lookup. These queries parallel the resolution queries in the typing rules, and can directly be derived from the specification.

While a query to find all subclasses of a certain class is not directly present in the typing rules, we can phrase such a query as a constraint, which we solve with respect to the given program model. For instance, how would we specify – in constraints – the query to get all subclasses of class $\mathtt{A}_1$? We assume as input the declaration itself, and the scope 0 of the class definition, which should be part of the program model. The general idea of the query is to find the class scope, find other class scopes that are connected to it by inheritance edges, and find their corresponding declarations. This is encoded by the following constraint:

$$\textsf{query}\; 0 \xmapsto{\vdots} \textsc{decl}(\mathtt{A}_1)\;\textsf{as}\;\mathtt{A}_1 \,:\, \textsc{class}(s_c)$$

$$\textsf{query}\; s_d \xmapsto{\mathsf{s}+} \textsc{scope}(s_c)$$

$$\textsf{query}\; s' \xmapsto{\vdots} \textsc{true}\;\textsf{as}\;x_i \,:\, \textsc{class}(s_d)$$

where s_c, s_d and s' are existentially quantified, and x_i is the output. The first constraint says that the class declaration is typed by a scope s_c. The second constraint states that there is a path from some subclass scope s_d to the class scope s_c. The final constraint indicates that there is a declaration with any name x_i, which is typed by the subclass scope s_d.

None of these constraints appear as such in the typing rules, and we have to do work to find possible solutions. This may seem daunting, given the free variables for scopes and names, both of which have infinite domains. However, we are only interested in solutions that are valid in the context of an existing scope graph. This scope graph is always finite, which gives us an initial, if maybe inefficient, strategy to find possible solutions. In the case of our example, there is one solution, where $s_c = 1$, $s_d = 2$, $s' = 0$, and $x_i = \mathtt{B}_3$.

The given formulation requires an algorithm quite different from the current, deterministic solver of Statix. Instead of strictly relying on inference via forward resolution and unification, it needs to be able to guess values, try different alternatives, and back-track on failed attempts. An alternative approach could have been to change scope graph queries to allow backward edge steps. For example, if we use $\hat{l}$ for backward steps in the regular expression, our second constraint might have been:

$$\textsf{query}\; s_c \xmapsto{\hat{\mathsf{s}}+} \textsc{true}\;\textsf{as}\;s_d$$

In this case, we could do forward resolution from scope s_c again, reusing the resolution algorithm that is already there. Although this approach may work for queries designed specifically with editor services in mind, it does not work if we want to use our typing rules as-is. Therefore, we choose not to change the formalism, but require a solver that supports more flexible inference.

Summary

We showed that queries on the program model can be expressed as constraints, and that finding answers to these queries corresponds to solving these constraints in the context of a given program model. We discussed that solving these queries requires different solver strategies to be supported by the solver for Statix. However, this solver would be independent of the specific object language the query is for, and is therefore reusable between languages. Given such a solver, implementing such editor services reduces to being able to specify the query as a constraint.

5 Code Completion

Code completion is an editor service that suggests a valid code fragment to be inserted at the caret position. This assists the user while typing, attempts to minimize typing errors, and aids in discovery by showing the possible syntax and references. Syntactic code completion is the most basic kind of code completion: it suggests only syntax fragments that fit at the caret location, with no regard for whether the proposal fits semantically. Semantic code completion improves on this by suggesting only those proposals that conform to the static semantics of the language, such as only suggesting expression syntax that can produce a value of the expected type. Additionally, semantic code completion proposes inserting references to declarations, such as variables, fields, and functions, that are visible from the scope at the caret location. In this section we discuss how the type system and semantic specification of a language can be used to provide accurate semantic code completion without additional work on the part of the language designer.

In Figure 3 we show an example Java program with the caret position denoted by |, near the end of the last line of class $\mathtt{X}_{11}$. The program is incomplete: it is not syntactically valid because the user has not yet finished typing. Despite this, we would want the semantic model of the program so we can suggest relevant syntax and references.

As a first step, we propose to use the techniques described by Amorim et al. in [2] to use the syntactic specification of the language to introduce *placeholders* into the abstract syntax. A placeholder is a term in the syntax that represents a place where syntax of a certain sort, such as an expression or a declaration, could be inserted. This makes the program syntactically complete, and the placeholders provide us with syntax terms which we can constrain. Therefore, to the completion service, the incomplete line of code has the following syntax, with placeholder ***$Exp*** for a possible expression that would complete the program:

$$\mathtt{int}\ \mathtt{i}_{13}\ \mathtt{=}\ \boldsymbol{\$Exp}\ \mathtt{;}$$

At this point, we would want to invoke the solver and let it verify our program using the rules shown in Figure 3. However, no rules apply to the placeholder term ***$Exp***. Instead, we propose to replace any occurrence of a placeholder in the syntax terms with a corresponding constraint variable in the constraint terms. In this example, we use ε for ***$Exp***, which, because of the semantic rule (J-FieldDec), results in the following constraints for this line:

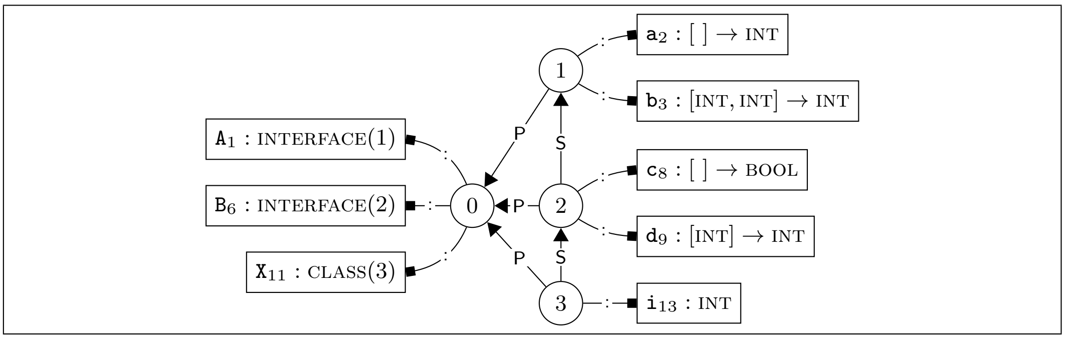
$$3 \vdash [\![\mathtt{int}]\!] \Rightarrow T \qquad 3 \overset{:}{\longrightarrow}\!\blacksquare\ \mathtt{i}_{13} : T \qquad 3 \vdash \varepsilon : T' \qquad \vdash T' <: T$$

Solving these constraints assigns $T' \mapsto$ INT and $T \mapsto$ INT. In other words, the editor service has inferred that the expected type of the expression on that line must be INT, and produced the scope graph shown in Figure 3. The solver can continue, trying to find an assignment for ε. There are two rules in Figure 3 that it could apply: (J-Plus) and (J-ThisMethodCall). In fact, we would want the solver to return both solutions for code completion. We will explore both these alternatives.

Expression Completion

From rule (J-Plus) ($s \vdash e_1 + e_2 : T$) we would get the assignment $\varepsilon = \varepsilon_1 + \varepsilon_2$, where ε_1 and ε_2 are new constraint variables introduced by the solver. We would like to stop here, and let the solver return the solution $\varepsilon = \varepsilon_1 + \varepsilon_2$. Note that this solution is incomplete: it does not

```
interface A1 {
    int a2();
    int b3(int x4, int y5);
}

interface B6 extends A7 {
    boolean c8();
    int d9(int x10);
}

class X11 implements B12 {
    int i13 = |;
}
```



$$(\text{J-InterfaceDec})\ \frac{\begin{array}{c}\nabla s_c \qquad s_c \xrightarrow{\mathsf{P}} s \qquad s \xrightarrow{:}\!\blacksquare\ C_i : \text{INTERFACE}(s_c) \\ \textsf{query}\ s \overset{:}{\mapsto} \text{DECL}(\overline{D_j})\ \textsf{as}\ \{\overline{D_k} : \text{INTERFACE}(\overline{s_d})\} \\ s_c \xrightarrow{\mathsf{S}} \overline{s_d} \qquad\qquad s_c \vdash \overline{d}\ \text{OK}\end{array}}{s \vdash \texttt{interface}\ C_i\ \texttt{extends}\ \overline{D_j}\ \{\ \overline{d}\ \}\ \text{OK}}$$

$$(\text{J-InterfaceMethodDec})\ \frac{s \vdash [\![C_i]\!] \Rightarrow T \qquad s \vdash [\![\overline{C_k}]\!] \Rightarrow \overline{T_k} \qquad s \xrightarrow{:}\!\blacksquare\ x_j : \overline{T_k} \to T}{s \vdash C_i\ x_j(\overline{C_k\ x_k});\ \text{OK}}$$

$$(\text{J-ClassDec})\ \frac{\begin{array}{c}\nabla s_c \qquad s_c \xrightarrow{\mathsf{P}} s \qquad s \xrightarrow{:}\!\blacksquare\ C_i : \text{CLASS}(s_c) \qquad s_c \vdash \overline{d}\ \text{OK} \\ \textsf{query}\ s \overset{:}{\mapsto} \text{DECL}(\overline{D_j})\ \textsf{as}\ \{\overline{D_k} : \text{INTERFACE}(\overline{s_d})\} \qquad s_c \xrightarrow{\mathsf{S}} \overline{s_d}\end{array}}{s \vdash \texttt{class}\ C_i\ \texttt{implements}\ \overline{D_j}\ \{\ \overline{d}\ \}\ \text{OK}}$$

$$(\text{J-FieldDec})\ \frac{\begin{array}{c}s \vdash [\![t]\!] \Rightarrow T \qquad s \xrightarrow{:}\!\blacksquare\ f_j : T \\ s \vdash e : T' \qquad \vdash T' <: T\end{array}}{s \vdash t\ f_j = e;\ \text{OK}}$$

$$(\text{J-ThisMethodCall})\ \frac{\begin{array}{c}s \vdash \overline{e} : \overline{V} \qquad\qquad \vdash \overline{V} <: \overline{U} \\ \textsf{query}\ s \overset{\mathsf{S}^*:}{\longmapsto} \text{DECL}(m_i)\ \textsf{as}\ \{m_j : \overline{U} \to T\}\end{array}}{s \vdash m_i(\overline{e}) : T}$$

$$(\text{J-Plus})\ \frac{s \vdash e_1 : T_1 \qquad s \vdash e_2 : T_2 \qquad T_1 = T_2 = T = \text{INT}}{s \vdash e_1 + e_2 : T}$$

Figure 3 Java program illustrating code completion, and the corresponding scope graph and relevant Statix typing rules.

describe the whole program as there are still free constraint variables in them. Therefore, the solver would need to be able to return incomplete solutions. As part of this solution, we get some constraints that not ground because they contain these free constraint variables:

$$3 \vdash \varepsilon_1 : \textsc{int} \qquad\qquad 3 \vdash \varepsilon_2 : \textsc{int}$$

Translated back to syntax terms, replacing the free constraint variables by placeholders, this would result in the following syntax on the line being completed:

```
int i₁₃ = $Exp + $Exp ;
```

Of course, we could also let the solver continue its search to find assignments for ε_1 and ε_2, but this would likely result in an ever expanding sequence of $\varepsilon_1 + \varepsilon_2 + \varepsilon_3 + \ldots$. Ultimately, there are infinitely many solutions if we were to try to make all variables ground. This shows that we need a way to instruct the solver on how deep we want a constraint variable to be solved. In this example, we want solutions for ε only one level deep.

Method Call Completion

When the solver instead applies rule (J-ThisMethodCall), we get method call completion: where code completion suggests calls to methods in scope at the caret position, and whose return a type is compatible with the expected type of the expression. From rule (J-ThisMethodCall) $(s \vdash m_i(\overline{e}) : T)$ we would get the assignment $\varepsilon = \mu(\overline{\varepsilon})$, again introducing new constraint variables μ and $\overline{\varepsilon}$ to represent the method name and arguments respectively.

Since proposing just the syntax for a method call is not very satisfactory to a user, this time we *do* want to get another level of solutions. At least, we want μ to be solved, but we do not care about $\overline{\varepsilon}$. We need a way to indicate this to the solver. Through the rule (J-ThisMethodCall) the solver would add these constraints:

$$3 \vdash \overline{\varepsilon} : \overline{V} \qquad \vdash \overline{V} <: \overline{U} \qquad \mathsf{query}\; 3 \xmapsto{\mathsf{S}^*:} \textsc{decl}(\mu)\; \mathsf{as}\; \{m_j : \overline{U} \to \textsc{int}\}$$

There are multiple possible assignments for constraint variables μ and $\overline{\varepsilon}$, and for code completion to work, the solver must find them all. The following table shows the possible assignments for μ, $\overline{\varepsilon}$, T, $\overline{U}$, and $\overline{V}$ that the solver might yield.

Solution	μ	$\overline{\varepsilon}$	T	$\overline{U}$	$\overline{V}$
Solution 1	a_2	$[\,]$	INT	$[\,]$	$[\,]$
Solution 2	b_3	$[\varepsilon_1, \varepsilon_2]$	INT	$[\textsc{int}, \textsc{int}]$	$[\tau_1, \tau_2]$
~~Solution 3~~	c_8	$[\,]$	BOOL	$[\,]$	$[\,]$
Solution 4	d_9	$[\varepsilon_1]$	INT	$[\textsc{int}]$	$[\tau_1]$

Note that solution 3 is not valid, as it tries to assign $T \mapsto \textsc{bool}$ whereas T had previously already been assigned INT. Also note how the solver could infer lists of constraint variables for $\overline{U}$ and $\overline{V}$. But, as before, we would not want the solver to keep expanding on the constraint variables it has introduced. If we had not relaxed these variables such that they may remain free, the solver would have to find some assignment for the variables that satisfies them. In this example the solver might have added a method call to an arbitrary method with a compatible return type, such as a_2. In other scenarios the solver may not be able to find such a solution, or find infinitely many.

The solutions returned by the solver can be turned into syntax fragments and presented to the user as code completion proposals, where we replace the free constraint variables by syntax placeholders. The order of the proposals is not determined by the solver, as we consider this to be a separate concern. For example, we may want to order the proposals by their frequency of use, or use the semantic model to order the proposals by closeness (e.g., local variables before global variables). In this example, code completion would propose the following method calls:

```
a_2()
b_3($Exp, $Exp)
d_9($Exp)
```

Summary

To use the semantic of the programming language for code completion, we first need a semantic specification that includes a model for name binding. This is already provided by the scope graphs used by the Statix constraint solver. However, the solver also needs to support returning incomplete solutions. The solver needs to be able to distinguish between constraint variables that we want to have solved and those that may remain free, and we need to be able to indicate how deep we want a given constraint variable to be solved. By using the semantic rules, a solution can include syntactic assignments to variables. Finally, the solver must be able to return more than one solution, so we can display them all to the user as part of code completion.

6 Extract Definition

A common refactoring is the *extract definition* refactoring, where the user selects a subexpression and the refactoring replaces any occurrences of that expression by a reference to a variable definition initialized by the subexpression. In the example in Figure 4, we want to extract the `x - 3` subexpression into a separate definition. We assume the program is syntactically complete and semantically correct.

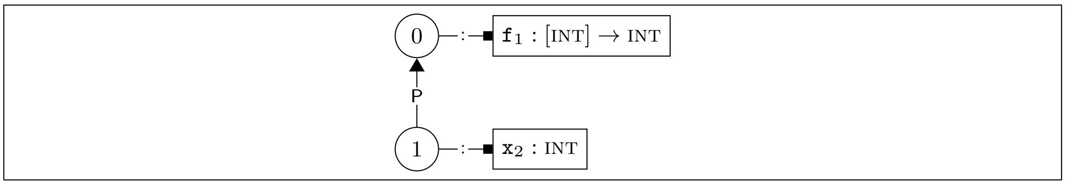
The first step in this refactoring is to determine the new syntax that we expect as a result of the refactoring. This is language-specific syntax, selected by the user and specified in advance by the language developer. The syntax fragment uses placeholders, as shown below, where *$Type* is a placeholder for the type of the newly created variable and *$ID* is a placeholder for a variable name. In this case we want all three occurrences *$ID* to refer to the same variable.

```
int f_1(int x_2) {
    $Type $ID = x_3 - 3;
    return $ID + $ID * x_5;
}
```

We create a copy of the previous, valid, solution returned by the solver, and adapt it to this refactoring. This is a two-part process: relaxing the solution, and adding new constraints to the problem. Relaxing the solution removes any variables, resolutions, constraints, and scope graph nodes that are no longer valid or that impact the aspects we want to refactor. For extracting a definition, relaxation only involves removing the reference relation $x_4 \mapsto x_2$, since the reference x_4 has been removed. However, we still want the variable references x_3 and x_5 to resolve to the same definition x_2.

Now we can add new constraints to the problem, but the refactoring should add only those constraints that result from the changed syntax. The constraints may contain syntax terms, but we replace any occurrences of the placeholders by constraint variables. For the

```
int f₁(int x₂) {
    return (x₃ - 3) + (x₄ - 3) * x₅;
}
```



$$\text{(J-MethodDec)}\ \frac{\begin{array}{c} s \vdash [\![C_i]\!] \Rightarrow T \quad \nabla s_m \quad s_m \xrightarrow{\mathsf{P}} s \\ s \vdash [\![\overline{C_k}]\!] \Rightarrow \overline{T_k} \quad s_m \xrightarrow{:}\!\blacksquare\ \overline{p_k} : \overline{T_k} \\ s_m \vdash \overline{b}\ \text{OK} \quad s \xrightarrow{:}\!\blacksquare\ m_j : \overline{T_k} \to T \\ s_m \vdash e : U \quad \vdash U <: T \end{array}}{s \vdash C_i\ m_j(\overline{C_k}\ \overline{p_k})\ \{\ \overline{b}\ \texttt{return}\, e;\}\ \text{OK}}$$

$$\text{(J-VarDec)}\ \frac{s \vdash [\![C_x]\!] \Rightarrow T \quad s \xrightarrow{:}\!\blacksquare\ v_y : T \quad s \vdash e : T}{s \vdash C_x\ v_y = e;\ \text{OK}}$$

$$\text{(J-Var)}\ \frac{\mathsf{query}\ s \xmapsto{\mathsf{P^*S^*:}} \text{DECL}(x_i)\ \mathsf{as}\, x_j : T}{s \vdash x_i : T}$$

$$\text{(J-IntBinOp)}\ \frac{s \vdash e_1 : T_1 \quad s \vdash e_2 : T_2 \quad T_1 = T_2 = T = \text{INT}}{s \vdash e_1 \oplus e_2 : T}$$

Figure 4 Java program before applying the *extract definition* refactoring, and the corresponding scope graph and relevant Statix typing rules.

type placeholder ***$Type***, we will use the constraint variable τ. Since we want all occurrences of the ***$ID*** placeholder to refer to the same variable, we should replace all occurrences with the same constraint variable.

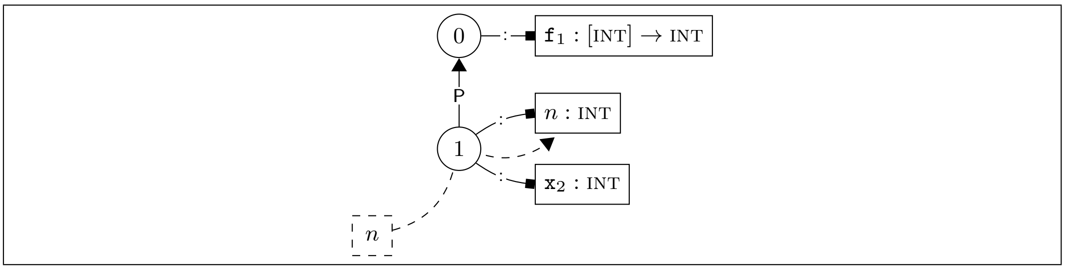
Where other approaches use the solver to find a concrete name for the variable ([16]), we argue that this is not necessary for the solver to give a correct result. We merely want to indicate to the solver that the new name is different from all other names in the program. This gives a separation of concerns: the solver can verify that the program satisfies the constraints without needing to produce any concrete names, and generating the concrete names can be done externally after the solver has verified the program. For example, some IDEs provide a list of name suggestions that they generate from the context, such as the type of the expression.

To distinguish the abstract name from any other name in the program, we can use a rigid variable: one that is distinct from any other variable or name. Similar to how rigid variables are used to create new distinct scopes in the scope graph (through ∇s), we create a new rigid variable to represent the name of the newly introduced variable: ∇n. Due to rules (J-VarDec) and (J-Var), this results in the following constraints:

$$\begin{array}{ll} \nabla n & \\ 1 \vdash [\![\tau]\!] \Rightarrow T & 1 \xrightarrow{:}\!\blacksquare\ n : T \\ 1 \vdash (\mathtt{x}_3 - 3) : T & \mathsf{query}\ 1 \xmapsto{\mathsf{P^*S^*:}} \text{DECL}(n)\ \mathsf{as}\, x_j : T \end{array}$$

Solving these constraints results in the variable assignment and scope graph shown in Figure 5.

```
int f(int x) {
    int n = x - 3;
    return n + n * x;
}
```



	n	τ	T	v_j
Solution	n	`int`	INT	n

Figure 5 Java program after applying the *extract definition* refactoring, and the corresponding scope graph and variable assignments. Note that n in the program is a rigid variable, which has yet to be assigned a concrete name.

From this we can conclude that the refactoring is valid, does not semantically change the program (since the existing constraints and reference resolutions are preserved), and that the type of the newly introduced variable is `int`. However, to finish the refactoring we have to decide on a concrete name for rigid variable n. A concrete name can be provided by the refactoring tool or by the user. In any case, we can test whether the suggested name is allowed by reinvoking the solver with the new solution and one additional constraint: to constrain n to the chosen name, say `i`.

$$n = \texttt{i}$$

The new constraint may result in an invalid solution, for example when the chosen name overlaps with another, or causes inadvertent name capture somewhere in the program. In this example, `x` is not allowed as a concrete name for n. However, if this results in a valid solution, the concrete name is acceptable and the refactoring can finish. In this example it would produce the following code:

```
int f(int x) {
    int i = x - 3;
    return i + i * x;
}
```

Summary

As part of the refactoring we generate new syntax, where we use placeholders to indicate where we need more information. In the newly generated constraints we have a constraint variable taking the place of every placeholder, which allows us to use the solver to find a solution to the problem. By using a rigid variable in place of a concrete name, we can indicate that the name is different from all other names in the program without having to specify such a name concretely, giving a separation of concerns between finding whether the program is valid and what concrete name to choose.

7 Related Work

Erdweg et al. [3] identify editor services as an important aspect of language workbenches, and give an overview of commonly supported editor services. However, Omar et al. have pointed out that the study of the semantic foundations of editors and editor interactions has received little attention so far [12]. We discuss work related to code completion, and refactoring, as those are most relevant to the editor services we covered in this paper.

Reference Resolution

Language workbenches such as Xtext [4] and Spoofax [6, 22] provide support for language-parametric reference resolution based on declarative name binding specifications. Xtext supports specification of references in the language grammar as crosslinks, which specify the sort that an identifier can refer to. Xtext will check the validity of the references and add them to the model.

The first approach to declarative name binding specification in Spoofax was the NaBL name binding language [7]. The name binding rules defined the definition sites and their scopes based on the abstract syntax of the program. The built-in reference resolution algorithm could only create an index in which references can be looked up, which limits its flexibility to be used in other editor services.

Instead, the approach we use in this paper uses the expressiveness of the constraints and the flexibility of the Statix constraint solver to enable reference resolution to be used in various editor services.

Code Completion

The Xtext and Spoofax language workbenches also provide support for language-parametric syntactic completion, based on a syntax definition. In the case of Xtext, it suggests possible keywords. Spoofax suggests complete syntactic constructs, and represents incomplete syntax trees using placeholders that act like holes in the program text [2]. This representation is instrumental for translating an incomplete program to an abstract syntax tree with variables, which allows us to use it in a constraint context. A program with placeholders is similar to the representation of an AST with holes that is common in structure editors. Although structure editors are primarily concerned with guaranteeing that the program is well-typed with respect to the abstract syntax signature, recent work investigates editors that also maintain other well-formedness properties, such as well-typedness.

The Hazelnut editor provides a language-parametric structure editor that guarantees well-typed ASTs for languages whose type system is defined in a bidirectional style [11]. JastAdd extends their reference attribute grammars to provide a context-sensitive completion service that suggest the names of variables and functions, but still requires some language-specific effort to derive these suggestions [18]. Steimann et al. use constraint-based language specifications to ensure edits preserve well-formedness [17]. They focus on an architecture that allows interaction between the solver and the user during the editing process, to resolve conflicts that may have been introduced. Our aim is to generate semantic completion proposals by combining the mechanisms for syntactic completion, with checking and inference based on the language specification. Another important difference is that issues around name resolution are largely ignored in their work, because references are actual references in the underlying model, whereas in our text-based setting we need to consider naming issues.

Semantic code completion also has similarities to interactive proof search, such as offered by proof assistants. For example, the editor of Agda [1] features holes that are similar to placeholders. An automatic procedure tries to find proof terms (expressions) that fit the goal (type). There are some important differences with our approach to code completion. The procedure to find these terms is not language-parametric, but specific to Agda. The search procedure does not exploit the typing rules, but duplicates knowledge from the type checker. Type correctness is guaranteed by type checking the fragment after it is generated.

Refactoring

There is a long line of research on the specification and implementation of refactorings. Tip et al. [19] study type related refactorings, such as adding type parameters, extracting interfaces, and pulling up methods. They use type constraints to specify the invariants that ensure correct behavior. Steimann and others [16] extend this work to include constraints for other aspects such as access modifiers and names. By representing the program itself using constraint variables, both the invariants and the refactoring intent can be represented as constraints. Finding the refactored program, within the limits of the given constraints, is delegated to the constraint solver. This approach is in many aspects similar to ours, and hopefully techniques they developed for performance carry over to our approach. An important difference is that by using Statix, the constraint solver is aware of the complete binding model. In their approach preventing capture requires the introduction of inequality constraints between names. These constraints do not follow from regular constraint-based typing rules. In our approach, the resolution constraints that are part of the typing rules can also be used to ensure the invariance of name resolution during refactoring.

8 Conclusion

In this paper we have discussed various semantic editor services, and shown how they can be expressed in terms of the semantic rules, constraints, and scope graph. We show that Statix constraints are expressive enough to formulate interesting editor services. We have pointed out that the Statix solver used for type checking is not suitable for the scenarios that arise in editor services, and we have identified several requirements that such alternative solver strategies should have. The main requirements we identified are:

- The solver must be able to try different alternatives, guess values, back-track on failed attempts, and able to return multiple solutions. As we have shown, this is a requirement to implementing code completion, but also for other editor services such as find usages and find all subclasses.
- Instead of always searching for complete solutions (i.e., assignments to all variables), the search should be controlled by user-defined criteria, including whether the solver should only consider deterministic inference, or whether it tries to find solutions non-deterministically. These criteria should be able to depend on variables appearing in the constraints. Specifying which constraint variables may remain unconstrained, and how deep the solver should search for an assignment, allows us to direct the search to finding solutions to only those constraint variables we are interested in, and prevent the solver from getting stuck.

Finally, we propose to extend the mechanisms of creating scopes in Statix to a general mechanism of rigid variables. These rigid variables can be used to solve problems around inventing new concrete names in the constraint solver, and should make it easier to implement various refactorings without having to deal with concrete names, accidental variable capture, and ambiguous names. Factoring out the choice of finding concrete names separates concerns, and also allows for language-specific strategies (e.g., suggesting variables names based on types).

Future Work

This paper presented ideas on what would be needed for language-parametric semantic editor services. To verify our approach, we need to implement the proposed extensions to the Statix solver. This will allows us to evaluate their feasibility in practice, as it is not clear whether implementing some of these techniques, such as having the solver back-track and trying to find multiple possible solutions, would cause performance issues or introduce non-termination. And if so, how we could avoid that without impacting the expressiveness of the semantic rules too much.

There are editor services, other than those we discussed, to which we could apply our approach, such as *fix import*, *search for symbol*, and in particular *rename refactoring*. Rename refactoring is interesting because it not only needs to rename the references to the renamed declaration, but possibly other references and declarations as well. For example, when a method is renamed, all overriding methods need to be renamed too, and this relation is not visible in the program model, but only encoded in the semantic rules.

While our current approach is focussed on preserving the static semantics of the program, for certain refactorings it may be required to extend the approach to also preserve certain dynamic aspects of the semantics. Additionally, a combination of our approaches might be used to implement program generation that is guaranteed to produce programs that are semantically correct.

References

1 Catarina Coquand, Makoto Takeyama, and Dan Synek. An Emacs-Interface for Type-Directed Supportfor Constructing Proofs and Programs. In *European Joint Conferences on Theory and Practice of Software, ENTCS*, volume 2, 2006.

2 Luis Eduardo de Souza Amorim, Sebastian Erdweg, Guido Wachsmuth, and Eelco Visser. Principled syntactic code completion using placeholders. In Tijs van der Storm, Emilie Balland, and Dániel Varró, editors, *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering, Amsterdam, The Netherlands, October 31 - November 1, 2016*, pages 163–175. ACM, 2016.

3 Sebastian Erdweg, Tijs van der Storm, Markus Völter, Meinte Boersma, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad A. Vergu, Eelco Visser, Kevin van der Vlist, Guido Wachsmuth, and Jimi van der Woning. The State of the Art in Language Workbenches - Conclusions from the Language Workbench Challenge. In Martin Erwig, Richard F. Paige, and Eric Van Wyk, editors, *Software Language Engineering - 6th International Conference, SLE 2013, Indianapolis, IN, USA, October 26-28, 2013. Proceedings*, volume 8225 of *Lecture Notes in Computer Science*, pages 197–217. Springer, 2013. doi:10.1007/978-3-319-02654-1_11.

4 M. Eysholdt and H. Behrens. Xtext: implement your language faster than the quick and dirty way. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pages 307–309. ACM, 2010.

5 Martin Fowler. Language Workbenches: The Killer-App for Domain Specific Languages?, 2005.

6 Lennart C. L. Kats and Eelco Visser. The Spoofax language workbench: rules for declarative specification of languages and IDEs. In William R. Cook, Siobhán Clarke, and Martin C. Rinard, editors, *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010*, pages 444–463, Reno/Tahoe, Nevada, 2010. ACM. doi:10.1145/1869459.1869497.

7 Gabriël Konat, Lennart C. L. Kats, Guido Wachsmuth, and Eelco Visser. Declarative Name Binding and Scope Rules. In Krzysztof Czarnecki and Görel Hedin, editors, *Software Language Engineering, 5th International Conference, SLE 2012, Dresden, Germany, September 26-28, 2012, Revised Selected Papers*, volume 7745 of *Lecture Notes in Computer Science*, pages 311–331. Springer, 2012. doi:10.1007/978-3-642-36089-3_18.

8 Tom Mens, Serge Demeyer, Bart Du Bois, Hans Stenten, and Pieter Van Gorp. Refactoring: Current Research and Future Trends. *Electronic Notes in Theoretical Computer Science*, 82(3):483–499, 2003.

9 Gail C. Murphy, Mik Kersten, and Leah Findlater. How Are Java Software Developers Using the Eclipse IDE? *IEEE Software*, 23(4):76–83, 2006. doi:10.1109/MS.2006.105.

10 Pierre Néron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. A Theory of Name Resolution. In Jan Vitek, editor, *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9032 of *Lecture Notes in Computer Science*, pages 205–231. Springer, 2015. doi:10.1007/978-3-662-46669-8_9.

11 Cyrus Omar, Ian Voysey, Michael Hilton, Jonathan Aldrich, and Matthew A. Hammer. Hazelnut: a bidirectionally typed structure editor calculus. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 86–99. ACM, 2017.

12 Cyrus Omar, Ian Voysey, Michael Hilton, Joshua Sunshine, Claire Le Goues, Jonathan Aldrich, and Matthew A. Hammer. Toward Semantic Foundations for Program Editors. In Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi, editors, *2nd Summit on Advances in Programming Languages, SNAPL 2017, May 7-10, 2017, Asilomar, CA, USA*, volume 71 of *LIPIcs*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017. doi:10.4230/LIPIcs.SNAPL.2017.11.

13 Martin P. Robillard, Wesley Coelho, and Gail C. Murphy. How Effective Developers Investigate Source Code: An Exploratory Study. *IEEE Trans. Software Eng.*, 30(12):889–903, 2004. doi:10.1109/TSE.2004.101.

14 Max Schäfer and Oege de Moor. Specifying and implementing refactorings. In William R. Cook, Siobhán Clarke, and Martin C. Rinard, editors, *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010*, pages 286–301, Reno/Tahoe, Nevada, 2010. ACM. doi:10.1145/1869459.1869485.

15 Max Schäfer, Andreas Thies, Friedrich Steimann, and Frank Tip. A Comprehensive Approach to Naming and Accessibility in Refactoring Java Programs. *IEEE Trans. Software Eng.*, 38(6):1233–1257, 2012. doi:10.1109/TSE.2012.13.

16 Friedrich Steimann. Constraint-Based Refactoring. *ACM Transactions on Programming Languages and Systems*, 40(1), 2018. doi:10.1145/3156016.

17 Friedrich Steimann, Marcus Frenkel, and Markus Voelter. Robust projectional editing. In Benoît Combemale, Marjan Mernik, and Bernhard Rumpe, editors, *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2017, Vancouver, BC, Canada, October 23-24, 2017*, pages 79–90. ACM, 2017. doi:10.1145/3136014.3136034.

18 Emma Söderberg and Görel Hedin. Building semantic editors using JastAdd: tool demonstration. In Claus Brabrand and Eric Van Wyk, editors, *Language Descriptions, Tools and Applications, LDTA 2011, Saarbrücken, Germany, March 26-27, 2011. Proceeding*, page 11. ACM, 2011. doi:10.1145/1988783.1988794.

19 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):9, 2011. doi:10.1145/1961204.1961205.

20 Hendrik van Antwerpen, Pierre Néron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. A constraint language for static semantic analysis based on scope graphs. In Martin Erwig and Tiark Rompf, editors, *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 49–60. ACM, 2016. doi:10.1145/2847538.2847543.

21 Hendrik van Antwerpen, Casper Bach Poulsen, Arjen Rouvoet, and Eelco Visser. Scopes as types. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA), 2018. doi:10.1145/3276484.

22 Eelco Visser, Guido Wachsmuth, Andrew P. Tolmach, Pierre Néron, Vlad A. Vergu, Augusto Passalaqua, and Gabriël Konat. A Language Designer's Workbench: A One-Stop-Shop for Implementation and Verification of Language Designs. In Andrew P. Black, Shriram Krishnamurthi, Bernd Bruegge, and Joseph N. Ruskiewicz, editors, *Onward! 2014, Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software, part of SPLASH '14, Portland, OR, USA, October 20-24, 2014*, pages 95–111. ACM, 2014. doi:10.1145/2661136.2661149.

Multiverse Debugging: Non-Deterministic Debugging for Non-Deterministic Programs

Carmen Torres Lopez
Vrije Universiteit Brussel, Belgium
ctorresl@vub.be

Robbert Gurdeep Singh
Universiteit Gent, Belgium
Robbert.GurdeepSingh@ugent.be

Stefan Marr
School of Computing, University of Kent, United Kingdom
s.marr@kent.ac.uk

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

Christophe Scholliers
Universiteit Gent, Belgium
Christophe.Scholliers@ugent.be

Abstract

Many of today's software systems are parallel or concurrent. With the rise of Node.js and more generally event-loop architectures, many systems need to handle concurrency. However, its non-deterministic behavior makes it hard to reproduce bugs. Today's interactive debuggers unfortunately do not support developers in debugging non-deterministic issues. They only allow us to explore a single execution path. Therefore, some bugs may never be reproduced in the debugging session, because the right conditions are not triggered.

As a solution, we propose multiverse debugging, a new approach for debugging non-deterministic programs that allows developers to observe all possible execution paths of a parallel program and debug it interactively. We introduce the concepts of multiverse breakpoints and stepping, which can halt a program in different execution paths, i.e. universes. We apply multiverse debugging to AmbientTalk, an actor-based language, resulting in *Voyager*, a multiverse debugger implemented on top of the AmbientTalk operational semantics. We provide a proof of non-interference, i.e., we prove that observing the behavior of a program by the debugger does not affect the behavior of that program and vice versa. Multiverse debugging establishes the foundation for debugging non-deterministic programs interactively, which we believe can aid the development of parallel and concurrent systems.

2012 ACM Subject Classification Software and its engineering → Concurrent programming languages; Software and its engineering → Software testing and debugging

Keywords and phrases Debugging, Parallelism, Concurrency, Actors, Formal Semantics

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2019.27

Category Brave New Idea Paper

Supplement Material ECOOP 2019 Artifact Evaluation approved artifact available at `https://dx.doi.org/10.4230/DARTS.5.2.4`

Funding *Carmen Torres Lopez*: Funded by FWO Research Foundation Flanders (FWO), project number G004816N.
Robbert Gurdeep Singh: Doctoral fellowship from the Special Research Fund (BOF) of Ghent University (reference number: BOF18/DOC/327).

33rd European Conference on Object-Oriented Programming (ECOOP 2019).
Editor: Alastair F. Donaldson; Article No. 27; pp. 27:1–27:30
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Acknowledgements We would like to thank Thomas Dupriez (ENS Paris-Saclay - RMoD, Inria, Lille-Nord Europe) for an initial implementation of the underlying visualization and reduction code.

1 Introduction

Parallelism has become an integral part of modern software ranging from large-scale server code to responsive web applications or networked embedded systems. While a wide range of high-level concurrency abstractions are available for developers, understanding and debugging parallel programs remains challenging. The main reason why parallel programs are so difficult to debug is due to their non-determinism. Since the state of a parallel program at any given moment in time can alter to one of *many* possible successor states, it is very difficult to reason about their behavior and to reproduce bugs as they may only manifest in rare execution traces.

Debugging tools for parallel and concurrent programs have been studied in the past and can be categorized in two main families [48]: event-based debuggers (also known as log-based debuggers) and breakpoint-based debuggers (also known as online or interactive debuggers). While event-based approaches generate a program trace for offline browsing or deterministic replay, breakpoint-based debuggers control the program execution allowing developers to pause/resume program execution at well-defined points (e.g. on a breakpoint), inspect program state, and perform step-by-step execution.

Despite the presence of online debuggers in modern IDEs, a recent study showed that debugging parallel applications remains very problematic [53], because debuggers do not account for the non-determinism of concurrent applications. Most of the existing tools only provide support for deterministic debugging, i.e., they support the debugging of *only* one parallel entity at a time rather than the program as a whole. This means that one run of the debugger is very likely to miss the erroneous state in which the bug manifests itself, requiring many debugging cycles before being able to reproduce the bug. Even worse, the mere presence of a debugger may affect the order in which parallel entities are executed, making the reproduction of a bug even rarer. This condition akin to the Heisenberg uncertainty principle, is known as the *probe effect* [25].

In addition to debugging techniques, static analyses have been studied for parallel and concurrent programs to detect certain types of program errors without executing the program, e.g. static analysis to verify the boundedness of actor mailboxes [24], model checkers for concurrent programs written in Erlang [15] or Scala [38], type systems to ensure type safety on reciprocal communication channels [56]. These techniques detect synchronization errors such as deadlocks [10, 23], incorrect ordering of locks [4], and incorrect interleaving of messages in actor systems [15]. However, they often put severe restrictions on the way programs are organized (e.g. on how futures are used in actor-based programs [26]). More importantly, they expect developers to have a good understanding of what caused the bug as they verify a well-defined property over a program, but they currently cannot be used interactively to explore and search for a bug with an unknown cause. Finally, static analysis techniques are almost always about approximations, when the analysis detects a bug it might be impossible to find a concrete execution path which triggered the bug.

What is needed to *debug* non-deterministic programs is a technique which: 1) allows programmers to observe *all* the possible states a parallel program can exhibit at run time and 2) is probe-effect free. In this paper, we propose *multiverse debugging*, a novel debugging technique for parallel programs which combines breakpoint-based debugging with state exploration from static techniques. The key idea of multiverse debugging is that *non-*

deterministic programs require non-deterministic debugging. Contrary to current state-of-the-art debuggers, which only execute the program in one execution path (i.e. one *universe*), a multiverse debugger can observe all possible universes. A multiverse debugger is itself a non-deterministic program which is able to explore all possible states of a parallel program while leveraging breakpoints and stepping commands of online debuggers to interactively search for the root cause of a bug. This means that regular breakpoints become *multiverse breakpoints* which are potentially triggered multiple times in different universes. As such, a multiverse debugger ensures that if a bug is in the program, it will be observed during the debugging session.

In this paper we give an overview of how to design a multiverse debugger starting from the operational semantics of a non-deterministic language. We evaluate multiverse debugging by applying it to actor-based programs written in the AmbientTalk language. On top of the existing AmbientTalk operational semantics (known as Featherweight AmbientTalk) [62], we formalized the debugging semantics and developed *Voyager*, a tool that takes as input Featherweight AmbientTalk programs written in PLT-Redex, and allows programmers to interactively browse all possible execution states by means of multiverse breakpoints and stepping commands. We also provide a proof of *non-interference*, i.e. the base language and the debugger are observational equivalent.

The key contributions of this paper are:

- Definition of multiverse debugging and multiverse stepping, i.e. novel stepping semantics which allow developers to step to all possible states in the execution of a parallel program.
- A semantics for a non-deterministic debugger & proof of non-interference.
- An implementation of applying multiverse debugging to an actor-based language including a tool to interact with the debugged program written in PLT-Redex.

2 Brave New Idea: Multiverse Debugging

The vision of multiverse debugging is to allow programmers to debug concurrent non-deterministic programs with a debugging technique that allows them to observe *all* possible states the program can exhibit at run time and to interactively explore these states for bugs in a fashion similar to breakpointed-based debuggers while being probe-effect free. Current debuggers for non-deterministic programming languages do not allow such exploration because they only follow a single path of many possible execution paths. In this paper, we provide a concrete recipe on how to build debuggers which allow programmers to observe *all* possible states of a non-deterministic program. To this end, multiverse debugging builds on the operational semantics of the language in which target programs are written.

2.1 Multiverse Debugging Recipe

We now give an overview of the basic recipe for defining the semantics of a multiverse debugger:

1. Define the operational semantics of the base language, a language which can specify programs that exhibit non-deterministic behavior.
2. Define the operational semantics of the debugger in terms of the base language semantics. This implies to:
   a. define a debugger configuration, which includes the state the debugger needs to maintain to debug a target program.

b. define the debugging operations that the debugger offers to developers to interactively explore the target program, e.g. by pausing/resuming program execution on breakpoints, or performing step-by-step execution of the target program.

In this paper, we apply this recipe to define the semantics of two multiverse debuggers. First, in Section 3, we apply this recipe to build a debugger for a small language called λ_{amb}. Afterwards, in Section 6, we show that our technique scales for a mid-size actor-based language called AmbientTalk [62], a prototype-based object-oriented language with an event loop concurrency model featured by mainstream languages such as JavaScript.

We believe that those two steps are general enough to be applicable to a wide range of programming languages. Applying this recipe to other programming languages consists of identifying where and how non-determinism originates. This is, however, tied to the language's concurrency model. The behavior and properties of concurrent entities (e.g. threads, transactions, actors) differ and hence these properties should be carefully considered when defining the operational semantics of the multiverse debugger.

2.2 Multiverse Debugging Main Challenges

It is our vision that the foundations explained in this paper, places multiverse debugging where research in program analysis and verification was three decades ago. At that time, static techniques were a new brave idea which could only be used to verify relatively small programs [12]. Likewise, the approach towards multiverse debugging explained in this paper is currently only feasible for relative small programs. Nevertheless, we hope that further research can be spawned from the seed we plant here to expand on what is possible today.

The main challenge of our approach is the growth in the number of states; the number of possible states increases exponentially by every non-deterministic step that is chosen in the program. This problem, called *state explosion*, is a well-researched problem in the context of program analysis and verification [61]. Multiverse debugging also suffers from this problem. It is, however, essential to make the complexity of these non-deterministic programs explicit to the programmer so that actions can be taken. We believe that providing a recipe on how to build multiverse debuggers is an important first step which gives developers the tools to explore *parts of this state space* interactively. After all, being able to inspect part of this enormous state space is better than to have a debugging tool which can only explore one execution path without any guarantees that the path being explored triggers the bug.

Symbolic execution and model checking have studied scalable solutions for the state explosion problem [17, 14, 44, 41]. Future research is needed to investigate how to adopt those techniques in a debugging tool to increase the scale on which multiverse debugging can be applied. In section 8.2 we further compare multiverse debugging with symbolic execution and model checking.

In our proof of concept implementation, we apply two techniques to help the programmer to keep an overview of the state space. First, we do not blindly explore all the possible states but let the developer decide which states to explore next, either explicitly or by using multiverse breakpoints. We believe this makes multiverse debugging comparable to bounded model checking [9]. Second, whenever two states are syntactically the same we merge those states into one node. Depending on the programming language other means of equality could be applied to further reduce the amount of states exposed to the programmer.

3 Multiverse Debugging for Ambiguous Programs

In this section, we apply the multiverse debugging recipe to debug ambiguous programs written in λ_{amb}. In Section 3.1, we specify the base language semantics (step 1) and Section 3.2 defines the semantics of a multiverse debugger on top of it (step 2). To simplify the exposition and focus on the core idea of multiverse debugging, we do not model any breakpoints, stepping commands nor user interaction with the debugger in this section. They are detailed later when we apply the idea of multiverse debugging for actor-based programs in Section 4.2. There, we will specify the semantics of a base language in which to write actor-based concurrent programs (step 1), and Section 6 describes the semantics of a multiverse debugger on top of it including multiverse breakpoints and stepping commands (step 2).

3.1 Syntax and Operational Semantics of the Base Language λ_{amb}

We now show how the multiverse debugging idea can be applied to the λ_{amb} calculus, a small functional language. Non-determinism in this language is introduced by a variation of McCarthy's ambiguity operator [47] called `amb` which behaves as a non-deterministic choice. Intuitively, when this operator is applied to a number of arguments it returns one of them in an unpredictable way.

Figure 1 gives an overview of the syntax and reduction rules of the λ_{amb} calculus. Expressions consist of numbers, addition, and the `amb` operator. We define an evaluation context E which dictates a left to right evaluation order of the arguments. The only values v in the language are numbers. The ADD rule shows how the addition of two numbers reduces and the AMB rule shows the `amb` operator non-deterministically picks one of its arguments.

$$\begin{array}{lcll} e & ::= & (+\ e\ e)|(amb\ e\ e)|number & \text{Expressions} \\ E & ::= & (+\ E\ e)|(+\ v\ E)|(amb\ E\ e)|(amb\ v\ E) & \text{Context} \\ v & ::= & number & \text{Values} \end{array}$$

$$\text{(ADD)}\quad \frac{n = \lfloor n_1 + n_2 \rfloor}{E[(+\ n_1\ n_2)] \rightarrow_{amb} E[n]} \qquad \text{(AMB)}\quad \frac{e_x \in [e_1, e_2]}{E[(amb\ e_1\ e_2)] \rightarrow_{amb} E[e_x]}$$

Figure 1 Semantic entities and reduction rules of the λ_{amb} calculus.

To get an intuition of the λ_{amb} calculus, consider the evaluation graph of the program `(+ (amb  1 2) (amb 3 4))` shown in Figure 2. While in a deterministic evaluator there is at most one applicable rule for each expression in a non-deterministic evaluator it is possible that multiple reduction rules apply for the same expression. In our example, this is clearly the case. In the start state, there are two possible reductions leading to two execution paths the program could take. We denote a *universe* to each distinct state in which a program can be. In this example, the top universe denotes the state in which the `amb` operator selected the value `1` while in the bottom universe it chose `2`. For these two universes, there are again two possible successor universes possible by choosing between number `3` and `4`. Finally, each of these universes can be reduced by applying the ADD rule leading to three possible end universes.

3.2 Syntax and Operational Semantics of the Debugger D_{amb}

Armed with the semantics of our non-deterministic language λ_{amb}, we can now define a multiverse debugger for this base language, which allows us to pause a program and resume its evaluation until it reaches an end state. Resuming a program corresponds to a user stepping through the program, expression by expression.

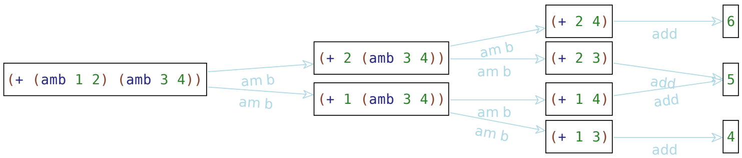


Figure 2 Multiverse evaluation graph of a λ_{amb} program.

Figure 3 gives an overview of D_{amb}, the semantics of a debugger for the λ_{amb} calculus. We first define the debugger configuration that keeps track of the state that the debugger needs to store to debug a target λ_{amb} program. In this case, the debugger is either paused or has resumed execution evaluating an expression at a time. The debugger configuration is thus a pair which consists of a *state* label (either STEP or PAUSED) and a λ_{amb} expression e.

The debugger operations are defined by two reduction rules. The STEP rule takes one evaluation step of the enclosing λ_{amb} expression e and transitions the debugger state by changing the *state* label from STEP to PAUSED. This means, we take one evaluation step, and yield to the debugger, where a user could inspect the program. Though, for simplicity, the only other operation our debugger has is the RESUME rule, which transitions a paused program back to the step state.

$$\begin{array}{rcll} state & ::= & \text{STEP} \mid \text{PAUSED} & \text{Debugger State} \\ e_d & ::= & (state,\ e) & \text{Debugger Configuration} \end{array}$$

$$(\text{STEP})\ \frac{e \ \rightarrow_{amb}\ e'}{(\text{STEP},\ e) \rightarrow_{debug} (\text{PAUSED},\ e')} \qquad (\text{RESUME})\ \frac{}{(\text{PAUSED},\ e) \rightarrow_{debug} (\text{STEP},\ e)}$$

Figure 3 Semantic entities and reduction rules of the D_{amb} calculus.

As an example of a multiverse debugging session, let us execute the program shown in Figure 2 in D_{amb}. The resulting session is shown in Figure 4. It starts by applying the STEP rule on the `(+ (amb 1 2) (amb 3 4))` expression. The first step will cause the reduction of the AMB rule in λ_{amb} for the `(amb 1 2)` expression. This leads to two possible reductions the debugger could take, i.e. one universe in which the AMB rule reduces to `1` and one in which it reduces to `2`. This means stepping to a next state is non-deterministic. By defining the debugger operations in terms of the non-deterministic evaluator of the base language, we automatically obtain a multiverse debugger, i.e. a step does not lead to one possible next universe but to a set of universes.

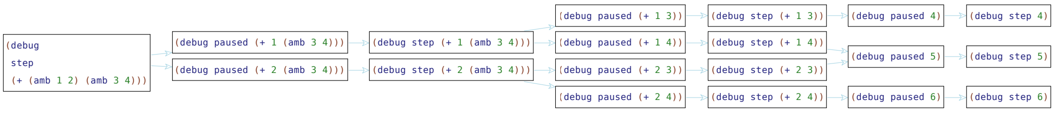


Figure 4 Multiverse debugging graph of a λ_{amb} program.

While this multiverse debugger is simplistic, it already shows two important characteristics. First, *all* evaluation steps observed in the base-level semantic are also observed in the multiverse debugger. This means that when programmers debug their programs in the multiverse debugger, the error will manifest in the debugger. Second, there are no states in the debugger which are not observed in the base-level semantics. This means that the act of debugging the program does not introduces any state (and thus also no bugs) which are not observed in the base-level semantics. As such, a multiverse debugger is probe-effect free.

4 Communicating Event Loops (CEL)

The overall goal of this work is to improve the debugging of parallel and concurrent programs. To this end, we now apply the multiverse debugging recipe defined in Section 2.1 to create a multiverse debugger for actor-based concurrent programs. To this end, we first need to specify a base language in which target programs will be written in. In this work, we use AmbientTalk [62], a distributed programming language originally designed to develop mobile peer-to-peer applications. AmbientTalk is a prototype-based object-oriented language featuring a concurrency model based on Communicating Event Loops (CEL). The language paradigm is featured by mainstream languages such as JavaScript, the Proxy API of which was actually inspired by AmbientTalk's reflective model [50].

In this section, we first describe the necessary background information on Communicating Event Loops.We then apply the first step of the multiverse debugging recipe by defining the operational semantics of AmbientTalk in Section 4.2.

4.1 Communicating Event Loops Concurrency Model

The Communicating Event Loops is a non-blocking variant of the actor model [32] first introduced by the E language [49]. The model was also adopted by languages such as AmbientTalk [62], Newspeak [11], and it is embraced by the asynchronous programming model of JavaScript and Node.js [58].

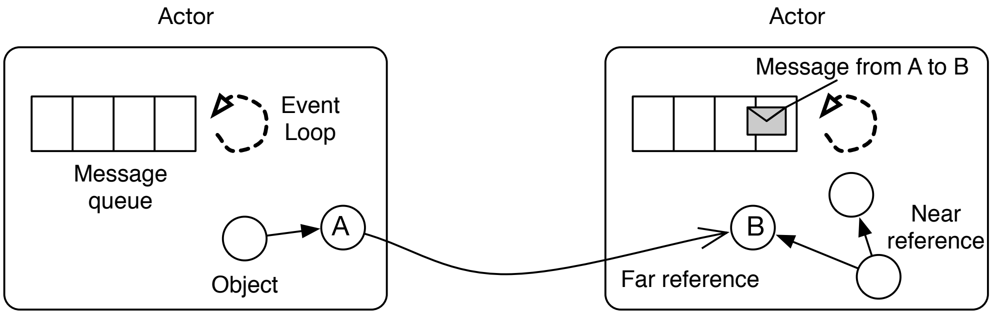


Figure 5 Overview of the CEL model (from [62]).

Figure 5 shows an overview of the CEL model. Each actor is a container of objects, a message queue (or mailbox), and an event loop (or thread of control). An actor executes sequentially messages from its mailbox, i.e. messages are processed one by one in order of arrival. The processing of one message by an actor defines a *turn*. Actors have exclusive access to their mutable state. This means that each object is owned by one actor and only the owner can directly access it. Communication with objects owned by other actors happens using asynchronous messages via *far references*. When a far reference receives a message, it forwards it to the mailbox of the actor owning the object. Objects passed as arguments in asynchronous message sends are parameter-passed either by far reference, or by (deep) copy.

An asynchronous message send immediately returns a *future* (also known as a promise). A future is a placeholder for the result that is to be computed. The future itself is an object, which can receive asynchronous messages. Those messages are accumulated within the future object and forwarded to the result value once it is available. Once the result value is available, the future is said to be *resolved* with the value. Programmers can register a block of code with a future which is asynchronously executed when the future becomes resolved. Access to the result thus happens in a *non-blocking* way.

4.2 Syntax and Operational Semantics of the AmbientTalk Language

Recall that in order to build a multiverse debugger for non-deterministic concurrent programs, we first need to define the operational semantics of the base language (step 1 of the recipe in Section 2.1). In this work, we employ the semantics of the AmbientTalk language, i.e the *Featherweight AmbientTalk* (AT^f) semantics [62]. It formalizes common features of the CEL model such as actors, objects, blocks, non-blocking functions and asynchronous message sending. Non-determinism in the CEL model is exhibited in the order in which actors process messages. Non-blocking futures also introduce additional non-determinism as messages sent to futures, while futures are not resolved, messages are forwarded to the result value once it is available.

The core calculus of AT^f consists of 30 evaluation rules (excluding helper functions). Considering that Featherweight Java [34], a minimal core calculus for Java and GJ, only has 10 evaluation rules, we believe that the AmbientTalk semantics should be not be considered a small language but at least a mid-size one. For brevity, we sketch only the parts of AT^f that are necessary to follow the contributions of this work and refer to Van Cutsem et al. [62] for the complete semantics. In a nutshell, AT^f specifies that actors evaluate messages as expressions to obtain a result value.[1] It is based on a small step operational semantics. This means that the representation of each of the steps of the program execution is atomic, i.e. there is no intermediate execution steps. This is useful because it is possible to get all possible states of the evaluation of a non-deterministic program.

$K \in$ **Configuration**	::=	$\overline{a}$	Actor configurations
$a \in$ **Actor**	::=	$\mathcal{A}\langle \iota_a, O, Q_{in}, e \rangle$	Actors
Object	::=	$\mathcal{O}\langle \iota_o, t, F, M \rangle$	Objects
$t \in$ **Tag**	::=	$\textsc{o} \mid \textsc{i}$	Object tags
Future	::=	$\mathcal{F}\langle \iota_f, Q_{in}, v \rangle$	Futures
Resolver	::=	$\mathcal{R}\langle \iota_r, \iota_f \rangle$	Resolvers
m $\in$ **Message**	::=	$\mathcal{M}\langle v, m, \overline{v} \rangle$	Messages
$Q_{in} \in$ **Queue**	::=	$\overline{\mathrm{m}}$	Inbox queues
$M \subseteq$ **Method**	::=	$m(\overline{x})\{e\}$	Methods
$F \subseteq$ **Field**	::=	$f := v$	Fields
$v \in$ **Value**	::=	$r \mid \text{null} \mid \epsilon$	Values
$r \in$ **Reference**	::=	$\iota_a.\iota_o \mid \iota_a.\iota_f \mid \iota_a.\iota_r$	References
$e \in E \subseteq$ **Expr**	::=	$\ldots \mid e \leftarrow m(\overline{e}) \mid r$	Runtime Expressions

$$o \in O \subseteq \textbf{Object} \cup \textbf{Future} \cup \textbf{Resolver}$$
$$\iota_a \in \textbf{ActorId}, \iota_o \in \textbf{ObjectId}, \iota_n \in \textbf{NetworkId}$$
$$\iota_f \in \textbf{FutureId} \subset \textbf{ObjectId}, \iota_r \in \textbf{ResolverId} \subset \textbf{ObjectId}$$

Figure 6 Semantic entities of the AT^f calculus.

Figure 6 shows the semantic entities of the operational semantics of AT^f. A configuration K represents the set of actors that are executed concurrently in the program. An actor is represented by an identity ι_a, a set of objects O, an inbox queue Q_{in} that stores the messages to be processed and an expression e the actor is currently evaluating. An object O consists of an identity ι_o, a tag t and a set of fields F and methods M. The tag distinguishes between objects passed by reference O, and passed by copy I. A future consists of an identity ι_f, a queue for the pending messages Q_{in} and a resolved value v. A resolver object allows to

[1] In this work we use the subset of Featherweight AmbientTalk for concurrency, i.e. without the notion of networks for distribution.

assign a value to its unique paired future and as such, it consists of an identity ι_r and the identity of its corresponding future ι_f. A message m is represented by an identifier ι_m, a receiver value v, a method name m and a sequence of arguments values $\overline{v}$. References to objects r consist of an identifier for the actor ι_a owning the referenced value and a local component that can be ι_o, ι_f or ι_r. The local component indicates that the reference refers to either an object, a resolver or a future. An expression e can include references r or an asynchronous message send $e \leftarrow m(\overline{e})$.

5 Multiverse Debugging for Actor-based Programs

Having the operational semantics of AmbientTalk, we can now apply the second step of the multiverse debugging recipe to create a multiverse debugger for actor-based programs by defining the operational semantics of the debugger in terms of the AmbientTalk ones. Before detailing this operational semantics in Section 6, we informally describe the debugger's breakpoints and stepping semantics. Additionally, we show a debugging session in the resulting multiverse debugger called *Voyager*.

5.1 Breakpoint-based Debugging for Actor-based Programs

As explained before, multiverse debugging allows developers to interactively explore a target program (step 2.b. in the multiverse debugging recipe of Section 2.1). In this section we describe the two main features that multiverse debugging borrows from breakpoint-based debuggers to enable such an interactive exploration of the program's state, i.e. breakpoints and stepping operations.

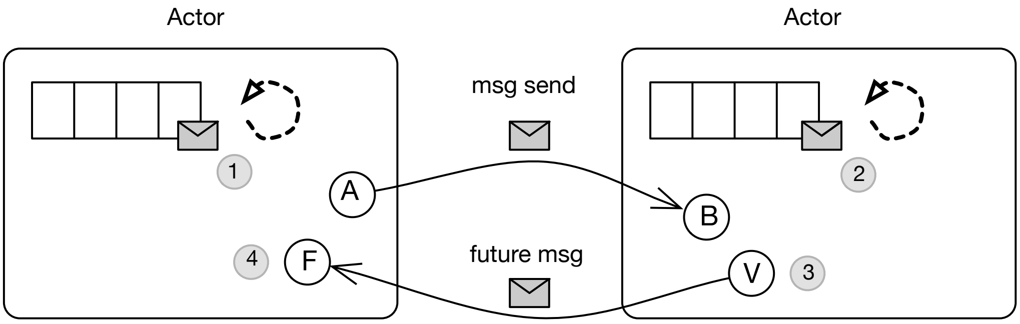


Figure 7 Points of interest for debugging actor-based programs.

A *breakpoint* defines a point of interest in a program in which to pause execution for further inspection. The main idea of breakpoints is to allow developers to observe the effects on an operation that may interleave with other operations in the system. Since turns run till completion, operation interleavings in CEL programs happen at message level. As such, breakpoints need to be applied to asynchronous message passing.

As a general principle, we consider the point in time right before and right after a message is processed as relevant for breakpoints. Figure 7 shows the points of interests involved in an asynchronous message send in CEL. When object A sends a message to object B, it is first placed in the sender actor's mailbox (point 1) and a future object F is immediately returned. The message is then sent to the actor hosting the receiver object B, and is placed in its mailbox (point 2). After the value for the future is computed, a message with the result value

V is placed in the receiver actor's mailbox (point 3) and sent back to the actor hosting the sender object. The message carrying the result value is then placed in the sender's mailbox (point 4) and the future resolution listeners are finally executed.

Stepping is a debugger feature that allows developers to follow the execution of a program between various points of interest. In sequential programs, stepping operations typically allow to step through the program line by line. In CEL programs, stepping also needs to allow developers step through program execution concurrently, i.e. let them follow the execution between the points of interest involved in asynchronous message passing. Stepping operations can thus be applied at each of the points shown in Figure 7, and it will allow the developer to step to the next point of interest. For example, the program may be paused at point 2, before the receiver actor has processed a message. A possible stepping operation is to *step to the next turn*, which will let the actor process the message sent by A, and halt before processing the next message, i.e. at the beginning of the turn.

In prior work we have explored catalogs of breakpoint types and stepping operations for actor-based programs [29, 46]. These debugging operations are used in breakpoint-based debuggers for CEL programs written in AmbientTalk and SOMNS[2], respectively. In this work, we apply the debugging operations that we proposed in [46] to our multiverse debugger.

5.2 Voyager: a Multiverse Debugger for AmbientTalk Programs

To showcase the use of a multiverse debugger for AmbientTalk programs, we implemented a tool called *Voyager*, which allows developers to interactively explore a target program through the debugger operational semantics. Voyager is a web application build on top of PLT-Redex, which executes the operation semantics of a multiverse debugger. Both the operational semantics of the debugger and AT^f are implemented in PLT-Redex. Target programs are written in PLT-Redex and can be loaded in Voyager. Voyager then asks PLT-Redex to reduce the program according to the debugger semantics, which results in the reduction graph for the program. All states in this graph are stored in a graph database[3] for easy manipulation and exploration of the reduction graph.

5.2.1 Debugging a Sample Program

We now show a debugging session in Voyager for the AmbientTalk program depicted in Listing 1. The program shows an interaction between 3 actors: a `math` actor (created in Line 20, and two client actors, `client1` (created in Line 21) and `client2` (Line 22). The `math` actor (Lines 1 to 7) understands the messages `double`, which doubles its argument and stores the result for further operations, as well as the `getResult` message, which returns the result of a number of operations. After creating the three actors, the program sends a `start` message to both client actors (Lines 23 and 24). As a result, `client1` sends a `double(12)` message followed by a `getResult` one. Concurrently, `client2` sends a `double(33)` message to the `math` actor as well.

Despite being a simple program, it contains a *bad message interleaving* bug [60], which is common for actor-based programs. It is possible that `client1` gets the result of doubling `33` instead of doubling `12`. Table 1 shows all possible interleavings that the program exhibits.

[2] SOMNS is a Newspeak implementation build on top of the Truffle platform. https://github.com/smarr/SOMns

[3] Our prototype uses ArangoDB. https://www.arangodb.com/

```
 1 def makeMath() {
 2     actor:{
 3         def result := 0;
 4         def double(x){result := x+x};
 5         def getResult(){result};
 6     }
 7 };
 8 def makeClient1(math){
 9     actor:{ |math|
10         def start(){
11           math<-double(12);
12           when: math<-getResult()@FutureMessage becomes: {|res|
13               system.println(res);
14           }}}
15 };
16 def makeClient2(math){
17     actor: { |math|
18         def start(){ math<-double(33) }}
19 };
20 def math := makeMath();
21 def client1 := makeClient1(math);
22 def client2 := makeClient2(math);
23 client1<-start();
24 client2<-start();
```

Listing 1 AmbientTalk program containing a bad message interleaving bug.

It also depicts the message sender for each message. Line 13 would print the result value, which is `24` for the correct interleavings and `66` for the faulty one.

Table 1 Message interleavings for the AmbientTalk program shown in Listing 1.

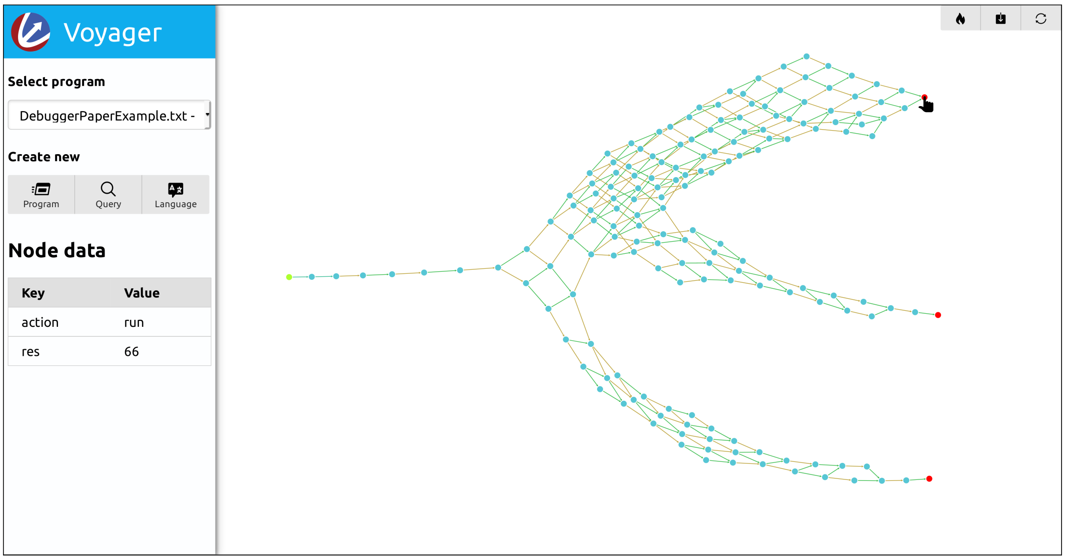
Faulty Interleaving	Correct Interleaving	Correct Interleaving
client 1 - `double(12)`	client 1 - `double(12)`	client 2 - `double(33)`
client 2 - `double(33)`	client 1 - `getResult()` → 24	client 1 - `double(12)`
client 1 - `getResult()` → 66	client 2 - `double(33)`	client 1 - `getResult()` → 24

5.2.2 Overview of a Debugging Session

Taking the faulty interleaving of our example program of Listing 1 as example, a developer may choose to explore the issue in Voyager and identify why the unexpected result is `66`. Thus, the developer needs to find the cause of the bad message interleaving exhibited by the program. For a screencast of the debugging session, we refer the reader to `https://tinyurl.com/VoyagerYoutube`.

Figure 8 shows the Voyager UI. The left panel allows developers to upload the target program to debug (either by selecting an existing file or creating a new one directly), and shows information on the selected node in the "Node data" section. The right panel shows the reduction graph for the target program. In this case, Voyager shows all possible universes for the sample program. The end states of the program are shown in red. As expected (cf. Table 1), there are three possible end states. "Node Data" shows the information for the end state under the cursor. The selected state corresponds to the end state of an execution path with the faulty interleaving since the result stored in `res` is `66`.

Let us now start a debugging session to understand how we arrived at the result being `66`. Since `math` actor is the central point of synchronization in the program, we set a breakpoint that pauses the program's execution each time the `math` actor receives a message (before processing it). In Voyager, this is called a *message receiver breakpoint*; its semantics are shown in Section 6. With this breakpoint activated, we run the program again in Voyager.



Select program

DebuggerPaperExample.txt -

Create new

Node data

Key	Value
action	run
res	66

Figure 8 Overview of the Voyager tool.

Figure 9(a) shows the new reduction graph, with the execution paused once the breakpoint was reached. The blue nodes denote the state of a running debugger executing the program. When the message receiver breakpoint on the `math` actor is reached, in one of the universes the debugger halts the execution (in that universe) and highlights the node in pink. As a result, the evaluation of the underlying AmbientTalk program pauses. The magnifier glass shows the pink node representing the triggering of the `Message-Receiver-Breakpoint` rule (later detailed in Section 6). At this point in the execution, a developer can click on the node to inspect the state, resume execution, or execute one of the step commands. The debugging operations applicable at this point are accessible by means of a radial menu.

For this example, we make Voyager step to the next turn of the `math` actor. Figure 9(b) shows the resulting graph after applying that stepping command. The first pink node in Figure 9(b) corresponds to the node shown with the magnified glass in Figure 9(a). Notice that the dashed lines are used to indicate user interaction. The new graph shows how the debugger stops again at all possible universes in which the `math` actor receives the second message.

The initial expectation of a developer may be that the next message in the mailbox of the `math` actor will always be the `getResult` message. However, in Figure 9(b) we see that there are two possible kinds of universes the base level program can evolve to. Inspecting the actors inbox reveals that in some universes the next processed message is from `client2`. As shown in the figure the top universe corresponds to the interleaving in which the `double` message from `client2` arrives first, and the bottom universe corresponds to the interleaving in which the `getResult` message from `client1` correctly arrives after the doubling message. At this point, a developer sees that the initial expectation does not hold and a fix can be developed to account for this bad interleaving.

One might be tempted to think that the program could be debugged by a traditional concurrent debugger using a deterministic message order. While single-stepping individual messages in a deterministic pattern would create a deterministic message order, traditional concurrent debuggers only keep track of one universe. Because such a debugger simply picks one of many universes determined by the execution order of messages, and it may not be the

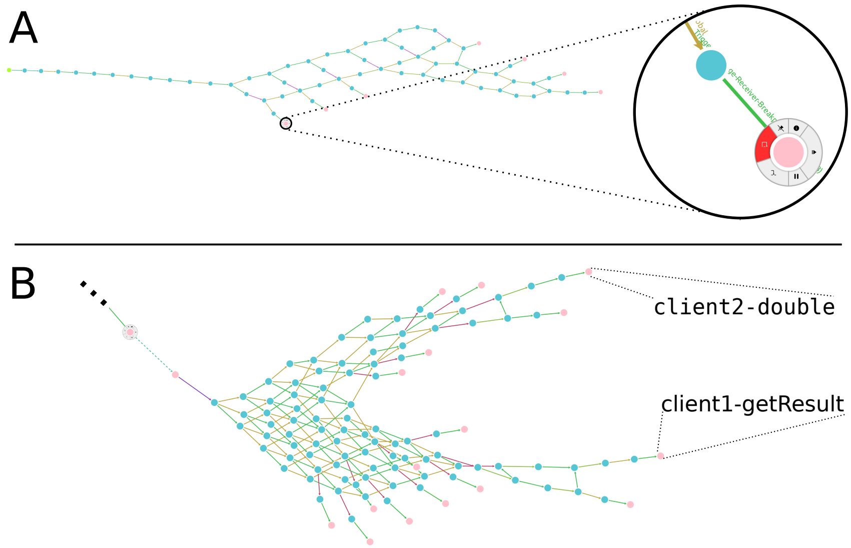


Figure 9 A debugging session in Voyager for the program displayed in Listing 1.

universe in which the bug manifests. Hence, traditional concurrent debuggers do not avoid the probe effect. In contrast, multiverse debugging allows developers to explore all possible non-deterministic execution paths of a concurrent application. In order to steer the state exploration, Voyager provides query facilities that we detail below.

5.2.3 Querying the state graph

As previously mentioned, the Voyager debugger stores the state graph in a graph database which we can use to query the state of the multiverse graph. The left panel of the Voyager UI (Figure 8) has a button to create new queries. Figure 10(a) shows the original graph for the program. Figure 10(b) shows a more simplified view on the multiverse after applying a query to the original graph that filters all paths except the shortest path from the start node to all end nodes, i.e., nodes that cannot be reduced any further.

Listing 2 shows the code for the query that generates the simplified graph shown in Figure 10(b) with the shortest path to the three possible end states of our sample problem. The used query language is AQL[4] using Bind parameters[5] (`@start, @graph, and @@nodes`). The query computes the shortest path to all end states as follows:

- First, it finds all stuck nodes, by querying the database for nodes that have been marked as *stuck*. The `FOR` operation returns an array of the values that have been returned by `RETURN` in its body.
- Second, Line 5 to Line 9 find the shortest path from the start node to each stuck node using ArangoDB's `SHORTEST_PATH`. This primitive returns an array of the nodes and edges on the shortest path between two nodes in a graph. Since the `SHORTEST_PATH` is used for each of the stuck nodes, we end up with an array of arrays which is then flattened. The `path` variable now holds an array of nodes and edges on the shortest path from the start node to a stuck node.
- Finally, in order to visualize the paths, this array is converted to an object that lists the edges and nodes to be displayed separately (Line 12 to Line 15).

[4] https://docs.arangodb.com/3.4/AQL/Fundamentals/Syntax.html
[5] https://docs.arangodb.com/3.4/AQL/Fundamentals/BindParameters.html

```
// make an array containing all stuck nodes
LET stuckNodes = (FOR n in @@nodes FILTER n._stuck == true RETURN n)

// find the path to each of them and join the results
LET path = FLATTEN(
    FOR target IN stuckNodes
        FOR n,e IN OUTBOUND SHORTEST_PATH
            @start TO target._id GRAPH @graph
                RETURN {n,e})

// convert to the needed format
RETURN {
    edges: (FOR d in path FILTER d.e != null RETURN DISTINCT d.e),
    nodes: (FOR d in path FILTER d.n != null RETURN DISTINCT d.n)
}
```

Listing 2 AQL query to search the shortest path from the start state to all possible end states of a program.

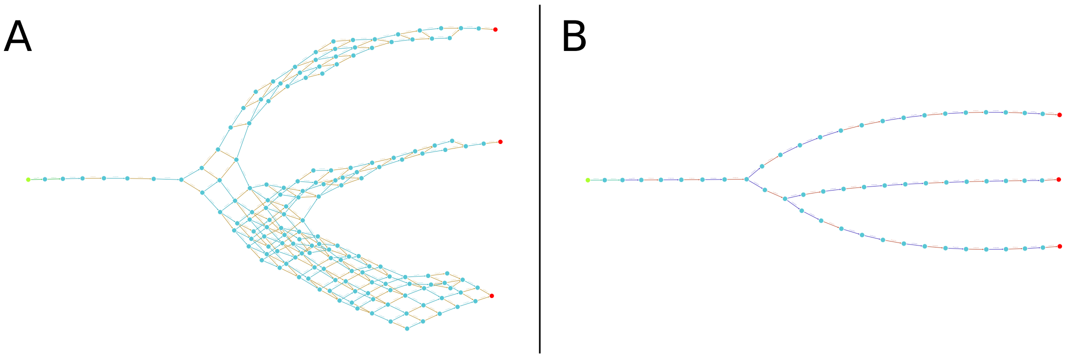


Figure 10 Application of a shortest-path-to-end-states query to the program displayed in Listing 1.

The breakpoints and stepping operations combined with the query facilities of Voyager provide an interactive experience of browsing the multiverse graph of a program to find the root cause of bugs. It is important to note that in contrast to static analyses, a multiverse debugger allows developers to explore and query states of the concrete program execution interactively. This enables developers to focus on relevant elements and thereby directly steer the state exploration.

6 Syntax and Operational Semantics of the Voyager Multiverse Debugger

In this section we finally apply step 2 of the multiverse debugging recipe and describe the syntax and operational semantics of our multiverse debugger, Voyager. We first describe the general strategy of the debugger to be able to debug AmbientTalk programs, and we then detail the elements of the *debugger configuration* of Voyager (step 2a) and the key mechanisms for supporting breakpoints and stepping operations as the one described in the previous section (step 2b).

6.1 Overview of the Debugger Semantics

The Voyager debugger keeps track of both the state of the underlying AmbientTalk program and its own state. The semantics of the debugger consists of a set of reduction rules which transition from one debugger state to the next one. In order to model the catalog of breakpoints and stepping operations that we proposed in [46], we define the debugger state $\mathcal{D}$, which consists of six elements, B_p, B_c, d_s, C, A_s, K.

- The first two elements B_p and B_c are respectively the list of pending breakpoints and the list of already checked breakpoints.
- To keep track of which action the debugger is performing, the debugger configuration contains a debugger state d_s for representing the state *run* and *pause*. When the debugger is in the *run* state it verifies whether there is an applicable breakpoint. When a breakpoint hits, the debugger transitions itself to the *pause* state and halts execution.
- To model the possible debugging operations offered to the user, e.g. stepping, resume, and pause, the debugger state contains a list of commands C.
- To keep track of the state of the actor the debugger configuration contains a map A_s.
- Finally, K is the state of the actor configuration being debugged, i.e. the state of the AmbientTalk program.

The general form of the reduction rules of the Voyager debugger consists of transitions between debugger states:

$$\mathcal{D}\langle B_p, B_c, d_s, C, A_s, K\rangle \to_d \mathcal{D}\langle B'_p, B'_c, d'_s, C', A'_s, K'\rangle$$

Note that the transition relation of the debugger is denoted by $\to_d$ while the transition rules of the base language are defined as $\to_k$. The evaluation strategy of the debugger consists of traversing the list of pending breakpoints B_p one-by-one from left to right, moving the debugger to a stopped state when a breakpoint hits. When a breakpoint does not apply for the current state of the actor configuration, it is moved from the list of pending breakpoints to the list of checked ones. When there are no pending breakpoints left the debugger instructs the actor configuration to take one step and swaps the checked breakpoints with the pending breakpoints, this continues till either a breakpoint is hit or an end state is reached.

6.2 Syntax of the Debugger Semantics

Figure 11 shows the semantics of two elements that we needed to extend in the AmbientTalk semantics AT^f presented in Figure 6.

- The first element corresponds to the message entity m, which we extended with an id ι_m to identify the message.
- The second element corresponds to the send expression $e \leftarrow_{id} m(\overline{e})$ that we extended also with an identifier id to determine which message is breakpointed. This identifier is needed because different types of breakpoints can be set on the same message.

m $\in$ **Message**	$::=$	$\mathcal{M}\langle \iota_m, v, m, \overline{v}\rangle$	Messages
$e \in E \subseteq$ **Expr**	$::=$	$\ldots \mid e \leftarrow_{id} m(\overline{e})$	Runtime Expressions

Figure 11 Extended semantic entities in AT^f for debugging in Voyager calculus.

Figure 12 shows an overview of the elements of the Voyager calculus. More concretely, it includes all the entities of the semantics that are needed by the six elements of the debugger configuration D, i.e. b_u, b_t, c, c_s, a_s, t_{ub}, t_{tb}, t_c.

$d \in$ **Debugger**	::=	$\mathcal{D}\langle B_p, B_c, d_s, C, A_s, K\rangle$	Debugger configurations
$B_p \in$ **Pending breakpoint**	::=	$\overline{b_u \mid b_t}$	Pending breakpoints
$B_c \in$ **Checked breakpoint**	::=	$\overline{b_t}$	Checked breakpoints
$d_s \in$ **Debugger state**	::=	run | pause	Debugger states
$C \in$ **Command**	::=	$\overline{c}$	Commands
$A_s \in$ **Actor state map**	::=	$\overline{c_s}$	Actor state map
$b_u \in$ **User breakpoint**	::=	$\mathcal{B}\langle t_{ub}, \iota_i\rangle$	User Breakpoints
$b_t \in$ **Trigger breakpoint**	::=	$\mathcal{B}\langle t_{tb}, \iota_a, \iota_i\rangle$	Trigger Breakpoints
$c \in$ **C**	::=	$\mathcal{C}\langle t_c\rangle \mid \mathcal{C}\langle t_c, n\rangle$	Commands
$c_s \in$ **Current actor state**	::=	$\mathcal{CS}\langle \iota_a, a_s\rangle$	Current actor state
$a_s \in$ **Actor state**	::=	run | pause | hold | step n	Actor states
$t_{ub} \in$ **User breakpoint tag**	::=	msb | mrb	User breakpoint tags
$t_{tb} \in$ **Trigger breakpoint tag**	::=	mrb-trigger	Trigger breakpoint tags
$t_c \in$ **Command tag**	::=	step-next-turn ι_a | resume | pause	Command tags

$\iota_i \in$ **BreakpointId**

Figure 12 Semantic entities of the Voyager calculus.

- To define a breakpoint b_u we use a two-element tuple consisting of a breakpoint tag t_{ub} and an expression id ι_i.
- Additionally, we define breakpoints at the level of the debugger semantics, i.e. breakpoints which are defined by the semantics itself rather than by the developer debugging a target program. We call these breakpoints *trigger* breakpoints to distinguish them from the *user* ones aforementioned. A trigger breakpoint b_t consists of a tuple of three elements, a breakpoint tag t_{tb}, an actor id ι_a, and an expression id ι_i.
- A command c is defined by a tag t_c. In the case of a step to next turn we need to define also the number of steps n the command needs to take in the evaluation of the program, i.e. $\mathcal{C}\langle c, step\ n\rangle$.
- The map of actors A_s keeps a list of pairs $\mathcal{CS}\langle \iota_a, a_s\rangle$ consisting of the id of the actor ι_a and the current state a_s.
- An actor can be in *run*, *pause* or *hold* state. In addition, an actor can have a state *step n*.
- The breakpoint tags t_{ub} indicate the tags a user can identify when defining a breakpoint.
- The trigger breakpoint tags t_{tb} correspond to the tags built-in in the semantics to actually trigger the breakpoint.
- The command tags t_c refer to the debugging commands the user can specify to debug the program, i.e. several stepping operations and resume/pause commands.

6.3 Operational Semantics of the Voyager Debugger

Having defined the syntax of the Voyager debugger and the debugger configuration (step 2a of the multiverse debugging recipe), we can now define the semantics of the debugger operations that Voyager offers to developers to interactively explore the target program (step 2b).

The reduction rules of Voyager can be separated in five groups:

1. Reduction rules for modeling the connection of the debugger with the base level language (cf. Section 6.3.1)

2. Reduction rules for breakpoints (cf. Section 6.3.2), including rules needed to model breakpoints which require trigger breakpoints for their functioning.
3. Bookkeeping reduction rules (cf. Section 6.3.3), i.e. rules that are related to the actor state when breakpoints are not applicable and when new actors are created.
4. Reduction rules for the stepping operations (cf. Section 6.3.4), consists of the rules for stepping commands that can be applied on the level of messages, futures, and turns.
5. Reduction rules for other debugging commands (cf. Section 6.3.5), i.e. rules that will resume and pause the program's execution.

For brevity, the following sections focus on the rules required to define the semantics for the message receiver breakpoint and the *step to next turn* command employed in the debugging session shown in Section 5.2. The complete set of reduction rules is included in Appendix A.

6.3.1 Connection with the Base Level Language

Recall that the semantics of a multiverse debugger is defined in terms of the underlying base language semantics, AT^f in the case of Voyager. Two reductions rules (shown below) govern the connection of Voyager's semantics with AT^f: cel-step-global and cel-step-local. The cel-step-global rule transitions the actor configuration K to the actor configuration K' by applying the global AmbientTalk reduction relation ($\rightarrow_k$). This relation governs all the actor transitions rules that affect two or more actors, i.e. sending asynchronous messages and creating new actors. The cel-step-local rule, on the other hand, non-deterministically picks an actor a from the actor configuration K and transitions it to an actor a' by applying the local multi-step AmbientTalk relation $\xrightarrow{*}_a$. This reduction relation applies one or more single-step local reduction which can be applied to the actor, all these single-step reductions are deterministic. Finally, we require that the actor which we transition is in a local running state and update it accordingly, i.e. when the actors local state is (*step* n) the *update* meta-function will update the actors state to (*step* $n-1$).

Both the cel-step-global and cel-step-local rule can only be triggered when all the pending breakpoints are checked. Note that after taking a step in AT^f, the checked breakpoints and the pending breakpoints are swapped. At certain points during the execution it could be that both cel-step-global and cel-step-local are applicable at the same time. This is intentional and is part of the non-deterministic nature of executing the AmbientTalk semantics that we want to capture in the debugger.

(cel-step-global)

$$\frac{\begin{array}{c} K \rightarrow_k K' \\ not-applicable-add-new-actor \end{array}}{\mathcal{D}\langle (), B_c, \text{run}, C, A_s, K\rangle \rightarrow_d \mathcal{D}\langle B_c, (), \text{run}, C, A_s, K'\rangle}$$

(cel-step-local)

$$\frac{\begin{array}{c} K = K' \dot{\cup} \{a\} \qquad a \xrightarrow{*}_a a' \qquad A'_s = update(A_s, a) \\ not-applicable-add-new-actor \end{array}}{\mathcal{D}\langle (), B_c, \text{run}, C, A_s, K\rangle \rightarrow_d \mathcal{D}\langle B_c, (), \text{run}, C, A'_s, K' \dot{\cup} a'\rangle}$$

6.3.2 Reduction Rules for Breakpoints

As mentioned before, the Voyager semantics features two families of breakpoints: *user breakpoints* denote breakpoints that are activated by the user while *trigger breakpoints* denote breakpoints generated by the debugger. As an example of user breakpoint consider the *message receiver breakpoint*, which we explained in the debugging session shown in Section 5.2. It halts execution of an actor before it processes a message (identified by a unique id).

Generally, we only know during program execution which actor hosts the receiver object of a message. Therefore, the debugger monitors the program and inserts a new trigger breakpoint when the id of the receiver actor becomes known. The trigger breakpoint is used by the debugger semantics to later halt the execution when the message is actually received at the receiver side.

The SAVE-MRB reduction rule below governs the semantics of transforming a message receiver breakpoint into a trigger message receiver breakpoint. When the message is about to be sent the user breakpoint $\mathcal{B}\langle mrb, \iota_i\rangle$ that is in the list of pending breakpoints, the SAVE-MRB is triggered if the actor id of the breakpoint corresponds to the actor id of the receiver actor. In this case, the breakpoint is removed from the pending list and a trigger breakpoint $\mathcal{B}\langle mrb - trigger, \iota_{a'}, \iota_i\rangle$ is added in the list of checked breakpoints. Note that the sender and receiver actors of that message continue with *run* state, but the addition of the trigger message breakpoint will make the execution of the debugger pause at the receiver actor (the actor id of which is included in the trigger breakpoint itself).

(SAVE-MRB)

$$\frac{\mathcal{A}\langle \iota_a, O, Q_{in}, e_\square[\iota_{a'}.\iota_o \leftarrow_{\iota_i} m(\overline{v})]\rangle \in K}{\mathcal{D}\langle \mathcal{B}\langle mrb, \iota_i\rangle \cdot B_p, B_c, \text{run}, C, A_s, K\rangle \rightarrow_d \mathcal{D}\langle B_p, B_c \cdot \mathcal{B}\langle mrb - trigger, \iota_{a'}, \iota_i\rangle, \text{run}, C, A_s, K\rangle}$$

The next reduction rule is TRIGGER-MRB and it governs the semantics of the trigger breakpoint added for a message receiver breakpoint. Back in the example debugging session, Figure 9 showed that the triggering of this rule resulted in the pink node under the magnifying glass. In the trigger breakpoint the id of the actor ι_a is saved to identify the actor the user wants to halt, and the ι_i is saved to identify in which message. When the message arrives in the queue of the receiver actor, the trigger breakpoint is removed from the pending list B_p and the debugger and the receiver actor changes its state to *pause*. Note that the receiver actor cannot process local operations but it can execute global ones, e.g. receive a new message from another actor. The two operations of the message receiver breakpoint for saving the information needed when the message is about to be sent and triggering the breakpoint are shown in Appendix A, i.e. SAVE-MRB and TRIGGER-MRB.

(TRIGGER-MRB)

$$\frac{\mathcal{A}\langle \iota_a, O, m \cdot Q_{in}, v\rangle \in K \qquad A'_s = A_s + \{\mathcal{CS}\langle \iota_a, pause\rangle\}}{\mathcal{D}\langle \mathcal{B}\langle mrb - trigger, \iota_a, \iota_i\rangle \cdot B_p, B_c, run, C, A_s, K\rangle \rightarrow_d \mathcal{D}\langle B_p, B_c, \text{pause}, C, A'_s, K\rangle}$$

6.3.3 Bookkeeping Reduction Rules

For each of the breakpoint triggering rules there should be a rule which instructs the debugger to move the breakpoint to the list of checked breakpoints when the breakpoint does not hit. Instead of listing all these individual rules we compressed them into one rule called BREAKPOINT-NOT-APPLICABLE which should be triggered when the breakpoint at the head of the list is not applicable. The BREAKPOINT-NOT-APPLICABLE rule is shown Appendix A.

Appendix A also includes the ADD-NEW-ACTOR reduction rule for the creation of new actors. This rule basically updates the A_s map when an actor is created.

6.3.4 Reduction Rules for Stepping Operations

Similarly to the formalisation of the message sender breakpoint, some stepping commands need to be encoded with several reduction rules. For example, the step-next-turn employed in debugging session in Section 5.2, is formalised with two reduction rules: PREPARE-STEP-NEXT-TURN and TRIGGER-STEP-NEXT-TURN. The PREPARE-STEP-NEXT-TURN rule is triggered when the debugger is in the paused state and transitions a particular actor with id ι_a from the paused state to the (*step* 1) state indicating that the actor is allowed to take exactly one local step (cf. CEL-STEP-LOCAL in Section 6.3.1).

The TRIGGER-STEP-NEXT-TURN rule is triggered when a particular actor with id ι_a is in the state (*step* 0). When this rule is triggered, the debugger moves form the *run* state to the *paused* state. At the same time, the local actor state is also changed to the *pause* state.

(PREPARE-STEP-NEXT-TURN)

$$\frac{\mathcal{A}\langle \iota_a, O, m \cdot Q_{in}, e\rangle \in K \qquad A'_s = A_s \dot{\cup} \{\mathcal{CS}\langle \iota_a, (step\ 1)\rangle\}}{\begin{array}{c}\mathcal{D}\langle B_p, B_c, \text{pause}, (StepNextTurn\ \iota_a) \cdot C, A_s\ \dot{\cup}\ \mathcal{CS}\langle \iota_a, (pause)\rangle, K\rangle \rightarrow_d \\ \mathcal{D}\langle B_p, B_c, \text{run}, (StepNextTurn\ \iota_a) \cdot C, A'_s, K\rangle\end{array}}$$

(TRIGGER-STEP-NEXT-TURN)

$$\frac{\mathcal{A}\langle \iota_a, O, m \cdot Q_{in}, v\rangle \in K \qquad A'_s = A_s \dot{\cup} \{\mathcal{CS}\langle \iota_a, pause\rangle\}}{\mathcal{D}\langle B_p, B_c, \text{run}, (StepNextTurn\ \iota_a) \cdot C, A_s\ \dot{\cup}\ \mathcal{CS}\langle \iota_a, (step\ 0)\rangle, K\rangle \rightarrow_d \mathcal{D}\langle B_p, B_c, \text{pause}, C, A'_s, K\rangle}$$

Note that other breakpoints and stepping commands can be encoded in a similar vain as we have shown for the message receiver breakpoint and step to next turn. Some of these breakpoints such as the message sender breakpoint (cf. Appendix A) are easier because they do not require any bookkeeping, i.e. all the information to pause the execution is known at the start of the program.

6.3.5 Reduction Rules for Basic Debugging Commands

Finally, we show below the rules which govern basic debugging commands to control the execution of a program, namely pause and resume. The RESUME-EXECUTION rule guarantees that the execution of the program continues from any pause state of the debugger. As such, the debugger state transits from *pause* to *run*. The rule updates the state of the local actors to *run*.

The PAUSE-EXECUTION rule halts the execution of all actors in the actor configuration, transitioning the debugger state from *run* to *pause*. The rule updates the state of the local actors to *pause*.

(RESUME-EXECUTION)

$$\frac{A'_s = run(A_s)}{\mathcal{D}\langle B_p, B_c, \text{pause}, Resume \cdot C, A_s, K\rangle \rightarrow_d \mathcal{D}\langle B_p, B_c, \text{run}, C, A'_s, K\rangle}$$

(PAUSE-EXECUTION)

$$\frac{A'_s = pause(A_s)}{\mathcal{D}\langle B_p, B_c, \text{run}, Pause \cdot C, A_s, K\rangle \rightarrow_d \mathcal{D}\langle B_p, B_c, \text{pause}, C, A'_s, K\rangle}$$

6.3.6 Discussion

There are a number of design decisions and limitations worth discussing. First, in an early prototype of the debugger semantics [59], we did not separate the global from the local AT^f reduction rules. This turned out to be problematic because it makes it hard to pause a specific actor from processing messages while still allowing it to receive messages in its inbox. Separating the global from the local semantics simplified the semantics significantly.

Second, the early prototype used the single-step operational semantics to transition the local actor semantics [59], while in the final version reported here, we are using a multi-step relation. As previously mentioned, in the actor model the only points where the non-determinism matters is when messages are being exchanged between the actors. However, when using the single-step local reduction relation a lot of additional and irrelevant non-determinism is introduced. This made working with the Voyager debugger very tedious and reduced its usefulness for larger programs. By switching to the local multi-step relation the amount of states being shown to the end user is significantly reduced while the non-deterministic behavior due to message passing is completely preserved.

Finally, it is worth noting that even though the multi-step relation alleviates the problem of growing number of states, the number of states still grows depending on the program size, as previously mentioned. Further research is needed to investigate ways to reduce the number of states without removing relevant sources of non-determinism in the program. To this end, advances in the context of static techniques like symbolic execution and model checking can be employed as starting point (cf. Section 8.2). We believe that with the current hardware evolution of multicore machines, the size of programs which can be debugged with multiverse debugging is steadily growing as well. At this point, we have used the Voyager tool to debug programs of the size of dining philosophers. Further research is also needed to investigate techniques to guide the exploration of the state graph, e.g. novel stepping semantics that work at the level of universes. Of course, applying this technique to industrial-strength languages will also required further work. But the goal would be that a traditional breakpoint-based debugger can be a foundation for such multiverse debugging.

7 Proof of Non-Interference

In this section we provide a proof of *non-interference* for the semantics of the Voyager debugger. More specifically, we prove observational equivalence between the debugger and the base language semantics. Intuitively, this means that any execution of the Voyager debugger corresponds to an execution of an AmbientTalk program, and any execution of an AmbientTalk program is observed by Voyager. Formally,

▶ **Theorem 1** (Equivalence of evaluation steps). *Let K be an actor configuration in the AT^f semantics, for which there exists a transition to an actor configuration K'. Let D be a debugging configuration for K and B_p, B_c, d_s, C, A_s elements of D such that the commands C resume all the paused actors then:*

$$(K \rightarrow_k K') \iff$$

$$(\mathcal{D}\langle B_p, B_c, d_s, C, A_s, K\rangle \rightarrow_{d_k} \mathcal{D}\langle B_p', B_c', d_s', C', A_s', K'\rangle)$$

The left handside of the biconditional relation represents the evaluation of the program in the AmbientTalk semantics AT^f, i.e. the configuration of actors K, to another program state K'. Where $\rightarrow_k$ corresponds to the evaluation regarding the reduction rules of the base language.

The right handside of the biconditional relation represents the evaluation of the program in the debugger semantics $\mathcal{D}\langle B_p, B_c, d_s, C, A_s, K\rangle$, which yields in an another debugger configuration $\mathcal{D}\langle B'_p, B'_c, d'_s, C', A'_s, K'\rangle$. Where $\rightarrow_{d_k}$ represents one or more evaluation *steps* taken by the debugger transition rules in K, until the debugger configuration $\mathcal{D}\langle B'_p, B'_c, d'_s, C', A'_s, K'\rangle$ is reached.

To prove the biconditional relation of Theorem 1 we divide our proof in two parts, which corresponds to the two implications of the relation.

Implication 1. An evaluation step in the AmbientTalk semantics implies equivalent evaluation steps in the debugger semantics

Proof sketch. We proceed by induction over the set of pending breakpoints B_p.

Base case: In this case the list B_p is empty. Either the actor is in running or in the paused state. By assumption, when the debugger is in a pause state, the commands C will un-pause the debugger.
In general a step in the base-level language can be done in two modes. In case the base level semantics performed a global reduction, there is a corresponding transition in the debugger by taking a step with CEL-STEP-GLOBAL. Similarly if it was a local rule, there is a possible transition with the CEL-STEP-LOCAL rule.

Inductive case: Assuming that there is list of pending breakpoints B_p leading to the actor configuration K'. When adding one breakpoint to that list we need to consider two cases. Either the breakpoint is applicable or it is not. When the breakpoint does not apply the corresponding NOT-APPLICABLE-BREAKPOINT rule will move the breakpoint to the list of checked breakpoints and the induction hypothesis applies. In the other case the breakpoint applies, in which by assumption the commands C will transition the debugger back to the run state at which point the induction hypothesis can be applied. ◀

Implication 2. An evaluation step in the debugger semantics implies an equivalent evaluation step in the AmbientTalk semantics

Proof sketch. By construction, the only two rules CEL-STEP-LOCAL and CEL-STEP-GLOBAL where the debuggers K field transitions to K' directly rely on the underlying AmbientTalk semantics. ◀

8 Related Work

To the best of our knowledge, multiverse debugging is the first debugging approach that allows developers to interactively browse all execution paths of parallel programs. In this section, we compare our work to other efforts on formalizing debuggers for actor languages and other programming paradigms. We also relate our work to static analysis techniques for debugging non-deterministic programs such as model checking and symbolic execution.

8.1 Formal specifications for debuggers

The first formal specification for debuggers was proposed by Da Silva [19]. He used a structural operational semantics that considers a debugger as a system, which transitions from one state to another using an evaluation history. He defines the semantics of his debugging approach on top of a deterministic relation specification of a programming language. To prove debugger correctness, Da Silva presented a proof of equivalence between two debugger approaches.

This work served as inspiration for multiverse debugging, but we focus on proving the equivalence between the base language and the debugger, i.e., their non-interference. While Da Silva does not address non-deterministic languages, he argues that non-repeatability of evaluation can be avoided by recording all choices where more than one evaluation rule could be chosen. However, to the best of our knowledge Da Silva never put this theory into practice. Our approach differs from Da Silva by embracing the non-deterministic nature of the base language and using it to derive a non-deterministic debugger.

Bernstein et al. [8] developed a debugging semantics based on transitions for a deterministic functional programming language. The evaluation steps in the debugging session correspond to executing subexpressions of the program. Similar to Voyager, developers can select terms (represented as nodes in the graph) corresponding to the program states and create new programs from them to debug. Bernstein et al. did not apply their techniques to non-deterministic languages.

In the context of distributed systems, Ferrari et al. [22] proposed a debugging calculus for mobile ambients. Similar to our approach, they model a debugger as an extension of the operational semantics of an underlying programming language. Their operational semantics is a causal model of behaviors which they represent using Petri nets. In a later work, Ferrari et al. [21] proposed Causal Nets which allows the developer to query a causal message graph generated by the execution of a set of distributed processes. We have experimented with converting the multiverse execution graph into a Petri net, but due to the size of the execution graphs the resulting Petri nets offered few additional insight into the program behavior.

In the context of algorithmic debugging, Luo et al. [45] proposed a formal model of tracing for functional programs. The authors proved correctness of evaluation dependency trees to identify faulty nodes, i.e. a node with erroneous computation. They consider correctness when the debugging algorithm detects a faulty node that matches the answer of the user. In contrast to multiverse debugging, this approach does not show an exploration of different non-deterministic paths, but the exploration of one path of execution of a functional program based on a trace. Similarly, Caballero et al. [13] uses a technique of algorithmic debugging to detect liveness issues in Erlang programs. Their approach can analyze sequential and concurrent programs using a calculus based on proofs to build execution trees.

Li et al. [40] introduced a formal semantics for debugging synchronous message-passing programs, e.g. MPI, Occam, and JCSP. They propose a structural operational semantics for a tracing procedure and bug/fix locating procedure. The goal of these procedures is to record useful information that helps to build the execution history of the program. More concretely, the tracing procedure records to one execution path in the evaluation of the program, ignoring non-determinism. In contrast, our approach considers all possible execution paths.

Giachino et al. [27] provide a causal consistent reversible semantics for the μOz language, featuring thread-based concurrency and asynchronous communication over ports. These semantics however, do not explore different paths of the execution of a concurrent program. Following the idea of reversible semantics, Lanese et al. [37] proposed a causal consistent reversible debugger for Erlang processes. More concretely, they use a reversible semantics for Erlang [52], in which they record a history of all the computed expressions, corresponding to each execution step. In contrast, our semantics only keep track of the state of actors and breakpoint information. In addition, the rules related to the reversible semantics are said to be non-deterministic, but no concrete exploration examples of different execution paths are included in the paper.

In the context of Petri nets, Van Mierlo et al. [63] proposed a debugging tool for observing erroneous states of non-deterministic behavior. The tool takes a model of a system as input, and builds a *Petri net reachability graph* which can be debugged in an interactive way. Similar to our approach, they provide online debugging operations, e.g., breakpoints and stepping, to explore specific program states. Multiverse debugging however, takes as input programs based on the operational semantics of the programming language and allows to debug the *execution graph* of the program.

8.2 Static Analysis Techniques

Since multiverse debugging allows developers to explore all possible paths of execution of an application, it can be considered closely related to static analysis techniques such as model checking and symbolic execution. Below, we provide an overview of such techniques, with a focus on actor-based approaches, and compare them to multiverse debugging.

Model checking. Model checking is a static technique for automatically verifying correctness properties of programs given a specification of the property. It has been studied for thread-based concurrency models to verify safety requirements such as the absence of deadlocks [31, 51, 36]. Ignoring recursion or relying on finite state models avoids undecidability problems due to synchronization as well as applying bounded analysis using testing or bounded model checking techniques [18].

Model checking has also been explored in the context of actor-based languages to verify properties like boundedness of actor mailboxes, and incorrect interleavings of messages. In the context of Erlang programs, Fredlund et al. [24] proposed a model checker that verifies boundedness of their mailboxes and process spawning. Other approaches have focused on verifying the property of mutual exclusion in Erlang programs [33, 20] or to analyze message interleavings between Erlang processes [16]. There exist model checkers for other actor-based languages such as Basset [39], a model checker that can analyze message schedules in actor-based programs written in Scala and ActorFoundry library for Java. Tasharofi et al. proposed an algorithm based on partial order reduction to prune the number of message interleavings in Scala and ActorFoundry programs [57]. Additionally, a more theoretic approach uses model checking to verify actors behavior based on compositional analysis of schedules [35].

Model checking tools excel at finding a set of bugs of which the programmer knows exactly how to describe them. Multiverse debugging is meant for debugging and interactively exploring the state space in order to discover bugs for which the programmer may not have a good description. Similar to model checking, multiverse debugging can suffer from the state explosion problem. As mentioned before, our approach does not blindly explore all the possible states but lets the developer decide which states to explore next, either explicitly or by using multiverse breakpoints, which makes multiverse debugging similar to bounded model checking [9]. Other techniques in model checking have been proposed to handle the state explosion problem including symbolic model checking with binary decision diagrams, partial order reductions and counter example guided abstraction refinement [17].

Symbolic execution. Symbolic execution is a static technique to test whether certain predefined properties can be violated by a program [5]. A key idea in symbolic execution is to explore programs taking as input *symbolic* values rather than concrete ones. According to [42], most symbolic execution techniques can be categorized in either techniques that create and search in a *subset* of the concrete search space (i.e. an under-approximation), or techniques that create and search in a *superset* of it (i.e. an over-approximation). Under-approximation techniques lead to false negatives (i.e. missing real errors) but are preferred over over-approximation techniques because the latter ones introduce false positives (report errors that do not exist) and do not scale as well because of the cost of handling infeasible states. Many research efforts on symbolic execution for multi-threaded programs have focused on improving the efficiency of over-approximation techniques [7, 43].

Concolic execution is a mix of concrete and symbolic execution which has been throughly studied for thread-based programs [5, 3, 28, 55]. In the context of actors, much work has also focused on concolic execution. Sen et al. [54] proposed a testing algorithm based on concolic execution together with runtime partial order reduction for detecting deadlock states in a language related to the actor semantics. Albert et al. [1] developed a test case generation framework which avoids redundant computations when exploring the order of several tasks. More recently, Albert et al. [2] proposed a variant of a dynamic partial order reduction algorithm which can be used when searching for deadlocks. Their algorithm aims to reduce state space exploration by distinguishing between two sources of non-determinism: actor selection and task selection. Recently, Li et al. proposed an exploration of the state space using symbolic execution based on heuristics that consider paths where only interact with a small number of actors [41].

Like multiverse debugging, symbolic execution can explore all possible execution paths of a program. While the use of abstract states alleviates the state explosion problem, that may imply missing execution paths (i.e. universes) containing a bug due to under approximation. In contrast to symbolic execution, multiverse debugging models the program execution only with concrete values, and can not miss executions paths. While multiverse debugging does not solve the state explosion problem, developers can pause and resume the program, and select themselves the different execution paths to explore. In the context of thread-based programs, some work in symbolic execution has studied solutions for reducing the complexity of path exploration based on merging paths [14], state pruning [30], probabilistic computations [44] and search heuristics [41].

9 Conclusion

We proposed multiverse debugging as a new debugging approach to tackle the problem of non-determinism in concurrent and parallel programs. Contrary to traditional concurrent debugging approaches, multiverse debugging allows developers to explore non-deterministic execution paths corresponding to the evaluation of a program. This is meant to simplify the reproduction and inspection of concurrency bugs, because it removes chance and probability from the equation of hitting the problematic interleaving. Instead, an execution path that can lead to a bug can be explored interactively and a developer can see the state in all possible universes.

To build a multiverse debugger, we provided a recipe with two steps. First, we need to define the operational semantics of a non-deterministic base language. Second, we need to define a debugger configuration and its operational semantics in terms of the base language semantics. In this paper, we have applied this recipe to provide a proof-of-concept multiverse debugger for actor-based programs called *Voyager*. Voyager uses as input a PLT-Redex program implemented in the AmbientTalk operational semantics and gives as output the reduction graph corresponding to all possible universes of the program. To make this exploration manageable, the graph can be explored interactively as one would do in a classic breakpoint-based debugger. Besides providing the semantics of a multiverse debugger, we also demonstrate that there is no interference between the debugger and the target program by proving non-interference. This shows that the debugger is probe-effect free.

We consider multiverse debugging to be a good basis for further experiments in debugging non-deterministic concurrent programs. Voyager's debugging operations merely scratch the surface of a new branch in debugging tools. The main open research question is how to make multiverse debugging practical for complex concurrent applications. While we believe that the interactive nature alleviates some of the scalability issues of static analyses, exploring the multiverses of larger programs can become unwieldy. Thus, research is needed to guide the exploration of the state graph, e.g. novel stepping semantics that work at the level of universes. Furthermore, the technique needs to be applied to a concrete language implementation beyond a PLT-Redex-based formalism. Efficient runtime techniques will be cornerstone to make it practical, but we may be able to leverage work of static analyses and back-in-time debugging [6].

References

1 Elvira Albert, Puri Arenas, and Miguel Gómez-Zamalloa. Test Case Generation of Actor Systems. In Bernd Finkbeiner, Geguang Pu, and Lijun Zhang, editors, *ATVA*, volume 9364 of *Lecture Notes in Computer Science*, pages 259–275. Springer, 2015. doi:10.1007/978-3-319-24953-7_21.

2 Elvira Albert, Puri Arenas, and Miguel Gómez-Zamalloa. Systematic testing of actor systems. *Softw. Test., Verif. Reliab.*, 28(3), 2018. doi:10.1002/stvr.1661.

3 Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley. Enhancing symbolic execution with veritesting. In Pankaj Jalote, Lionel C. Briand, and André van der Hoek, editors, *ICSE*, pages 1083–1094. ACM, 2014. doi:10.1145/2568225.2568293.

4 Thibaut Balabonski, Franc$_c$ois Pottier, and Jonathan Protzenko. Type Soundness and Race Freedom for Mezzo. In Michael Codish and Eijiro Sumii, editors, *Functional and Logic Programming*, pages 253–269, Cham, 2014. Springer International Publishing. doi:10.1007/978-3-319-07151-0_16.

5 Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. A Survey of Symbolic Execution Techniques. *ACM Comput. Surv.*, 51(3):50:1–50:39, May 2018. doi:10.1145/3182657.

6 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. ACM, 2016. doi:10.1145/2950290.2983933.

7 Tom Bergan, Dan Grossman, and Luis Ceze. Symbolic execution of multithreaded programs from arbitrary program contexts. In Andrew P. Black and Todd D. Millstein, editors, *OOPSLA*, pages 491–506. ACM, 2014. doi:10.1145/2660193.2660200.

8 Karen L. Bernstein and Eugene W. Stark. Operational Semantics of a Focusing Debugger. *Electronic Notes in Theoretical Computer Science*, 1:13–31, 1995. MFPS XI, Mathematical Foundations of Programming Semantics, Eleventh Annual Conference. doi:10.1016/S1571-0661(04)80002-1.

9 Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic Model Checking Without BDDs. In *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '99, pages 193–207, Berlin, Heidelberg, 1999. Springer-Verlag. doi:10.1007/3-540-49059-0_14.

10 Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership Types for Safe Programming: Preventing Data Races and Deadlocks. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '02, pages 211–230, New York, NY, USA, 2002. ACM. doi:10.1145/582419.582440.

11 Gilad Bracha, Peter von der Ahé, Vassili Bykov, Yaron Kashai, William Maddox, and Eliot Miranda. Modules as Objects in Newspeak. In Theo D'Hondt, editor, *ECOOP 2010 –*

Object-Oriented Programming, volume 6183 of *LNCS*, pages 405–428. Springer, 2010. doi:10.1007/978-3-642-14107-2_20.

12 Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic Model Checking: 10̂20 States and Beyond. In *LICS*, pages 428–439. IEEE Computer Society, 1990. doi:10.1016/0890-5401(92)90017-A.

13 Rafael Caballero, Enrique Martin-Martin, Adrián Riesco, and Salvador Tamarit. Declarative debugging of concurrent Erlang programs. *Journal of Logical and Algebraic Methods in Programming*, 101:22–41, 2018. doi:10.1016/j.jlamp.2018.07.005.

14 Cristian Cadar and Koushik Sen. Symbolic Execution for Software Testing: Three Decades Later. *Commun. ACM*, 56(2):82–90, February 2013. doi:10.1145/2408776.2408795.

15 M. Christakis, A. Gotovos, and K. Sagonas. Systematic Testing for Detecting Concurrency Errors in Erlang Programs. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 154–163, March 2013.

16 Maria Christakis, Alkis Gotovos, and Konstantinos F. Sagonas. Systematic Testing for Detecting Concurrency Errors in Erlang Programs. In *ICST*, pages 154–163. IEEE Computer Society, 2013.

17 Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Progress on the State Explosion Problem in Model Checking. In Reinhard Wilhelm, editor, *Informatics*, volume 2000 of *Lecture Notes in Computer Science*, pages 176–194. Springer, 2001. doi:10.1007/3-540-44577-3_12.

18 Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors. *Handbook of Model Checking*. Springer, 2018. doi:10.1007/978-3-319-10575-8.

19 Fabio Q. B. da Silva. *Correctness proofs of compilers and debuggers: an approach based on structural operational semantics*. PhD thesis, University of Edinburgh, UK, 1992. British Library, EThOS. URL: http://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.649061.

20 Emanuele D'Osualdo, Jonathan Kochems, and C. H. Luke Ong. Automatic Verification of Erlang-Style Concurrency. In Francesco Logozzo and Manuel Fähndrich, editors, *20th International Symposium on Static Analysis*, SAS 2013, pages 454–476. Springer, June 2013. doi:10.1007/978-3-642-38856-9_24.

21 Gian Luigi Ferrari, Roberto Guanciale, Daniele Strollo, and Emilio Tuosto. Debugging Distributed Systems with Causal Nets. *ECEASST*, 14:1–10, 2008. doi:10.14279/tuj.eceasst.14.190.181.

22 GianLuigi Ferrari and Emilio Tuosto. A Debugging Calculus for Mobile Ambients. In *Proceedings of the 2001 ACM Symposium on Applied Computing*, SAC '01, page 2, New York, NY, USA, 2001. ACM. doi:10.1145/372202.380701.

23 Cormac Flanagan and Stephen N. Freund. Type-based Race Detection for Java. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 219–232, New York, NY, USA, 2000. ACM. doi:10.1145/349299.349328.

24 Lars-Ake Fredlund, Dilian Gurov, Thomas Noll, Mads Dam, Thomas Arts, and Gennady Chugunov. A verification tool for ERLANG. *STTT*, 4(4):405–420, 2003. doi:10.1007/s100090100071.

25 Jason Gait. A probe effect in concurrent programs. *Software: Practice and Experience*, 16(3):225–233, 1986. doi:10.1002/spe.4380160304.

26 Elena Giachino, Carlo A. Grazia, Cosimo Laneve, Michael Lienhardt, and Peter Y. H. Wong. Deadlock Analysis of Concurrent Objects: Theory and Practice. In Einar Broch Johnsen and Luigia Petre, editors, *Integrated Formal Methods*, pages 394–411, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. doi:10.1007/978-3-642-38613-8_27.

27 Elena Giachino, Ivan Lanese, and Claudio Antares Mezzina. Causal-Consistent Reversible Debugging. In Stefania Gnesi and Arend Rensink, editors, *FASE*, volume 8411 of *Lecture Notes in Computer Science*, pages 370–384. Springer, 2014. doi:10.1007/978-3-642-54804-8_26.

28 Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed Automated Random Testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language*

Design and Implementation, PLDI '05, pages 213–223, New York, NY, USA, 2005. ACM. doi:10.1145/1065010.1065036.

29 Elisa Gonzalez Boix, Carlos Noguera, and Wolfgang De Meuter. Distributed Debugging for Mobile Networks . *Journal of Systems and Software*, 90:76–90, 2014. doi:10.1016/j.jss.2013.11.1099.

30 Shengjian Guo, Markus Kusano, and Chao Wang. Conc-iSE: Incremental Symbolic Execution of Concurrent Software. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, pages 531–542, New York, NY, USA, 2016. ACM. doi:10.1145/2970276.2970332.

31 K. Havelund and T. Pressburger. Model Checking Java Programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4), 1998. doi:10.1007/s100090050043.

32 Carl Hewitt, Peter Bishop, and Richard Steiger. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *IJCAI'73: Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, pages 235–245. Morgan Kaufmann, 1973.

33 Frank Huch. Verification of Erlang Programs Using Abstract Interpretation and Model Checking. In *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming*, ICFP '99, pages 261–272, New York, NY, USA, 1999. ACM. doi:10.1145/317636.317908.

34 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.

35 Mohammad Mahdi Jaghoori, Frank S. de Boer, Delphine Longuet, Tom Chothia, and Marjan Sirjani. Compositional schedulability analysis of real-time actor-based systems. *Acta Inf.*, 54(4):343–378, 2017. doi:10.1007/s00236-015-0254-x.

36 Michalis Kokologiannakis, Ori Lahav, Konstantinos Sagonas, and Viktor Vafeiadis. Effective stateless model checking for C/C++ concurrency. *PACMPL*, 2(POPL):17:1–17:32, 2018. doi:10.1145/3158105.

37 Ivan Lanese, Naoki Nishida, Adrián Palacios, and Germán Vidal. CauDEr: A Causal-Consistent Reversible Debugger for Erlang. In John P. Gallagher and Martin Sulzmann, editors, *Functional and Logic Programming*, volume 10818 of *FLOPS'18*, pages 247–263, Cham, 2018. Springer. doi:10.1007/978-3-319-90686-7_16.

38 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. IEEE Computer Society. doi:10.1109/ASE.2009.88.

39 Steven Lauterburg, Rajesh K. Karmani, Darko Marinov, and Gul Agha. Basset: A Tool for Systematic Testing of Actor Programs. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 363–364, New York, NY, USA, 2010. ACM. doi:10.1145/1882291.1882349.

40 He Li, Jie Luo, and Wei Li. A formal semantics for debugging synchronous message passing-based concurrent programs. *Science China Information Sciences*, 57(12):1–18, December 2014. doi:10.1007/s11432-014-5150-4.

41 Sihan Li, Farah Hariri, and Gul Agha. Targeted Test Generation for Actor Systems. In Todd D. Millstein, editor, *ECOOP*, volume 109 of *LIPIcs*, pages 8:1–8:31. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018. doi:10.4230/LIPIcs.ECOOP.2018.8.

42 Yude Lin. *Symbolic execution with over-approximation.* PhD thesis, The University of Melbourne, 2017.

43 Yude Lin, Tim Miller, and Harald Søndergaard. Compositional Symbolic Execution: Incremental Solving Revisited. In Alex Potanin, Gail C. Murphy, Steve Reeves, and Jens Dietrich, editors, *APSEC*, pages 273–280. IEEE Computer Society, 2016. doi:10.1109/ASWEC.2015.32.

44 Kasper Søe Luckow, Corina S. Pasareanu, Matthew B. Dwyer, Antonio Filieri, and Willem Visser. Exact and approximate probabilistic symbolic execution for nondeterministic programs. In Ivica Crnkovic, Marsha Chechik, and Paul Grünbacher, editors, *ASE*, pages 575–586. ACM, 2014. doi:10.1145/2642937.2643011.

45 Yong Luo and Olaf Chitil. Proving the correctness of algorithmic debugging for functional programs. In Henrik Nilsson, editor, *Trends in Functional Programming*, volume 7 of *Trends in Functional Programming*, pages 19–34. Intellect, 2006.

46 Stefan Marr, Carmen Torres Lopez, Dominik Aumayr, Elisa Gonzalez Boix, and Hanspeter Mössenböck. A Concurrency-Agnostic Protocol for Multi-Paradigm Concurrent Debugging Tools. In Davide Ancona, editor, *Proceedings of the 13th Symposium on Dynamic Languages*, pages 3–14. ACM, 2017. doi:10.1145/3133841.3133842.

47 John McCarthy. A Basis for a Mathematical Theory of Computation, Preliminary Report. In *Papers Presented at the May 9-11, 1961, Western Joint IRE-AIEE-ACM Computer Conference*, IRE-AIEE-ACM '61 (Western), pages 225–238, New York, NY, USA, 1961. ACM. doi:10.1145/1460690.1460715.

48 Charles E McDowell and David P Helmbold. Debugging concurrent programs. *ACM Computing Surveys (CSUR)*, 21(4):593–622, 1989. doi:10.1145/76894.76897.

49 Mark S Miller, E Dean Tribble, and Jonathan Shapiro. Concurrency among strangers. In *International Symposium on Trustworthy Global Computing*, pages 195–229. Springer, 2005. doi:10.1007/11580850_12.

50 Stijn Mostinckx, Tom Van Cutsem, Stijn Timbermont, Elisa Gonzalez Boix, Éric Tanter, and Wolfgang De Meuter. Mirror-based reflection in AmbientTalk. *Softw. Pract. Exper.*, 39(7):661–699, 2009. doi:10.1002/spe.v39:7.

51 Madanlal Musuvathi and Shaz Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In Jeanne Ferrante and Kathryn S. McKinley, editors, *PLDI*, pages 446–455. ACM, 2007. doi:10.1145/1250734.1250785.

52 Naoki Nishida, Adrián Palacios, and Germán Vidal. A Reversible Semantics for Erlang. In Manuel V. Hermenegildo and Pedro López-García, editors, *LOPSTR*, volume 10184 of *Lecture Notes in Computer Science*, pages 259–274. Springer, 2016. doi:10.1007/978-3-319-63139-4_15.

53 Michael Perscheid, Benjamin Siegmund, Marcel Taeumel, and Robert Hirschfeld. Studying the advancement in debugging practice of professional software developers. *Software Quality Journal*, 25(1):83–110, 2016. doi:10.1007/s11219-015-9294-2.

54 Koushik Sen and Gul Agha. Automated Systematic Testing of Open Distributed Programs. In Luciano Baresi and Reiko Heckel, editors, *FASE*, volume 3922 of *Lecture Notes in Computer Science*, pages 339–356. Springer, 2006. doi:10.1007/11693017_25.

55 Koushik Sen and Gul Agha. CUTE and jCUTE: Concolic Unit Testing and Explicit Path Model-Checking Tools. In Thomas Ball and Robert B. Jones, editors, *Computer Aided Verification*, pages 419–423, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. doi:10.1007/11817963_38.

56 Kaku Takeuchi, Kohei Honda, and Makoto Kubo. An interaction-based language and its typing system. In Costas Halatsis, Dimitrios Maritsas, George Philokyprou, and Sergios Theodoridis, editors, *PARLE'94 Parallel Architectures and Languages Europe*, pages 398–413, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg. doi:10.1007/3-540-58184-7_118.

57 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 Holger Giese and Grigore Rosu, editors, *Formal Techniques for Distributed Systems: Joint 14th IFIP WG 6.1 International Conference, FMOODS 2012 and 32nd IFIP WG 6.1 International Conference, FORTE 2012, Stockholm, Sweden, June 13-16, 2012. Proceedings*, pages 219–234. Springer, 2012. doi:10.1007/978-3-642-30793-5_14.

58 Stefan Tilkov and Steve Vinoski. Node.js: Using JavaScript to Build High-Performance Network Programs. *IEEE Internet Computing*, 14(6):80–83, November 2010. `doi:10.1109/MIC.2010.145`.

59 Carmen Torres Lopez, Elisa Gonzalez Boix, Christophe Scholliers, Stefan Marr, and Hanspeter Mössenböck. A Principled Approach Towards Debugging Communicating Event-loops. In *Proceedings of the 7th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control*, AGERE!'17, pages 41–49. ACM, October 2017. `doi:10.1145/3141834.3141839`.

60 Carmen Torres Lopez, Stefan Marr, Elisa Gonzalez Boix, and Hanspeter Mössenböck. *A Study of Concurrency Bugs and Advanced Development Support for Actor-based Programs*, chapter 6, pages 155–185. Springer International Publishing, Cham, 2018. `doi:10.1007/978-3-030-00302-9_6`.

61 Antti Valmari. *The state explosion problem*, chapter 9, pages 429–528. Springer Berlin Heidelberg, Berlin, Heidelberg, 1998. `doi:10.1007/3-540-65306-6_21`.

62 Tom Van Cutsem, Elisa Gonzalez Boix, Christophe Scholliers, Andoni Lombide Carreton, Dries Harnie, Kevin Pinte, and Wolfgang De Meuter. AmbientTalk: programming responsive mobile peer-to-peer applications with actors. *Computer Languages, Systems & Structures*, 40(3-4):112–136, 2014. `doi:10.1016/j.cl.2014.05.002`.

63 Simon Van Mierlo and Hans Vangheluwe. Debugging non-determinism: a petrinets modelling, analysis, and debugging tool. In *CEUR workshop proceedings*, volume 2019, pages 460–462, 2017.

A Reduction Rules of the Operational Semantics of Voyager

In this appendix we give an overview of all the reduction rules of the Voyager debugger to debug the example application. We split the reduction rules into five groups:

1. Reduction rules for modeling the connection of the debugger with the base level language (cf. Section 6.3.1)
2. Reduction rules for breakpoints (cf. Section 6.3.2), including rules needed to model breakpoints which require trigger breakpoints for their functioning.
3. Bookkeeping reduction rules (cf. Section 6.3.3), i.e. rules that are related to the actor state when breakpoints are not applicable and when new actors are created.
4. Reduction rules for the stepping operations (cf. Section 6.3.4), consists of the rules for stepping commands that can be applied on the level of messages, futures, and turns.
5. Reduction rules for other debugging commands (cf. Section 6.3.5), i.e. rules that will resume and pause the program's execution.

$$\textsc{(cel-step-global)}\quad \frac{K \rightarrow_k K' \qquad \mathit{not-applicable-add-new-actor}}{\mathcal{D}\langle (), B_c, \mathrm{run}, C, A_s, K\rangle \rightarrow_d \mathcal{D}\langle B_c, (), \mathrm{run}, C, A_s, K'\rangle}$$

$$\textsc{(cel-step-local)}\quad \frac{K = K'\dot{\cup}\{a\} \qquad a \overset{*}{\rightarrow}_a a' \qquad A'_s = \mathit{update}(A_s, a) \qquad \mathit{not-applicable-add-new-actor}}{\mathcal{D}\langle (), B_c, \mathrm{run}, C, A_s, K\rangle \rightarrow_d \mathcal{D}\langle B_c, (), \mathrm{run}, C, A'_s, K'\dot{\cup}a'\rangle}$$

Figure 13 Reduction rules for connecting the debugger with the base language.

(TRIGGER-MSB)

$$\frac{\mathcal{A}\langle \iota_a, O, Q_{in}, e_{\square}[\iota_{a'}.\iota_o \leftarrow_{\iota_i} m(\overline{v})]\rangle \in K \qquad A'_s = A_s + \{\mathcal{CS}\langle \iota_a, pause\rangle\}}{\mathcal{D}\langle \mathcal{B}\langle msb, \iota_i\rangle \cdot B_p, B_c, \text{run}, C, A_s, K\rangle \rightarrow_d \mathcal{D}\langle B_p, B_c, \text{pause}, C, A'_s, K\rangle}$$

(SAVE-MRB)

$$\frac{\mathcal{A}\langle \iota_a, O, Q_{in}, e_{\square}[\iota_{a'}.\iota_o \leftarrow_{\iota_i} m(\overline{v})]\rangle \in K}{\mathcal{D}\langle \mathcal{B}\langle mrb, \iota_i\rangle \cdot B_p, B_c, \text{run}, C, A_s, K\rangle \rightarrow_d \mathcal{D}\langle B_p, B_c \cdot \mathcal{B}\langle mrb - trigger, \iota_{a'}, \iota_i\rangle, \text{run}, C, A_s, K\rangle}$$

(TRIGGER-MRB)

$$\frac{\mathcal{A}\langle \iota_a, O, m \cdot Q_{in}, v\rangle \in K \qquad A'_s = A_s + \{\mathcal{CS}\langle \iota_a, pause\rangle\}}{\mathcal{D}\langle \mathcal{B}\langle mrb - trigger, \iota_a, \iota_i\rangle \cdot B_p, B_c, run, C, A_s, K\rangle \rightarrow_d \mathcal{D}\langle B_p, B_c, \text{pause}, C, A'_s, K\rangle}$$

Figure 14 Reduction rules for breakpoints.

(ADD-NEW-ACTOR)

$$\frac{\mathcal{A}\langle \iota_a, O, Q_{in}, e_{\square}[\text{actor}\{\overline{f := e}, \overline{m(\overline{x})\{e\}}\}]\rangle \in K \qquad \mathcal{CS}\langle \iota_{new}, a_s\rangle \notin A_s}{\mathcal{D}\langle B_p, B_c, \text{run}, C, A_s, K\rangle \rightarrow_d \mathcal{D}\langle B_p, B_c, \text{run}, C, A_s \cdot \mathcal{CS}\langle \iota_{new}, run\rangle, K\rangle}$$

(NOT-APPLICABLE-BREAKPOINT[TRIGGER-MSB,SAVE-MRB,TRIGGER-MRB])

$$\frac{not - applicable - breakpoint}{\mathcal{D}\langle \mathcal{B}\langle t_{ub}, \iota_i\rangle \cdot B_p, B_c, \text{run}, C, A_s, K\rangle \rightarrow_d \mathcal{D}\langle B_p, B_c \cdot \mathcal{B}\langle t_{ub}, \iota_i\rangle, \text{run}, C, A_s, K\rangle}$$

Figure 15 Reduction rules for bookkeeping information about the program state needed for breakpoints and stepping operations.

(PREPARE-STEP-NEXT-TURN)

$$\frac{\mathcal{A}\langle \iota_a, O, m \cdot Q_{in}, e\rangle \in K \qquad A'_s = A_s \dot{\cup} \{\mathcal{CS}\langle \iota_a, (step\ 1)\rangle\}}{\mathcal{D}\langle B_p, B_c, \text{pause}, (StepNextTurn\ \iota_a) \cdot C, A_s \dot{\cup} \mathcal{CS}\langle \iota_a, (pause)\rangle, K\rangle \rightarrow_d \mathcal{D}\langle B_p, B_c, \text{run}, (StepNextTurn\ \iota_a) \cdot C, A'_s, K\rangle}$$

(TRIGGER-STEP-NEXT-TURN)

$$\frac{\mathcal{A}\langle \iota_a, O, m \cdot Q_{in}, v\rangle \in K \qquad A'_s = A_s \dot{\cup} \{\mathcal{CS}\langle \iota_a, pause\rangle\}}{\mathcal{D}\langle B_p, B_c, \text{run}, (StepNextTurn\ \iota_a) \cdot C, A_s \dot{\cup} \mathcal{CS}\langle \iota_a, (step\ 0)\rangle, K\rangle \rightarrow_d \mathcal{D}\langle B_p, B_c, \text{pause}, C, A'_s, K\rangle}$$

Figure 16 Reduction rules for stepping operations.

(RESUME-EXECUTION)

$$\frac{A'_s = run(A_s)}{\mathcal{D}\langle B_p, B_c, \text{pause}, Resume \cdot C, A_s, K\rangle \rightarrow_d \mathcal{D}\langle B_p, B_c, \text{run}, C, A'_s, K\rangle}$$

(PAUSE-EXECUTION)

$$\frac{A'_s = pause(A_s)}{\mathcal{D}\langle B_p, B_c, \text{run}, Pause \cdot C, A_s, K\rangle \rightarrow_d \mathcal{D}\langle B_p, B_c, \text{pause}, C, A'_s, K\rangle}$$

Figure 17 Reduction rules for basic debugging commands.

Motion Session Types for Robotic Interactions

Rupak Majumdar
MPI-SWS, Saarbrücken, Germany
rupak@mpi-sws.org

Marcus Pirron
MPI-SWS, Saarbrücken, Germany
mpirron@mpi-sws.org

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

Damien Zufferey
MPI-SWS, Saarbrücken, Germany
zufferey@mpi-sws.org

Abstract

Robotics applications involve programming concurrent components synchronising through messages while simultaneously executing *motion primitives* that control the state of the physical world. Today, these applications are typically programmed in low-level imperative programming languages which provide little support for abstraction or reasoning.

We present a unifying programming model for concurrent message-passing systems that additionally control the evolution of physical state variables, together with a compositional reasoning framework based on multiparty session types. Our programming model combines *message-passing concurrent processes* with *motion primitives*. Processes represent autonomous components in a robotic assembly, such as a cart or a robotic arm, and they synchronise via discrete messages as well as via motion primitives. Continuous evolution of trajectories under the action of controllers is also modelled by motion primitives, which operate in global, physical time.

We use multiparty session types as specifications to orchestrate discrete message-passing concurrency and continuous flow of trajectories. A global session type specifies the communication protocol among the components with joint motion primitives. A projection from a global type ensures that jointly executed actions at end-points are *communication safe* and *deadlock-free*, i.e., session-typed components do not get stuck. Together, these checks provide a compositional verification methodology for assemblies of robotic components with respect to concurrency invariants such as a progress property of communications as well as dynamic invariants such as absence of collision.

We have implemented our core language and, through initial experiments, have shown how multiparty session types can be used to specify and compositionally verify robotic systems implemented on top of off-the-shelf and custom hardware using standard robotics application libraries.

2012 ACM Subject Classification Computer systems organization → Robotics; Software and its engineering → Concurrent programming languages; Theory of computation → Process calculi; Theory of computation → Type theory

Keywords and phrases Session Types, Robotics, Concurrent Programming, Motions, Communications, Multiparty Session Types, Deadlock Freedom

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2019.28

Category Brave New Idea Paper

Funding *Rupak Majumdar*: DFG 389792660 TRR 248, ERC Synergy Grant 610150.
Marcus Pirron: DFG 389792660 TRR 248, ERC Synergy Grant 610150.
Nobuko Yoshida: EPSRC EP/K034413/1, EP/K011715/1, EP/L00058X/1, EP/N027833/1 and EP/N028201/1.
Damien Zufferey: DFG 389792660 TRR 248, ERC Synergy Grant 610150.

33rd European Conference on Object-Oriented Programming (ECOOP 2019).
Editor: Alastair F. Donaldson; Article No. 28; pp. 28:1–28:27
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Many cyber-physical systems today involve an interaction among communication-centric components which together control trajectories of physical variables. For example, consider an autonomous robotic system executing in an assembly line. The components in such an example would be robotic manipulators or arms as well as robotic carts onto which one or more arms may be mounted. A global task may involve communication between the carts and the arms – for example, to jointly decide the position of the arms and to jointly plan trajectories – as well as the execution of motion primitives – for example, to follow a trajectory or to grip an object. Today, a programmer developing such an application must manually orchestrate the messaging and the dynamics: errors in either can lead to potentially catastrophic system failures. Typically, programs are written in (untyped) imperative programming language using messaging libraries. Arguments about correctness are informal at best, with no support from the language.

In this paper, we take the first steps towards a uniform programming model for autonomous robotic systems. Our model combines message-based communication with physical dynamics ("motion primitives") over time. Our starting point is the notion of *multiparty session types* [25, 26, 10], a principled, type-based, discipline to specify and reason about *global communication protocols* in a concurrent system. We enrich a process-based core language for communication with the ability to execute *dynamic motion primitives* over time. Motion primitives encapsulate the actions of dynamic controllers on the physical world and define the continuous evolution of the trajectories of the system. At the same time, we enrich a type system for multiparty session-based communication with motion primitives.

The interaction of communication and dynamics is non-trivial. Since time is global to a physical system, every independently running process must be ready to execute their motion primitives simultaneously. Thus, for example, programs in which one component is blocked waiting for a message while another moves along a trajectory must be ruled out as ill-typed. To keep the complexity of the problem manageable, our semantics keeps, as much as possible, the message exchanges separate from the continuous trajectories. In particular, in our model, message exchanges occur instantaneously and at discrete time steps, à la synchronous reactive programming, while motion primitives execute in global time. System evolution is then organised into rounds; each round consists of a logical time for communication followed by physical time for motion. This assumption is realistic for systems where the speed of the trajectories is comparatively slow compared to the message transmission delay.

Our reasoning principles closely follow the usual type-checking approach of multiparty session types. Specifications are described through *global types*, which constrain both message sequences and motion sequences. Global types are projected to *local types*, which specify the actions in a session from the perspective of a single end-point process. Finally, a verification step checks that each process satisfies its local type. The soundness theorem ensures that in this last case, the composition of the processes satisfy a protocol compliance.

Our type system ensures communication safety and deadlock-freedom for messages, ensuring, for example, that communication is not stuck or time cannot progress. In addition, we verify safety properties of physical trajectories such as non-collision by constraint-based verification of simultaneously executed motion primitives specified in the global type.

Existing session type formalisms such as [9] fall short to model a combination of individual interactions and global synchronisations by motions. To demonstrate our initial step and to observe an effect of new primitives specific to robotics interactions, we start from the simplest multiparty session type system in [15, 18]. The programming model and type system introduced in this paper provides the foundations for PGCD programs, a practical programming system to develop concurrent robotics applications [4]. We have used our calculus and type system to verify correctness properties of (abstract versions of) multi-robot co-ordination programs written in PGCD, which then execute on real robotics hardware. Our evaluation shows that multiparty session types and choreographies for multi-robot co-ordination and manipulation can lead to statically verified implementations that run on off-the-shelf and custom robotics hardware platforms.

Outline. We first give a gentle introduction to motion session types to those who are interested in concurrent robotics programming, but not familiar with session types. Section 3 discusses a core abstract calculus of processes where motions are abstracted by just the passage of time; Section 4 defines a typing system with motion primitives; Section 5 extends our theory to deal with continuous trajectories; Section 6 discusses our implementation; Section 7 gives related work and Section 8 concludes.

2 A Gentle Introduction to Motion Session Types

The aim of this section is to give a gentle introduction of motion session types for readers who are interested in robotics programming but who are not familiar with session types nor process calculi.

A key difficulty in robotics programming is that the programmer has to reason about concurrent processes communicating through messages as well as about dynamics evolving in time. The idea of motion session types is to provide a typing framework to only allow programs that follow structured sequences of interactions and motion. A *session* will be a natural unit of structured communication and motion. *Motion session types* abstract the structure of a session. and provide a syntax-driven approach to restricting programs to a well-behaved subclass – for this subclass, one can check processes compositionally and derive properties of the composition.

Motion session types extend session types, introduced in a series of papers during the 1990s [23, 43, 24], in the context of pure concurrent programming. Session types have since been studied in many contexts over the last decade – see the surveys of the field [27, 17].

We begin by an overview of the key technical ideas of multiparty session types. Then we introduce motion primitives to multiparty session types for specifying actions over time. Finally, we refine the motion primitives to physical motion executed by the robots.

2.1 Communication: Multiparty Session Types

We begin with a review of multiparty session types, a methodology to enable compositional reasoning about communication.

As a simple example, consider a scenario in which a cart and arm assembly has to fetch objects. We associate a process with each physical component; thus, we model the scenario using a *cart* (Cart) and an *arm* (Arm) attached to the cart. The task involves synchronisation between the cart and the arm as well as co-ordinated motion. Synchronization is obtained through the exchange of messages. We defer the discussion on motion to Section 2.2.

Specifically, the protocol works as follows.

1. The cart sends the arm a fold command *fold*. On receiving the command, the arm folds itself. When the arm is completely folded, it sends back a message *ok* to the cart. On receipt of this message, the cart moves.

2. When the cart reaches the object, it stops and sends a *grab* message to the arm to grab the object. While the cart waits, the arm executes the grabbing operation, followed by a folding operation. Then the arm sends a message *ok* to the cart. This sequence may need to be repeated.
3. When all tasks are finished, the cart sends a message *done* to the arm, and the protocol terminates.

The multiparty session types methodology is as follows. First, define a *global type* that gives a shared contract of the allowed pattern of message exchanges in the system. Second, *project* the global type to each end-point participant to get a *local type*: an obligation on the message sends and receipts for each process that together ensure that the pattern of messages are allowed by the global type. Finally, check that the implementation of each process conforms to its local type.

In our protocol, from a global perspective, we expect to see the following pattern of message exchanges, encoded as a *global type* for the communication:

$$\mu\mathbf{t}.\mathsf{Cart} \to \mathsf{Arm}\colon \{\mathit{fold}.\mathsf{Arm} \to \mathsf{Cart}\colon \mathit{ok}.\mathsf{Cart} \to \mathsf{Arm}\colon \mathit{grab}.\mathsf{Arm} \to \mathsf{Cart}\colon \mathit{ok}.\mathbf{t},\ \mathit{done}.\mathtt{end}\} \quad (1)$$

The type describes the global pattern of communication between Cart and Arm using message exchanges, sequencing, choice, and repetition. The basic pattern $\mathsf{Cart} \to \mathsf{Arm}\colon m$ indicates a message m sent from the Cart to the Arm. The communication starts with the cart sending either a *fold* or a *done* command to the arm. In case of *done*, the protocol ends (type `end`); otherwise, the communication continues with the sequence *ok. grab. ok* followed by a repetition of the entire pattern. The operator "." denotes sequencing, and the type $\mu\mathbf{t}.T$ denotes recursion of T.

The global type states what are the valid message sequences allowed in the system. When we implement Cart and Arm separately, we would like to check that their composition conforms to the global type. We can perform this check compositionally as follows.

Since there are only two participants, projecting to each participant is simple. From the perspective of the Cart, the communication can be described by the type:

$$\mu\mathbf{t}.\,((\ !\mathit{fold}.\ ?\mathit{ok}.\ !\mathit{grab}.\ ?\mathit{ok}.\mathbf{t}\ ) \quad \oplus \quad (\ !\mathit{done}.\ \mathtt{end}\ )) \qquad (2)$$

where $!m$ denotes a message m sent (to the Arm) and $?m$ denotes a message m received from the Arm. and $\oplus$ denotes an (internal) choice. Thus, the type states that Cart repeats actions $!\mathit{fold}.\ ?\mathit{ok}.\ !\mathit{grab}.\ ?\mathit{ok}$ until at some point it sends *done* and exits.

Dually, from the viewpoint of the Arm, the same global session is described by the dual type

$$\mu\mathbf{t}.\,((\ \ ?\mathit{fold}.\ !\mathit{ok}.\ ?\mathit{grab}.\ !\mathit{ok}.\mathbf{t}\ ) \quad \& \quad (\ ?\mathit{done}.\ \mathtt{end})) \qquad (3)$$

in which & means that a choice is offered externally.

We can now individually check that the implementations of the cart and the arm conform to these local types.

The global type seems overkill if there are only two participants; indeed, the global type is uniquely determined given the local type (2) or its dual (3). However, for applications involving *multiple parties*, the global type and its projection to each participant are essential to provide a shared contract among all participants.

For example, consider a simple ring protocol, where the Arm process above is divided into two parts, Lower and Upper. Now, Cart sends a message *fold* to the lower arm Lower, which forwards the message to Upper. After receiving the message, Upper sends an acknowledgement *ok* to Cart. We start by specifying the global type as:

$$\mathsf{Cart} \to \mathsf{Lower}\colon \mathit{fold}.\mathsf{Lower} \to \mathsf{Upper}\colon \mathit{fold}.\mathsf{Upper} \to \mathsf{Cart}\colon \mathit{ok}.\mathtt{end} \tag{4}$$

As before, we want to check each process locally against a local type such that if each process conforms to its local type then the composition satisfies the global type.

The global type in (4) is *projected* into the three endpoint session types:

$$\begin{array}{ll} \mathsf{Cart}\text{'s endpoint type:} & \mathsf{Lower}!\mathit{fold}.\mathsf{Upper}?\mathit{ok}.\mathtt{end} \\ \mathsf{Lower}\text{'s endpoint type:} & \mathsf{Cart}?\mathit{fold}.\mathsf{Upper}!\mathit{fold}.\mathtt{end} \\ \mathsf{Upper}\text{'s endpoint type:} & \mathsf{Lower}?\mathit{fold}.\mathsf{Cart}!\mathit{ok}.\mathtt{end} \end{array}$$

where $\mathsf{Lower}!\mathit{fold}$ means “send to Lower a *fold* message,” and $\mathsf{Upper}?\mathit{ok}$ means “receive from Upper an *ok* message.” Then each process is type-checked against its own endpoint type. When the three processes are executed, their interactions automatically follow the stipulated scenario.

If instead of a global type, we only used three separate binary session types to describe the message exchanges between Cart and Lower, between Lower and Upper, and between Upper and Cart, respectively, without using a global type, then we lose essential sequencing information in this interaction scenario. Consequently, we can no longer guarantee deadlock-freedom among these three parties. Since the three separate binary sessions can be interleaved freely, an implementation of the Cart that conforms to $\mathsf{Upper}?\mathit{ok}.\mathsf{Lower}!\mathit{fold}.\mathtt{end}$ becomes typable. This causes the situation that each of the three parties blocks indefinitely while waiting for a message to be delivered. Thus, we shall use the power of multiparty session types to ensure correct communication patterns.

2.2 Motion: Motion Primitives and Trajectories

So far, we focused on the communication pattern and ignored the physical actions of the robots. Our framework of motion session types extends multiparty session types to also reason about *motion primitives*, which model change of state in the physical world effected by the robots. We add motion in two steps: first we treat motion primitives as abstract actions that have associated durations, and second as dynamic trajectories.

Abstractly, we model motion primitives as actions that take physical time. Accordingly, we extend session types with motion primitive $\mathsf{dt}\langle \mathsf{p}_i : a_i\rangle$, which indicates that the participants p_i jointly execute motion primitives a_i for the same duration of time.

Let us add the motion primitives to the cart and arm example. Recall that on receiving the command *fold*, the arm folds itself; meanwhile, the cart waits. When the arm is completely folded, it sends back a message to the cart, then the cart moves, following a trajectory to the object. This means the time the arm folds and the time the cart is idle (waiting for the arm) should be the same. Similarly, the time cart is moving and the idle time the arm waits for the cart should be synchronised. This explicit synchronisation is represented by the following global type:

$$\begin{array}{l} \mathsf{Cart} \to \mathsf{Arm} : \mathit{fold}.\mathsf{dt}\langle \mathsf{Cart} : \mathsf{idle}, \mathsf{Arm} : \mathsf{fold}\rangle. \\ \quad\quad \mathsf{Arm} \to \mathsf{Cart} : \mathit{ok}.\mathsf{dt}\langle \mathsf{Cart} : \mathsf{move}, \mathsf{Arm} : \mathsf{idle}\rangle.G \end{array}$$

where "$\mathsf{dt}\langle\mathsf{Cart} : \mathsf{idle}, \mathsf{Arm} : \mathsf{fold}\rangle$" specifies the joint motion primitives idle executed by the Cart and fold executed by the Arm are synchronised. We extend local types with motion primitives as well. The conformance check ensures that, if each process conforms to its local types, then the composition of the system conforms to the global type – which now includes both message-based synchronization as well as synchronization over time using motion primitives.

Finally, we expand the abstract motion primitives with the underlying dynamic controllers and ensure that the joint execution of motion primitives is possible in the system. This requires refining each motion primitive to its underlying dynamical system and checking that whenever the global type specifies a joint execution of motion primitives, there is in fact a joint trajectory of the system that can be executed.

3 Motion Session Calculus

We now introduce the syntax and semantics of a synchronous multiparty motion session calculus. Our starting point is to associate a process with the physical component it controls. This can be either a "complete" robot or parts of a robot (like the cart or arm in the previous section). This makes it possible to model modular robots where parts may be swapped for different tasks. In the following, we simply say "robot" to describe a physical component (which may be a complete robot or part of a larger robot). Our programming model will associate a process with each such robot.

We build our motion session calculus based on a session calculus studied in [15, 18], which simplifies the synchronous multiparty session calculus in [29] by eliminating both shared channels for session initiations and session channels for communications inside sessions.

▶ **Notation 3.1** (Base sets). *We use the following base sets:* values*, ranged over by* $\mathsf{v}, \mathsf{v}', \ldots$*;* expressions*, ranged over by* $\mathsf{e}, \mathsf{e}', \ldots$*;* expression variables*, ranged over by* $x, y, z \ldots$*;* labels*, ranged over by* $\ell, \ell', \ldots$*;* session participants*, ranged over by* $\mathsf{p}, \mathsf{q}, \ldots$*;* motion primitives*, ranged over by* a*,* b*,* $\ldots$*;* process variables*, ranged over by* $X, Y, \ldots$*;* processes*, ranged over by* $P, Q, \ldots$*; and* multiparty sessions*, ranged over by* $M, M', \ldots$*.*

Motion Primitives

When reasoning about communication and synchronisation, the actual trajectory of the system is not important and only the time taken by a motion is important. Therefore, we first abstract away trajectories by just keeping the name of the motion primitive (a, b, $\ldots$) and, for each motion, we assume we know up front how long the action takes. We use the notation $\mathsf{dt}\langle a\rangle$ to represent that a motion primitive executes and time elapses. Every motion can have a different, a priori known, duration denoted $\mathit{duration}(a)$. We write the tuple $\mathsf{dt}\langle(\mathsf{p}_i : a_i)\rangle$ to denote a group of processes executing their respective motion primitives at the same time. For the sake of simplicity, we sometimes use a for both single or grouped motions. In Section 5, we look in more details into the trajectories defined by the joint execution of motion primitives.

Syntax of Motion Session Calculus

A value v can be a natural number n, an integer i, a Boolean true / false, or a real number. An expression e can be a variable, a value, or a term built from expressions by applying (type-correct) computable operators. The processes of the synchronous multiparty session calculus are defined by:

$$\begin{array}{rcl} P & ::= & \mathsf{p}!\ell\langle\mathsf{e}\rangle.P \;\mid\; \sum_{i\in I}\mathsf{p}?\ell_i(x_i).P_i \;\mid\; \sum_{i\in I}\mathsf{p}?\ell_i(x_i).P_i + \mathsf{dt}\langle a\rangle.P \;\mid\; \mathsf{dt}\langle a\rangle.P \\ & \mid & \mathsf{if}\ \mathsf{e}\ \mathsf{then}\ P\ \mathsf{else}\ P \;\mid\; \mu X.P \;\mid\; X \;\mid\; \mathbf{0} \end{array}$$

The output process $\mathsf{p}!\ell\langle\mathsf{e}\rangle.Q$ sends the value of expression e with label ℓ to participant p. The sum of input processes (external choice) $\sum_{i\in I}\mathsf{p}?\ell_i(x_i).P_i$ is a process that can accept a value with label ℓ_i from participant p for any $i \in I$; $\sum_{i\in I}\mathsf{p}?\ell_i(x_i).P_i + \mathsf{dt}\langle a\rangle.P$ is an external choice with a *default branch* with a motion action $\mathsf{dt}\langle a\rangle.P$ which can always proceed when there is no message to receive. According to the label ℓ_i of the received value, the variable x_i is instantiated with the value in the continuation process P_i. We assume that the set I is always finite and non-empty. The conditional process $\mathsf{if}\ \mathsf{e}\ \mathsf{then}\ P\ \mathsf{else}\ Q$ represents the internal choice between processes P and Q. Which branch of the conditional process will be taken depends on the evaluation of the expression e. The process $\mu X.P$ is a recursive process. We assume that the recursive processes are *guarded*. For example, $\mu X.\mathsf{p}?\ell(x).X$ is a valid process, while $\mu X.X$ is not. We often omit $\mathbf{0}$ from the tail of processes.

We define a *multiparty session* as a parallel composition of pairs (denoted by $\mathsf{p} \triangleleft P$) of participants and processes:

$$M \quad ::= \quad \mathsf{p} \triangleleft P \;\mid\; M \mid M$$

with the intuition that process P plays the role of participant p, and can interact with other processes playing other roles in M. The participants correspond to the physical components in the system and the processes correspond to the code run by that physical component. A multiparty session is *well formed* if all its participants are different. We consider only well-formed multiparty sessions.

Operational Semantics of Motion Session Calculus

The value v *of expression* e (notation $\mathsf{e}\downarrow\mathsf{v}$) is computed as expected. We assume that $\mathsf{e}\downarrow\mathsf{v}$ is effectively computable and takes logical "zero time."

We adopt some standard conventions regarding the syntax of processes and sessions. Namely, we will use $\prod_{i\in I}\mathsf{p}_i \triangleleft P_i$ as short for $\mathsf{p}_1 \triangleleft P_1 \mid \ldots \mid \mathsf{p}_n \triangleleft P_n$, where $I = \{1,\ldots,n\}$. We will sometimes use infix notation for external choice process. For example, instead of $\sum_{i\in\{1,2\}}\mathsf{p}?\ell_i(x).P_i$, we will write $\mathsf{p}?\ell_1(x).P_1 + \mathsf{p}?\ell_2(x).P_2$.

The *computational rules of multiparty sessions* are given in Table 1. They are closed with respect to structural congruence. The structural congruence includes a recursion rule $\mu X.P \equiv P\{\mu X.P/X\}$, as well as expected rules for multiparty sessions such as $P \equiv Q \Rightarrow \mathsf{p} \triangleleft P \mid M \equiv \mathsf{p} \triangleleft Q \mid M$. Other rules are standard from [15, 18]. However, unlike the usual treatment of π-calculi, our structural congruence does not have a rule to simplify inactive processes ($\mathsf{p} \triangleleft \mathbf{0}$). The reason is that even when a program might be logically terminated, the physical robot continues to exist and may still collide with another robot. Therefore, in our model, all processes need to terminate at the same time, and so we need to keep $\mathsf{p} \triangleleft \mathbf{0}$.

In rule [COMM], the participant q sends the value v choosing the label ℓ_j to participant p, who offers inputs on all labels ℓ_i with $i \in I$. In rules [T-CONDITIONAL] and [F-CONDITIONAL], the participant p chooses to continue as P if the condition e evaluates to true and as Q if e evaluates to false. Rule [R-STRUCT] states that the reduction relation is closed with respect to structural congruence. We use $\longrightarrow^*$ for the reflexive transitive closure of $\longrightarrow$.

The motion primitives are handled with [MOTION] and [M-PAR]. Here, we need to label transitions with the time taken by the action and propagate these labels with the parallel composition. This ensures that when (physical) time elapses for one process, it elapses

Table 1 Reduction rules. The communication between an output and an external choice (without the default motion action) is formalised similarly to [COMM].

[COMM]

$$\frac{j \in I \qquad \mathsf{e} \downarrow \mathsf{v}}{\mathsf{p} \triangleleft \sum_{i\in I} \mathsf{q}?\ell_i(x).P_i + \mathsf{dt}\langle a\rangle.P \;\mid\; \mathsf{q} \triangleleft \mathsf{p}!\ell_j\langle \mathsf{e}\rangle.Q \longrightarrow \mathsf{p}\triangleleft P_j\{\mathsf{v}/x\} \;\mid\; \mathsf{q}\triangleleft Q}$$

[DEFAULT]

$$\mathsf{p} \triangleleft \sum_{i\in I} \mathsf{q}?\ell_i(x).P_i + \mathsf{dt}\langle a\rangle.P \xrightarrow{\mathsf{dt}\langle a\rangle} \mathsf{p} \triangleleft P$$

[MOTION]

$$\mathsf{p} \triangleleft \mathsf{dt}\langle a\rangle.P \xrightarrow{\mathsf{dt}\langle a\rangle} \mathsf{p} \triangleleft P$$

[T-CONDITIONAL]

$$\frac{\mathsf{e} \downarrow \mathsf{true}}{\mathsf{p} \triangleleft \mathsf{if}\ \mathsf{e}\ \mathsf{then}\ P\ \mathsf{else}\ Q \longrightarrow \mathsf{p} \triangleleft P}$$

[F-CONDITIONAL]

$$\frac{\mathsf{e} \downarrow \mathsf{false}}{\mathsf{p} \triangleleft \mathsf{if}\ \mathsf{e}\ \mathsf{then}\ P\ \mathsf{else}\ Q \longrightarrow \mathsf{p} \triangleleft Q}$$

[R-PAR]

$$\frac{\mathsf{p} \triangleleft Q \longrightarrow \mathsf{p} \triangleleft Q'}{\mathsf{p} \triangleleft Q \mid M \longrightarrow \mathsf{p} \triangleleft Q' \mid M}$$

[M-PAR]

$$\frac{\mathsf{p}_i \triangleleft P_i \xrightarrow{\mathsf{dt}\langle a_i\rangle} \mathsf{p}_i \triangleleft P_i' \qquad \forall i, j.\ \mathit{duration}(a_i) = \mathit{duration}(a_j)}{\Pi_i \mathsf{p}_i \triangleleft P_i \xrightarrow{\mathsf{dt}\langle(\mathsf{p}_i : a_i)\rangle} \Pi_i \mathsf{p}_i \triangleleft P_i'}$$

[R-STRUCT]

$$\frac{M_1' \equiv M_1 \qquad M_1 \longrightarrow M_2 \qquad M_2 \equiv M_2'}{M_1' \longrightarrow M_2'}$$

[M-STRUCT]

$$\frac{M_1' \equiv M_1 \qquad M_1 \xrightarrow{\mathsf{dt}\langle a\rangle} M_2 \qquad M_2 \equiv M_2'}{M_1' \xrightarrow{\mathsf{dt}\langle a\rangle} M_2'}$$

equally for all processes; every process has to spend the same amount of time. This style of synchronisation is reminiscent of broadcast calculi [39]. Instead of broadcast messages, we broadcast *time*.

In order to state that communications can always make progress, we formalise when a multiparty session contains communications or motion actions that will never be executed.

▶ **Definition 3.2.** *A multiparty motion session M is* stuck *if $M \not\equiv \prod_{i\in I} \mathsf{p}_i \triangleleft \mathbf{0}$ and there is no multiparty session M' such that $M \longrightarrow M'$. A multiparty session M gets* stuck*, notation $\mathtt{stuck}(M)$, if it reduces to a stuck motion multiparty session.*

We finish this section with some examples of multi-party sessions.

▶ **Example 3.3** (A Simple Fetch Scenario). Recall the scenario from Section 2 in which a cart and arm assembly has to fetch an object. There are two processes: a cart and an arm; the arm is attached to the cart. The task involves synchronization between the cart and the arm. Specifically, the protocol works as follows. Initially, the cart sends the arm a command to fold. On receiving the command, the arm folds itself. Meanwhile, the cart waits. When the arm is completely folded, it sends back a message to the cart. On receipt of this message, the cart moves, following a trajectory to the object. When it reaches the object, it stops and sends a message back to the arm to grab the object. While the cart waits, the arm executes the grabbing operation, followed by a folding operation. When the arm is done, it again synchronises with the cart. At this point, the cart moves back to its original position. (We simplify the example from Section 2 so that the sequence is not repeated.)

```
$\mathsf{Cart} \triangleleft$
    $\mathsf{Arm}!\mathit{fold}\langle\rangle.$
    $\mathsf{wait}\ (\mathsf{dt}\langle\mathsf{idle}\rangle)\{$
      $\mathsf{Arm}?\mathit{ok}().$
        $\mathsf{dt}\langle\mathsf{move}\rangle.$
        $\mathsf{Arm}!\mathit{grab}.$
        $\mathsf{wait}\ (\mathsf{dt}\langle\mathsf{idle}\rangle)\{$
          $\mathsf{Arm}?\mathit{ok}().$
            $\mathsf{dt}\langle\mathsf{move}\rangle.$
            $\mathsf{Arm}!\mathit{done}().\mathbf{0}$
      $\}$
    $\}$
```

```
$\mathsf{Arm} \triangleleft$
    $\mu X.\mathsf{wait}\ (\mathsf{dt}\langle\mathsf{idle}\rangle)\{$
        $\mathsf{Cart}?\mathit{fold}().\mathsf{dt}\langle\mathsf{fold}\rangle.\mathsf{Cart}!\mathit{ok}\langle\rangle.X$
      $+$ $\mathsf{Cart}?\mathit{grab}().$
          $\mathsf{dt}\langle\mathsf{grip}\rangle.$
          $\mathsf{Cart}!\mathit{ok}\langle\rangle.X$
      $+$ $\mathsf{Cart}?\mathit{done}().\mathbf{0}$
    $\}$
```

Figure 1 A cart and arm example.

Figure 1 shows how the cart and arm processes can be encoded in our core language. We introduce some syntactic sugar for readability. We write $\mathsf{wait}\ (\mathsf{dt}\langle a\rangle)\ \{\ \sum_{i\in I} \mathsf{p}?\ell_i(x_i).P_i\}$ as shorthand for the process $\mu X.\sum_{i\in I} \mathsf{p}?\ell_i(x_i).P_i + \mathsf{dt}\langle a\rangle.X$, which keeps running the default motion a until it receives a message.

The motion primitive idle keeps the cart or the arm stationary. The primitive move moves the cart, the primitives grip and fold respectively move the arm to grab an object or to fold the arm. At this point, we focus on the communication pattern and therefore abstract away the actual trajectories traced by the motion primitives. We come back to the trajectories in Section 5.

Finally, the multiparty session is the parallel composition of the participants Cart and Arm with the corresponding processes.

The processes in our calculus closely follow the syntax of PGCD programs [4]. In Figure 2, we show a side by side comparison of a PGCD program and the corresponding process expressed in the motion session calculus.

▶ **Example 3.4** (Multi-party Co-ordination: Handover). We describe a more complex *handover* example in which a cart and arm assembly transfers an object to a second cart, called the carrier. The process for the arm is identical to Figure 1, but the cart now co-ordinates with the carrier as well. Figure 3 shows all the processes. Note that the cart now synchronises both with the arm and with the carrier.

The protocol is as follows. As before, the cart moves to a target position, having ensured that the arm is folded, and then waits for the carrier to be ready. When the carrier is ready, the arm is instructed to grab an object on the carrier. Once the object is grabbed, the arm synchronises with the cart, which then informs the carrier that the handover is complete. The cart and the carrier move back to their locations and the protocol is complete. The multiparty session is the parallel composition of the participants Cart, Arm, and Carrier, with the corresponding processes.

```
PGCD: pseudo code for the Arm
while true do
    receive (idle)
        fold ⇒
            fold();
            send(Cart, ok)
        grab ⇒
            grip();
            send(Cart, ok)
        done ⇒
            break
```

$$\begin{array}{l}
\mathsf{Arm}\triangleleft \\
\quad \mu X.\mathsf{wait}\ (\mathsf{dt}\langle\mathsf{idle}\rangle)\{ \\
\qquad \mathsf{Cart}?\mathit{fold}(). \\
\qquad\ \ \mathsf{dt}\langle\mathsf{fold}\rangle. \\
\qquad\ \ \mathsf{Cart}!\mathit{ok}\langle\rangle.X \\
\qquad + \ \mathsf{Cart}?\mathit{grab}(). \\
\qquad\ \ \mathsf{dt}\langle\mathsf{grip}\rangle. \\
\qquad\ \ \mathsf{Cart}!\mathit{ok}\langle\rangle.X \\
\qquad + \ \mathsf{Cart}?\mathit{done}(). \\
\qquad\ \ \mathbf{0} \\
\quad \}
\end{array}$$

Figure 2 Comparison of a PGCD code and the corresponding motion session calculus process.

4 Multiparty Motion Session Types

This section introduces motion session types for the calculus presented in Section 3. The formulation is based on [29, 30, 14], with adaptations to account for our motion calculus.

4.1 Motion Session Types and Projections

Global types act as specifications for the message exchanges among robotic components.

▶ **Definition 4.1** (Sorts and global motion session types). Sorts*, ranged over by S, are used to define base types:*

$$S \quad ::= \quad \mathtt{unit} \ \mid \ \mathtt{nat} \ \mid \ \mathtt{int} \ \mid \ \mathtt{bool} \ \mid \ \mathtt{real}$$

Global types*, ranged over by G, are terms generated by the following grammar:*

$$G \quad ::= \quad \mathsf{dt}\langle(\mathsf{p}_i : a_i)\rangle.G \ \mid \ \mathsf{p} \to \mathsf{q} : \{\ell_i(S_i).G_i\}_{i\in I} \ \mid \ \mathbf{t} \ \mid \ \mu\mathbf{t}.G \ \mid \ \mathtt{end}$$

We require that $\mathsf{p} \neq \mathsf{q}$, $I \neq \emptyset$, $\ell_i \neq \ell_j$, and $duration(a_i) = duration(a_j)$ whenever $i \neq j$, for all $i, j \in I$. We postulate that recursion is guarded and recursive types with the same regular tree are considered equal [37, Chapter 20, Section 2].

In Definition 4.1, the type $\mathsf{dt}\langle(\mathsf{p}_i : a_i)\rangle.G$ is a *motion global type* which explicitly declares a *synchronisation* by a motion action among all the participants p_i. The rest is the standard definition of global types in multiparty session types [29, 30, 14]. The *branching* type $\mathsf{p} \to \mathsf{q} : \{\ell_i(S_i).G_i\}_{i\in I}$ formalises a protocol where participant p must send to q one message with label ℓ_i and a value of type S_i as payload, for some $i \in I$; then, depending on which ℓ_i was sent by p, the protocol continues as G_i. Value types are restricted to sorts. The type $\mathtt{end}$ represents a terminated protocol. A recursive protocol is modelled as $\mu\mathbf{t}.G$, where recursion variable $\mathbf{t}$ is bound and guarded in G, e.g., $\mu\mathbf{t}.\mathbf{t}$ is not a valid type. The notation $\mathtt{pt}\{G\}$ denotes a set of participants of a global type G.

```
Cart◁
    Arm!fold⟨⟩.
    wait (dt⟨idle⟩){
      Arm?ok().Carrier!ok⟨⟩.
       dt⟨move⟩.
       wait (dt⟨idle⟩){
         Carrier?ok().Arm!grab⟨⟩.
         wait (dt⟨idle⟩){
           Arm?ok().Carrier!ok⟨⟩.
           dt⟨move⟩.
           Arm!done⟨⟩.Carrier!done⟨⟩.0
         }
      }
    }
```

```
Carrier◁
      wait (dt⟨idle⟩){
        Cart?ok().dt⟨move⟩.
         Cart!ok⟨⟩.
         wait (dt⟨idle⟩){
           Cart?ok().
            dt⟨move⟩.
            wait (dt⟨idle⟩){Cart?done().0}
      }}
  Arm◁
      μX.wait (dt⟨idle⟩){
          Cart?fold().dt⟨fold⟩.Cart!ok⟨⟩.X
        + Cart?grab().dt⟨grip⟩.Cart!ok⟨⟩.X
        + Cart?done().0
      }
```

Figure 3 A multi-party handover example.

▶ **Example 4.2** (Global session types). The global session type for the fetch example (Example 3.3) is:

$$
\begin{array}{l}
\mathsf{Cart} \to \mathsf{Arm} : \mathit{fold}(\mathtt{unit}).\mathsf{dt}\langle \mathsf{Cart} : \mathsf{idle}, \mathsf{Arm} : \mathsf{fold}\rangle.\\
\mathsf{Arm} \to \mathsf{Cart} : \mathit{ok}(\mathtt{unit}).\mathsf{dt}\langle \mathsf{Cart} : \mathsf{move}, \mathsf{Arm} : \mathsf{idle}\rangle.\\
\mathsf{Cart} \to \mathsf{Arm} : \mathit{grab}(\mathtt{unit}).\mathsf{dt}\langle \mathsf{Cart} : \mathsf{idle}, \mathsf{Arm} : \mathsf{grip}\rangle.\\
\mathsf{Arm} \to \mathsf{Cart} : \mathit{ok}(\mathtt{unit}).\mathsf{dt}\langle \mathsf{Cart} : \mathsf{move}, \mathsf{Arm} : \mathsf{idle}\rangle.\\
\mathsf{Cart} \to \mathsf{Arm} : \mathit{done}(\mathtt{unit}).\mathsf{end}
\end{array}
$$

and the global session type for the handover example (Example 3.4) is:

$$
\begin{array}{l}
\mathsf{Cart} \to \mathsf{Arm} : \mathit{fold}(\mathtt{unit}).\mathsf{dt}\langle \mathsf{Cart} : \mathsf{idle}, \mathsf{Carrier} : \mathsf{idle}, \mathsf{Arm} : \mathsf{fold}\rangle.\\
\mathsf{Arm} \to \mathsf{Cart} : \mathit{ok}(\mathtt{unit}).\mathsf{Cart} \to \mathsf{Carrier} : \mathit{ok}(\mathtt{unit}).\\
\mathsf{dt}\langle \mathsf{Cart} : \mathsf{move}, \mathsf{Carrier} : \mathsf{move}, \mathsf{Arm} : \mathsf{idle}\rangle.\\
\mathsf{Carrier} \to \mathsf{Cart} : \mathit{ok}(\mathtt{unit}).\mathsf{Cart} \to \mathsf{Arm} : \mathit{grab}(\mathtt{unit}).\\
\mathsf{dt}\langle \mathsf{Cart} : \mathsf{idle}, \mathsf{Carrier} : \mathsf{idle}, \mathsf{Arm} : \mathsf{grip}\rangle.\\
\mathsf{Arm} \to \mathsf{Cart} : \mathit{ok}(\mathtt{unit}).\mathsf{Cart} \to \mathsf{Carrier} : \mathit{ok}(\mathtt{unit}).\\
\mathsf{dt}\langle \mathsf{Cart} : \mathsf{move}, \mathsf{Carrier} : \mathsf{move}, \mathsf{Arm} : \mathsf{idle}\rangle.\\
\mathsf{Cart} \to \mathsf{Arm} : \mathit{done}(\mathtt{unit}).\mathsf{Cart} \to \mathsf{Carrier} : \mathit{done}(\mathtt{unit}).\mathsf{end}
\end{array}
$$

A *(local) motion session type* describes the behaviour of a single participant in a multiparty motion session.

▶ **Definition 4.3** (Local motion session types). *The grammar of local types, ranged over by T, is:*

$$\begin{array}{lll} T & ::= & \mathsf{dt}\langle a\rangle.T \;\mid\; \&\{\mathsf{p}?\ell_i(S_i).T_i\}_{i\in I} \;\mid\; \&\{\mathsf{p}?\ell_i(S_i).T_i\}_{i\in I} \,\&\, \mathsf{dt}\langle a\rangle.T \;\mid\; \oplus\{\mathsf{q}!\ell_i(S_i).T_i\}_{i\in I} \\ & \mid & \mathbf{t} \;\mid\; \mu\mathbf{t}.T \;\mid\; \mathtt{end} \end{array}$$

We require that $\ell_i \neq \ell_j$ whenever $i \neq j$, for all $i, j \in I$. We postulate that recursion is always guarded. Unless otherwise noted, session types are closed.

Labels in a type need to be pairwise different, e.g., $\mathsf{p}?\ell(\mathtt{int}).\mathtt{end}\&\mathsf{p}?\ell(\mathtt{nat}).\mathtt{end}$ is not a type. The *motion local type* $\mathsf{dt}\langle a\rangle.T$ represents a motion action followed by the type T; the *external choice* or *branching type* $\&\{\mathsf{p}?\ell_i(S_i).T_i\}_{i\in I}$ requires to wait to receive a value of sort S_i (for some $i \in I$) from the participant p, via a message with label ℓ_i; if the received message has label ℓ_i, the protocol will continue as prescribed by T_i. The *motion branching choice* is equipped with a default motion type $\mathsf{dt}\langle a\rangle.T$. The *internal choice* or *selection type* $\oplus\{\mathsf{q}!\ell_i(S_i).T_i\}_{i\in I}$ says that the participant implementing the type must choose a labelled message to send to q; if the participant chooses the message ℓ_i, for some $i \in I$, it must include in the message to q a payload value of sort S_i, and continue as prescribed by T_i. Recursion is modelled by the session type $\mu\mathbf{t}.T$. The session type $\mathtt{end}$ says that no further communication is possible and the protocol is completed. We adopt the following conventions: we do not write branch/selection symbols in case of a singleton choice, we do not write unnecessary parentheses, and we often omit trailing $\mathtt{end}$s. The notation $\mathtt{pt}\{T\}$ denotes a set of participants of a session type T.

In Definition 4.4 below, we define the *global type projection* as a relation $G \restriction_{\mathsf{r}} T$ between global and local types. Our definition extends the one originally proposed by [25, 26], along the lines of [12] and [13] with motion types: i.e., it uses a *merging operator* $\bigsqcap$ to combine multiple session types into a single type.

▶ **Definition 4.4.** *The* projection of a global type onto a participant r *is the largest relation $\restriction_r$ between global and session types such that, whenever $G \restriction_r T$:*

- $G = \mathtt{end}$ *implies* $T = \mathtt{end}$*;* [PROJ-END]
- $G = \mathsf{dt}\langle(\mathsf{p}_i : a_i)\rangle.G'$ *implies* $T = \mathsf{dt}\langle a_j\rangle.T'$ *with* $r = \mathsf{p}_j$ *and* $G' \restriction_r T'$*;* [PROJ-MOTION]
- $G = \mathsf{p} \to \mathsf{r} : \{\ell_i(S_i).G_i\}_{i\in I}$ *implies* $T = \&\{\mathsf{p}?\ell_i(S_i).T_i\}_{i\in I}$ *with* $G_i \restriction_r T_i$*;* [PROJ-IN]
- $G = \mathsf{r} \to \mathsf{q} : \{\ell_i(S_i).G_i\}_{i\in I}$ *implies* $T = \oplus\{\mathsf{q}!\ell_i(S_i).T_i\}_{i\in I}$ *and* $G_i \restriction_r T_i$, $\forall i \in I$*;* [PROJ-OUT]
- $G = \mathsf{p} \to \mathsf{q} : \{\ell_i(S_i).G_i\}_{i\in I}$ *and* $\mathsf{r} \notin \{\mathsf{p}, \mathsf{q}\}$ *implies that there are* T_i, $i \in I$ *s.t.* [PROJ-CONT]
  $T = \bigsqcap_{i\in I} T_i$, *and* $G_i \restriction_r T_i$, *for every* $i \in I$.
- $G = \mu\mathbf{t}.\mathsf{G}$ *implies* $T = \mu\mathbf{t}.T'$ *with* $G \restriction_r T'$ *if* r *occurs in* G, *otherwise* $T = \mathtt{end}$. [PROJ-REC]

Above, $\bigsqcap$ *is the* merging operator*, that is a partial operation over session types defined as:*

$$T_1 \sqcap T_2 \;=\; \left\{\begin{array}{lll}
T_1 & \textit{if } T_1 = T_2 & [\textsc{mrg-id}] \\
T_3 & \textit{if } \exists I, J : \left\{\begin{array}{ll} T_1 = \&\{\mathsf{p}'?\ell_i(S_i).T_i\}_{i\in I} & \textit{and} \\ T_2 = \&\{\mathsf{p}'?\ell_j(S_j).T_j\}_{j\in J} & \textit{and} \\ T_3 = \&\{\mathsf{p}'?\ell_k(S_k).T_k\}_{k\in I\cup J} & \end{array}\right. & [\textsc{mrg-bra}1] \\
T_3 & \textit{if } \exists I, J : \left\{\begin{array}{ll} T_1 = \&\{\mathsf{p}'?\ell_i(S_i).T_i\}_{i\in I} \& \ \mathsf{dt}\langle a\rangle.T' & \textit{and} \\ T_2 = \&\{\mathsf{p}'?\ell_j(S_j).T_j\}_{j\in J} \& \ \mathsf{dt}\langle a\rangle.T' & \textit{and} \\ T_3 = \&\{\mathsf{p}'?\ell_k(S_k).T_k\}_{k\in I\cup J} \& \ \mathsf{dt}\langle a\rangle.T' & \end{array}\right. & [\textsc{mrg-bra}2] \\
T_3 & \textit{if } \exists I, J : \left\{\begin{array}{ll} T_1 = \&\{\mathsf{p}'?\ell_i(S_i).T_i\}_{i\in I} & \textit{and} \\ T_2 = \&\{\mathsf{p}'?\ell_j(S_j).T_j\}_{j\in J} \& \ \mathsf{dt}\langle a\rangle.T' & \textit{and} \\ T_3 = \&\{\mathsf{p}'?\ell_k(S_k).T_k\}_{k\in I\cup J} \& \ \mathsf{dt}\langle a\rangle.T' & \end{array}\right. & [\textsc{mrg-bra}3] \\
T_3 & \textit{if } \exists I, J : \left\{\begin{array}{ll} T_1 = \mathsf{dt}\langle a\rangle.T' & \textit{and} \\ T_2 = \&\{\mathsf{p}'?\ell_i(S_i).T_i\}_{i\in I} \& \ \mathsf{dt}\langle a\rangle.T' & \textit{and} \\ T_3 = \&\{\mathsf{p}'?\ell_i(S_i).T_i\}_{i\in I} \& \ \mathsf{dt}\langle a\rangle.T' & \end{array}\right. & [\textsc{mrg-bra}4] \\
T_3 & \textit{if } \exists I, J : \left\{\begin{array}{ll} T_1 = \mathsf{dt}\langle a\rangle.T' & \textit{and} \\ T_2 = \&\{\mathsf{p}'?\ell_j(S_j).T_j\}_{i\in I} & \textit{and} \\ T_3 = \&\{\mathsf{p}'?\ell_i(S_i).T_i\}_{i\in I} \& \ \mathsf{dt}\langle a\rangle.T' & \end{array}\right. & [\textsc{mrg-bra}5] \\
T_2 \sqcap T_1 & \textit{if } T_2 \sqcap T_1 \textit{ is defined,} & \\
\textit{undefined} & \textit{otherwise.} &
\end{array}\right.$$

We omit the cases for recursions and selections (defined as [42, S 3]).

Note that our definition is slightly simplified w.r.t. the one of [12] and [13]. Instead of this mergeability operator, one might use more general approach from [42]. This definition is sufficient for our purposes (i.e., to demonstrate an application of session types to robotics communications).

▶ **Example 4.5.** The projection of the global session type for the fetch example on the cart gives the following local session type:

$$\begin{array}{l} \mathsf{Arm}!\mathit{fold}\langle\texttt{unit}\rangle.\mathsf{dt}\langle\mathsf{idle}\rangle.\mathsf{Arm}?\mathit{ok}(\texttt{unit}).\mathsf{dt}\langle\mathsf{move}\rangle.\mathsf{Arm}!\mathit{grab}\langle\texttt{unit}\rangle. \\ \qquad \mathsf{dt}\langle\mathsf{idle}\rangle.\mathsf{Arm}?\mathit{ok}(\texttt{unit}).\mathsf{dt}\langle\mathsf{move}\rangle.\mathsf{Arm}!\mathit{done}\langle\texttt{unit}\rangle.\texttt{end} \end{array}$$

The local motion session type for the arm is:

$$\begin{array}{l} \mathsf{Cart}?\mathit{fold}(\texttt{unit}).\mathsf{dt}\langle\mathsf{fold}\rangle.\mathsf{Cart}!\mathit{ok}\langle\texttt{unit}\rangle.\mathsf{dt}\langle\mathsf{idle}\rangle.\mathsf{Cart}?\mathit{grab}(\texttt{unit}). \\ \qquad \mathsf{dt}\langle\mathsf{grip}\rangle.\mathsf{Cart}!\mathit{ok}\langle\texttt{unit}\rangle.\mathsf{dt}\langle\mathsf{idle}\rangle.\mathsf{Cart}?\mathit{done}(\texttt{unit}).\texttt{end} \end{array}$$

On Progress of Time. Our model assumes that the computation and message transmission time is much faster than the dynamics of the system and, therefore, the messages can be seen as instantaneous. This assumption depends on parameters of the system, like the speed of the network and the dynamics of the physical system, and also on the program being executed. While we cannot directly change the physical system, we can at least check the program is well behaved w.r.t. to time.

If a program can send an unbounded number of messages without executing a motion then this assumption, obviously, does not hold. From the perspective of using the motion calculus to verify a system, this may lead to situation where an unsafe program is deemed safe because time does not progress. For instance, a robot driving straight into a wall could “avoid” crashing into the wall by sending messages in a loop and, therefore, stopping the progress of time.

This problem is not unique to our system but a more general problem in defining the semantics of hybrid systems [20, 21]. In general, one needs to assume that time always *diverges* for infinite executions. In this work, we take a pragmatic solution and simply *disallow*

0-time recursion. When recursion is used, all the paths between a $\mu\mathbf{t}$ and the corresponding $\mathbf{t}$ must contain at least one motion primitive. This is a simple check which can be done at the syntactic level of global types and it is a sufficient condition for forcing the progress of time.

4.2 Motion Session Typing

We now introduce a type system for the multiparty session calculus presented in Section 3. We distinguish three kinds of typing judgments:

$$\Gamma \vdash \mathsf{e} : S \qquad\qquad \Gamma \vdash P : T \qquad\qquad \vdash M : G$$

where Γ is the *typing environment* defined as: $\Gamma \ ::= \ \emptyset \ \mid \ \Gamma, x : S \ \mid \ \Gamma, X : T$, i.e., a mapping that associates expression variables with sorts, and process variables with session types.

We use the subtyping relation $\leqslant$ to augment the flexibility of the type system by determining when a type T is "smaller" than T', it allows to use a process typed by the former whenever a process typed by the latter is required.

▶ **Definition 4.6** (Subsorting and subtyping). Subsorting $\leq:$ *is the least reflexive binary relation such that* `nat` $\leq:$ `int` $\leq:$ `real`. Subtyping $\leqslant$ *is the largest relation between session types coinductively defined by the following rules:*

[SUB-END]
$$\mathtt{end} \leqslant \mathtt{end}$$

[SUB-IN1]
$$\frac{\forall i \in I : \quad S'_i \leq: S_i \quad T_i \leqslant T'_i \quad T \leqslant T'}{\&\{\mathsf{p}?\ell_i(S_i).T_i\}_{i \in I \cup J} \ \& \ \mathsf{dt}\langle a \rangle.T \leqslant \&\{\mathsf{p}?\ell_i(S'_i).T'_i\}_{i \in I} \ \& \ \mathsf{dt}\langle a \rangle.T'}$$

[SUB-MOTION]
$$\frac{T \leqslant T'}{\mathsf{dt}\langle a \rangle.T \leqslant \mathsf{dt}\langle a \rangle.T'}$$

[SUB-IN2]
$$\frac{\forall i \in I : \quad S'_i \leq: S_i \quad T_i \leqslant T'_i}{\&\{\mathsf{p}?\ell_i(S_i).T_i\}_{i \in I \cup J} \ \& \ \mathsf{dt}\langle a \rangle.T \leqslant \&\{\mathsf{p}?\ell_i(S'_i).T'_i\}_{i \in I}}$$

[SUB-IN3]
$$\frac{T \leqslant T'}{\&\{\mathsf{p}?\ell_i(S_i).T_i\}_{i \in I} \ \& \ \mathsf{dt}\langle a \rangle.T \leqslant \mathsf{dt}\langle a \rangle.T'}$$

[SUB-OUT]
$$\frac{\forall i \in I : \quad S_i \leq: S'_i \quad T_i \leqslant T'_i}{\oplus\{\mathsf{p}!\ell_i(S_i).T_i\}_{i \in I} \leq: \oplus\{\mathsf{p}!\ell_i(S'_i).T'_i\}_{i \in I \cup J}}$$

The double line in the subtyping rules indicates that the rules are interpreted *coinductively* [37, Chapter 21].

The typing rules for expressions are given as expected and omitted. The typing rules for processes and multiparty sessions are the content of Table 2:

- [T-SUB] is the *subsumption rule*: a process with type T is also typed by the supertype T';
- [T-**0**] says that a terminated process implements the terminated session type;
- [T-REC] types a recursive process $\mu X.P$ with T if P can be typed as T, too, by extending the typing environment with the assumption that X has type T;
- [T-VAR] uses the typing environment assumption that process X has type T;
- [T-MOTION] types a motion process as a motion local type;
- [T-INPUT-CHOICE] types a summation of input prefixes as a branching type and a default branch as a motion type. It requires that each input prefix targets the same participant q, and that, for all $i \in I$, each continuation process P_i is typed by the continuation type T_i, having the bound variable x_i in the typing environment with sort S_i. Note that the rule implicitly requires the process labels ℓ_i to be pairwise distinct (as per Definition 4.3);

Table 2 Typing rules for motion processes.

[T-0]
$$\Gamma \vdash \mathbf{0} : \mathsf{end}$$

[T-REC]
$$\frac{\Gamma, X : T \vdash P : T}{\Gamma \vdash \mu X.P : T}$$

[T-VAR]
$$\Gamma, X : T \vdash X : T$$

[T-MOTION]
$$\frac{\Gamma \vdash Q : T}{\Gamma \vdash \mathsf{dt}\langle a \rangle.Q : \mathsf{dt}\langle a \rangle.T}$$

[T-OUT]
$$\frac{\Gamma \vdash \mathsf{e} : S \quad \Gamma \vdash P : T}{\Gamma \vdash \mathsf{q}!\ell(\mathsf{e}).P : \mathsf{q}!\ell(S).T}$$

[T-INPUT-CHOICE1]
$$\frac{\forall i \in I \quad \Gamma, x_i : S_i \vdash P_i : T_i}{\Gamma \vdash \sum_{i \in I} \mathsf{q}?\ell_i(x_i).P_i : \&\{\mathsf{q}?\ell_i(S_i).T_i\}_{i \in I}}$$

[T-INPUT-CHOICE2]
$$\frac{\forall i \in I \qquad \Gamma, x_i : S_i \vdash P_i : T_i \qquad \Gamma \vdash \mathsf{dt}\langle a \rangle.Q : T}{\Gamma \vdash \sum_{i \in I} \mathsf{q}?\ell_i(x_i).P_i + \mathsf{dt}\langle a \rangle.Q : \&\{\mathsf{q}?\ell_i(S_i).T_i\}_{i \in I} \;\&\; T}$$

[T-CHOICE]
$$\frac{\Gamma \vdash \mathsf{e} : \mathtt{bool} \qquad \exists k \in I \qquad \Gamma \vdash P_1 : T_k \qquad \Gamma \vdash P_2 : \oplus\{T_i\}_{i \in I \setminus \{k\}}}{\Gamma \vdash \mathsf{if\ e\ then}\ P_1\ \mathsf{else}\ P_2 : \oplus\{T_i\}_{i \in I}}$$

[T-SUB]
$$\frac{\Gamma \vdash P : T \quad T \leqslant T'}{\Gamma \vdash P : T'}$$

[T-SESS]
$$\frac{\forall i \in I \quad \vdash P_i : G\!\upharpoonright\!\mathsf{p}_i \qquad \mathtt{pt}\{G\} = \{\mathsf{p}_i \mid i \in I\}}{\vdash \prod_{i \in I} \mathsf{p}_i \triangleleft P_i : G}$$

- [T-OUT] types an output prefix with a singleton selection type, provided that the expression in the message payload has the correct sort S, and the process continuation matches the type continuation;
- [T-CHOICE] types a conditional process by matching the branches of the types to branches of the sub-processes;
- [T-SESS] types multiparty sessions, by associating typed processes to participants. It requires that the processes being composed in parallel can play as participants of a global communication protocol: hence, their types must be projections of a single global type G. As the temporal evolution (motion) synchronises all the processes condition $\mathtt{pt}\{G\} = \{\mathsf{p}_i \mid i \in I\}$ guarantees that motions are defined for every participant.

▶ **Example 4.7.** We sketch the main steps to show that the Arm process is typed by the local type from Example 4.5. The type derivation uses the subtyping rules. This is because the process for the arm makes an external choice between the messages *fold*, *grab*, *done*, and the default motion primitive idle, and the type fixes a specific sequence of messages. The usual subtyping rules [SUB-IN1] and [SUB-IN2] allow typing the process against the local type, by "expanding" the local type with the other possible choices. The interesting subtyping rule is [SUB-IN3], which states that an external choice with a default motion type refines only the default motion type. This is needed to type the process against the local type

$$\mathsf{dt}\langle \mathsf{idle} \rangle.\mathsf{Cart}?\mathit{grab}(\mathtt{unit}).T$$

This subtyping rule is sound, because the local type ensures that the other message choices cannot arise.

The proposed motion session type system satisfies two fundamental properties: typed sessions only reduce to typed sessions (subject reduction), and typed sessions never get stuck.

In order to state subject reduction, we need to formalise how global types are reduced when local session types reduce and evolve. Note that since the same motion actions always synchronise among all participants, they always make progress (hence they are always consumed).

▶ **Definition 4.8** (Global types consumption and reduction). *The* consumption *of the communication $\mathsf{p} \xrightarrow{\ell} \mathsf{q}$ and motion $\mathsf{dt}\langle a\rangle$ for the global type G (notation $G \setminus \mathsf{p} \xrightarrow{\ell} \mathsf{q}$ and $G \setminus \mathsf{dt}\langle a\rangle$) is the global type defined (up to unfolding of recursive types) as follows:*

$$\begin{aligned}
\mathsf{dt}\langle a\rangle.G \setminus \mathsf{dt}\langle a\rangle &= G \\
\big(\mathsf{p} \to \mathsf{q} : \{\ell_i(S_i).G_i\}_{i\in I}\big) \setminus \mathsf{p} \xrightarrow{\ell} \mathsf{q} &= G_k \qquad\qquad \text{if } \exists k \in I : \ell = \ell_k \\
\big(\mathsf{r} \to \mathsf{s} : \{\ell_i(S_i).G_i\}_{i\in I}\big) \setminus \mathsf{p} \xrightarrow{\ell} \mathsf{q} &= \mathsf{r} \to \mathsf{s} : \{\ell_i(S_i).G_i \setminus \mathsf{p} \xrightarrow{\ell} \mathsf{q}\}_{i\in I} \\
&\qquad \text{if } \{\mathsf{r},\mathsf{s}\} \cap \{\mathsf{p},\mathsf{q}\} = \emptyset \ \wedge\ \forall i \in I : \{\mathsf{p},\mathsf{q}\} \subseteq G_i
\end{aligned}$$

The reduction of global types *is the smallest pre-order relation closed under the rule: $G \Longrightarrow G \setminus \mathsf{p} \xrightarrow{\ell} \mathsf{q}$ and $G \Longrightarrow G \setminus \mathsf{dt}\langle a\rangle$.*

We can now state the main results.

▶ **Theorem 4.9** (Subject Reduction). *Let $\vdash M : G$. For all M', if $M \longrightarrow M'$, then $\vdash M' : G'$ for some G' such that $G \Longrightarrow G'$.*

▶ **Corollary 4.10.** *Let $\vdash M : G$. If $M \longrightarrow^* M'$, then $\vdash M' : G'$ for some G' such that $G \Longrightarrow G'$.*

▶ **Theorem 4.11** (Progress). *If $\vdash M : G$, then either $M \equiv \prod_{i\in I} \mathsf{p}_i \triangleleft \mathbf{0}$ or there is M' such that $M \longrightarrow M'$.*

As a consequence of subject reduction and progress, we get the safety property stating that a typed multiparty session will never get stuck.

▶ **Theorem 4.12** (Type Safety). *If $\vdash M : G$, then it does not hold $\mathtt{stuck}(M)$.*

Proof. Direct consequence of Corollary 4.10, Theorem 4.11, and Definition 3.2. ◀

5 Motion Primitives: Trajectories and Resources

So far, our motion calculus abstracted the trajectories of the robots and only considered the time it takes to execute motion primitives. This is sufficient to show that the synchronisation and communication protocol between the robots executes correctly. However, it is too abstract to prove more complex properties about executions of the system. In particular, for an execution to proceed correctly we need to check the existence of trajectories for all the robots. A joint trajectory may not exist, for example, if the motion primitives cause a collision in the physical world.

In this section, we explain how to make our model more detailed and how to look inside the motion primitives for the continuous evolution of trajectories. To accomplish this, first, we give a semantics that includes trajectories. Then, we refine our calculus to replace internal choice with guarded choice. Finally, we explain how to use session types to prove properties over the trajectories.

5.1 Model for the Robots and Motion Primitives

We proceed following the formalisation of trajectories in the PGCD language for robotics [4].

Robots. Each participant $(\mathsf{p}, \mathsf{q}, \ldots)$ maintains a state in the physical world. This state is updated when its own motion primitives execute as well as on potential physical interactions with other processes.

We model the physical state of a process as a tuple $(\mathit{Var}, \rho, \mathit{rsrc})$ where Var is a set of variables, with two distinguished disjoint subsets X and W of *physical state* and *external input* variables, $\rho : \mathit{Var} \to \mathbb{R}$ is a *store* mapping variables to values, and rsrc is a *resource function* mapping a store to a subset of $\mathbb{R}^3$. The resource function represents the geometric footprint in space occupied by the robot. We shall use this function to check the absence of collisions between robots.

When two robots p_1 and p_2 are in the same environment, we may connect some state variables of one process to the external inputs of the other. This represents physical coupling between these robots. A *connection* θ between p_1 and p_2 is a finite set of pairs of variables, $\theta = \{(x_i, w_i) \mid i = 1, \ldots, m\}$, such that: (1) for each $(x, w) \in \theta$, we have $x \in \mathsf{p}_1.X$ and $w \in \mathsf{p}_2.W$ or $x \in \mathsf{p}_2.X$ and $w \in \mathsf{p}_1.W$, and (2) there does not exist $(x, w), (x', w) \in \theta$ such that x and x' are distinct. Two connections θ_1 and θ_2 are *compatible* if $\theta_1 \cup \theta_2$ is a connection. We assume that all the participants in a session are connected by compatible connections.

For example, consider a cart and an arm. The physical variables can provide the position and velocities of the center of mass of the cart and of the arm. Note that if the arm is attached to the cart, then its position changes when the cart moves. Thus, the position and velocity of the cart are external inputs to the arm, and play a role in determining its own position. However, the arm can also move relative to the cart and the position of its end effector is determined both by the external inputs as well as its relative position and velocity. Furthermore, the mass and the position of the center of mass of the arm are external inputs to the cart, because these variables affect the dynamics of the cart.

Motion Primitives. Let X and W be two sets of real-valued variables, representing internal state and external input variables of a robotic system, respectively. A motion primitive updates the values of the variables in X over time, while respecting the values of variables in W set by the external world. This dynamic process results in a pair of state and input trajectories (ξ, ν), i.e., a valuation over time to variables in X and W.

Formally, a motion primitive m is a tuple $(T, \mathrm{Pre}, \mathrm{Inv}, \mathrm{Post})$ consisting of a *duration* T, a *pre-condition* $\mathrm{Pre} \subseteq \mathbb{R}^{|X|} \times \mathbb{R}^{|W|}$, an *invariant* $\mathrm{Inv} \subseteq \big([0, T] \to \mathbb{R}^{|X|}\big) \times \big([0, T] \to \mathbb{R}^{|W|}\big)$, and a *post-condition* $\mathrm{Post} \subseteq \mathbb{R}^{|X|} \times \mathbb{R}^{|W|}$. A *trajectory* of duration T of the motion primitive m is a pair of continuous functions (ξ, ν) mapping the real interval $[0, T]$ to $\mathbb{R}^{|X|}$ and $\mathbb{R}^{|W|}$, respectively, such that $(\xi, \nu) \in \mathrm{Inv}$, $(\xi(0), \nu(0)) \in \mathrm{Pre}$, and $(\xi(T), \nu(T)) \in \mathrm{Post}$.

Correspondingly, we need to update the semantics of our motion calculus:

- The participant executing a program $\mathsf{p} \triangleleft P$ now also carries a store containing a valuation for the physical state of the robot: $\mathsf{p}, \rho \triangleleft P$.
- The motion transitions $\xrightarrow{\mathsf{dt}\langle a \rangle}$ get labelled with trajectories: $\xrightarrow{\mathsf{dt}\langle(\xi, \nu)\rangle}$.
- The semantics rule for choice can use values from the store:

$$\text{[T-CONDITIONAL]}\quad \frac{\rho(\mathsf{e}) \downarrow \mathsf{true}}{\mathsf{p}, \rho \triangleleft \mathsf{if\ e\ then}\ P\ \mathsf{else}\ Q \longrightarrow \mathsf{p}, \rho \triangleleft P} \qquad \text{[F-CONDITIONAL]}\quad \frac{\rho(\mathsf{e}) \downarrow \mathsf{false}}{\mathsf{p}, \rho \triangleleft \mathsf{if\ e\ then}\ P\ \mathsf{else}\ Q \longrightarrow \mathsf{p}, \rho \triangleleft Q}$$

where $\rho(\mathsf{e})$ replaces the variables from Var in e with their value according to ρ.

- The semantics of a motion checks the trajectories against the motion primitive specification and the store:

$$\textsc{[motion]}\quad \frac{\begin{array}{c} a = (T, \mathrm{Pre}, \mathrm{Inv}, \mathrm{Post}) \qquad \mathit{range}(\xi) = [0, T] \qquad \rho = \xi(0) \qquad \rho' = \xi(T) \\ (\xi(0), \nu(0)) \in \mathrm{Pre} \qquad (\xi(T), \nu(T)) \in \mathrm{Post} \qquad \forall t \in [0, T].\, (\xi(t), \nu(t)) \in \mathrm{Inv} \end{array}}{\mathsf{p}, \rho \lhd \mathsf{dt}\langle a \rangle.P \xrightarrow{\mathsf{dt}\langle(\xi,\nu)\rangle} \mathsf{p}, \rho' \lhd P}$$

  The rule checks that the trajectory is valid w.r.t. a: the duration of the trajectory must match the duration of the motion primitive, the start and end of the trajectory match the state of ρ and ρ' respectively. Furthermore, the pre-condition, post-condition, and invariant must be respected.
- The parallel composition of motions connects the external inputs of each process according to the connections. For the notations, we use subscript to denote that an element belongs to a particular process p, e.g., X_{p} for the internal variables of p. We denote the restriction of a trajectory ξ over a subset X of the dimensions by $\xi|_X$.

$$\textsc{[m-par]}\quad \frac{\begin{array}{c} \forall i \qquad \xi_i = \xi|_{X_{\mathsf{p}_i}} \qquad \nu_i = \theta_{\mathsf{p}_i}(\xi)|_{W_{\mathsf{p}_i}} \qquad \mathsf{p}_i, \rho_i \lhd P_i \xrightarrow{\mathsf{dt}\langle(\xi_i,\nu_i)\rangle} \mathsf{p}_i, \rho'_i \lhd P'_i \\ \forall i, j, t.\ i \neq j \Rightarrow \mathit{rsrc}_{\mathsf{p}_i}(\xi|_{X_{\mathsf{p}_i}}(t), \theta_{\mathsf{p}_i}(\xi)|_{W_{\mathsf{p}_i}}(t)) \cap \mathit{rsrc}_{\mathsf{p}_j}(\xi|_{X_{\mathsf{p}_j}}(t), \theta_{\mathsf{p}_j}(\xi)|_{W_{\mathsf{p}_j}}(t)) = \emptyset \end{array}}{\Pi_i \mathsf{p}_i, \rho_i \lhd P_i \xrightarrow{\mathsf{dt}\langle(\xi,\nu)\rangle} \Pi_i \mathsf{p}_i, \rho'_i \lhd P'_i}$$

  Even at the top level, there is a ν as there can be elements which are under the control of the environment. Then, for each process we create the appropriate trajectory (ξ, ν) by applying the appropriate connection θ. Also, the resources used by each participants during the motion needs to disjoint from each other. This last check ensures the absence of collision between robots. We use this check to avoid the complexity of modelling collisions.

▶ **Example 5.1.** Let us look at the cart from Example 3.3. The cart is moving on the ground, a 2D plane and, therefore, we model its physical state (X_{Cart}) by its position $\mathbf{p}_{\mathsf{Cart}} \in \mathbb{R}^2$, orientation $\mathbf{r}_{\mathsf{Cart}} \in [-\pi; \pi)$, and speed $\mathbf{s}_{\mathsf{Cart}} \in \mathbb{R}$.

A trivial motion primitive $\mathbf{idle}(\mathbf{p}_0, \mathbf{r}_0)$ keeps the cart at its current position $\mathbf{p}_0$ and orientation $\mathbf{r}_0$; the pre-condition is $\mathbf{s}_{\mathsf{Cart}} = 0$ (i.e., it is at rest), the post-condition is $\mathbf{s}_{\mathsf{Cart}} = 0 \wedge \mathbf{p}_{\mathsf{Cart}} = \mathbf{p}_0 \wedge \mathbf{r}_{\mathsf{Cart}} = \mathbf{r}_0$, and the invariant is $\mathbf{p}_{\mathsf{Cart}}(t) = \mathbf{p}_0 \wedge \mathbf{r}_{\mathsf{Cart}}(t) = \mathbf{r}_0 \wedge \mathbf{s}_{\mathsf{Cart}} = 0$ for all $t \in [0, T]$.

A slightly more interesting motion primitive is $\mathsf{move}(\mathbf{p}_0, \mathbf{p}_t)$, which moves the cart from position $\mathbf{p}_0$ to $\mathbf{p}_t$. The pre-condition is $\mathbf{s}_{\mathsf{Cart}} = 0 \wedge \mathbf{p}_{\mathsf{Cart}} = \mathbf{p}_0$. The post-condition is $\mathbf{s}_{\mathsf{Cart}} = 0 \wedge \mathbf{p}_{\mathsf{Cart}} = \mathbf{p}_t$. The invariant can specify a bound on the velocity, e.g., $0 \leq \mathbf{s}_{\mathsf{Cart}} \leq v_{\max}$, that the cart moves in straight line between $\mathbf{p}_0$ and $\mathbf{p}_t$, etc.

We can also include external input. For instance, we may add an external variable $\mathbf{w}_{\mathsf{obj}}$ to represent the weight of any carried object, e.g., the arm attached on top. Then, the pre-condition of move may include an extra constraint $0 \leq \mathbf{w}_{\mathsf{obj}} \leq w_{\max}$ to say that the cart can only move if the weight of the payload is smaller than a given bound.

5.2 Motion Calculus with Guarded Choice

Before executing some motion, a process may need to test the state of the physical world and, according to the current state, decide what to do. Therefore, we extend the calculus with the ability for a process to test predicates over its *Var* as part of the $\mathsf{if} \cdot \mathsf{then} \cdot \mathsf{else} \cdot$. On the specification side, we also add predicates to the internal choice.

Let $\mathcal{P}$ range over predicates. The global and local motion session types are modified as follows:

- The branching type for global session types becomes $\mathsf{p} \to \mathsf{q} : \{[\mathcal{P}_i]\ell_i(S_i).G_i\}_{i\in I}$.
- The branching type for local session types becomes $\oplus\{[\mathcal{P}_i]\mathsf{q}!\ell_i(S_i).T_i\}_{i\in I}$.

To make sure the modified types can be projected and then used for typing they need to respect the following constraints. Assume that Var_{p} are the variables associated with the robot executing the role of p. (1) The choices are *local*, i.e., for $\mathsf{p} \to \mathsf{q} : \{[\mathcal{P}_i]\ell_i(S_i).G_i\}_{i\in I}$ we have that $\mathrm{fv}(\mathcal{P}_i) \subseteq Var_{\mathsf{p}}$ for all i in I. (2) The choices are *total*, i.e., for $\mathsf{p} \to \mathsf{q} : \{[\mathcal{P}_i]\ell_i(S_i).G_i\}_{i\in I}$ we have that $\bigvee_{i\in I}\mathcal{P}_i$ is valid. The local types have similar constraints.

The subtyping and typing relation are updated as follows:

[SUB-OUT]

$$\frac{\forall i \in I: \quad S_i \leq: S_i' \quad T_i \leqslant T_i' \quad \mathcal{P}_i \Rightarrow \mathcal{P}_i'}{\oplus\{[\mathcal{P}_i]\mathsf{p}!\ell_i(S_i).T_i\}_{i\in I} \leqslant \oplus\{[\mathcal{P}_i']\mathsf{p}!\ell_i(S_i').T_i'\}_{i\in I\cup J}}$$

The change in this rule is the addition of checking the implication $\mathcal{P}_i \Rightarrow \mathcal{P}_i'$ to make sure that if the pre-condition of a motion primitive relies on $\mathcal{P}_i'$, it still holds with $\mathcal{P}_i$. Notice that $\oplus\{[\mathcal{P}_i]\mathsf{p}!\ell_i(S_i).T_i\}_{i\in I}$ which can have more restricted predicates needs to be a valid local type and the guards still need to be total.

[T-CHOICE]

$$\frac{\Gamma \vdash \mathsf{e} : \mathsf{bool} \quad \exists k \in I \quad \mathsf{e} \Rightarrow \mathcal{P}_k \quad \Gamma \vdash P_1 : T_k \quad \Gamma \vdash P_2 : \oplus\{[e \vee \mathcal{P}_i]T_i\}_{i\in I\setminus\{k\}}}{\Gamma \vdash \mathsf{if}\ \mathsf{e}\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2 : \oplus\{[\mathcal{P}_i]T_i\}_{i\in I}}$$

Type checking the rules propagates the expression from if then else and matches it into a branch of the type. To deal with the else branch we modify the predicate in the remaining branches of the type. For the last else branch of a, possibly nested, if then else we need the following extra rule:

[T-CHOICE-FINAL]

$$\frac{\Gamma \vdash P : T}{\Gamma \vdash P : \oplus\{[\mathsf{true}]T\}}$$

▶ **Example 5.2.** Usually, for the propagation of tested expressions through the branches we modify the type. Let us make an example of how this works. Consider we have the following process $\mathsf{if}\ \mathsf{e}_1\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2$ which has the type $\oplus\{[\mathsf{e}_1]T_1, [\neg\mathsf{e}_1]T_2\}$. Assuming that $P_i : T_i$ for $i \in \{1, 2\}$ we can build the following derivation:

$$\frac{\mathsf{e}_1 \Rightarrow \mathsf{e}_1 \qquad \Gamma \vdash P_1 : T_1 \qquad \dfrac{\Gamma \vdash P_2 : T_2}{\Gamma \vdash P_2 : \oplus\{[\mathsf{e}_1 \vee \neg\mathsf{e}_1]T_2\}}}{\Gamma \vdash \mathsf{if}\ \mathsf{e}_1\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2 : \oplus\{[\mathsf{e}_1]T_1, [\neg\mathsf{e}_1]T_2\}}$$

With a bit of boolean algebra, we can show that $\mathsf{e}_1 \vee \neg\mathsf{e}_1 \Leftrightarrow \mathsf{true}$.

5.3 Existence of Joint Trajectories and Verification

The goal of the compatibility check is to make sure that abstract motion primitives specified in a global type can execute concurrently. This requires two checks. First, for motion primitives of different processes executed in parallel, we need to make sure that there exists a trajectory satisfying all the constraints of the motion primitives. Second, for motion primitives executed sequentially by the same process, we need to make sure that the post-condition of the first implies the pre-condition of the second motion primitive, taking into account the guards of choices in the middle.

To check that motion primitives executing in parallel have a joint trajectory, we use an assume-guarantee style of reasoning. When two processes are attached, one process relies on the invariants of the other's output (which can be an external input) to satisfy its own invariant and vice versa. We refer to standard methods [35, 4] for the details.

For the allowed trajectories, we need to also check the absence of collision. This means that once we have the constrains defining a joint trajectory ξ to check that for any two distinct processes p and q the property $rsrc_{\mathsf{p}}(\xi_{\mathsf{p}}) \cap rsrc_{\mathsf{q}}(\xi_{\mathsf{q}}) = \emptyset$.

▶ **Example 5.3.** In Example 3.4, the cart and the carrier are moving toward each other. They need to be close enough for the arm to grab the object but far enough to avoid colliding. We model the resources of the cart by a cylinder around the cart's position: $rsrc_{\mathsf{Cart}} = \{(x, y, z) || (x, y) - \mathbf{p}_{\mathsf{Cart}}| \leq r \wedge 0 \leq z \leq h\}$ where r is the "radius" of the cart and h its height. The carrier's resources are similar but with the appropriate radius and height r', h'. The cart and carrier does not collide if we can prove that $\forall t. \mid \xi_{\mathsf{Cart}}|_{\mathbf{p}_{\mathsf{Cart}}}(t) - \xi_{\mathsf{Carrier}}|_{\mathbf{p}_{\mathsf{Carrier}}}(t) \mid > r + r'$.

6 Evaluation

6.1 Implementation

We have implemented the system we describe on top of PGCD [4][1], a system for programming and verification of robotic systems. PGCD is build on top of the Robotic Operating System (ROS) [40], a software ecosystem for robots. The core of ROS is a publish-subscribe messaging system. PGCD uses ROS's messaging to implement its synchronous message-passing layer. On the verification side, PGCD uses a mix of model-checking (using Spin [22]) to deal with the message-passing structure, and symbolic reasoning (using SymPy [33]) and constraint solving (using dReal [16]) to reason about motion primitives.

We replace the global model-checking algorithm of PGCD with motion session calculus specifications but reuse PGCD's infrastructure to reason about the trajectories of motion primitives. Currently, our implementation uses a syntax for specifications closer to the state-machine form of session types [11] but without the parallel composition operator. This representation allows for more general guarded choice. $\mathsf{if} \cdot \mathsf{then} \cdot \mathsf{else} \cdot$ implicitly forces disjoint guards for the two branches. Our implementation allows overlapping guards. Algorithmically, since the types are represented in a form close to an automaton, the projection and merge operations are implemented using automata theoretic operation: morphism, minimisation, and checking determinism of the result.

The typing, including subtyping, is implemented by computing an alternating simulation [2] between programs and their respective local type. Intuitively, an alternative refinement relation check that a process implements its specification (subset of the behaviours) without restricting the other processes. For synchronous message passing programs, the subtyping relation for session type matches alternating refinement. We use this view on subtyping as the theory of alternating simulation [2] gives us an algorithm to compute this relation and, therefore, check subtyping.

[1] PGCD repository is `https://github.com/MPI-SWS/pgcd`. The code for this work is located in the `pgcd/nodes/verification/choreography` folder.

6.2 Experiments

For the evaluation, we take two existing PGCD programs and write global types in motion session calculus that describe the co-ordination in the program.

First, we describe our experimental setup, both for the hardware and for the software. Then, we explain the experiments. Finally, we report on the size of the specifications, and time to check the programs satisfy the specification.

Setup

We use three robots: a robotic arm and two carts, shown in Figure 4. The robots are built with a mix of off-the-self parts and 3D printed parts.

Arm. The arm is a modified BCN3D MOVEO,[2] where the upper arm section is shortened to make it lighter and easier to mount on the cart. The arm with its control electronics is mounted on top of the cart.

Cart. The cart is shown on Figure 4a. The control electronics and motors are situated below the wooden board. The cart is an omnidirectional driving platform. It uses omniwheels to get three degrees of freedom (two in translation, one in rotation) and can move between any two positions on a flat ground. The advantage of using such wheels is that all the three degrees of freedom are controllable and movement does not require complex planning. Due to the large power consumption of the arm mounted on top, this cart is powered by a tether.

Carrier. We call the second cart the carrier (Figure 4b) as we use it to carry the block that is grabbed by the arm. As the first cart, it is also omnidirectional (mecanum wheels).

All the three robots use stepper motors to move precisely. The robots do not have feedback on their position and keep track of their state using *dead reckoning*, i.e., they know their initial state and then they update their virtual state by counting the number of steps the motors turns. If we control slippage and do not exceed the maximum torque of the motors, there is little accumulation of error as long as the initial state is known accurately. In our experiments, we use markings on the ground to fix the initial state as can be seen in Figure 5. Furthermore, using stepper motors allows us to know the time it takes to execute a given motion primitive by fixing the rate of steps.

Each robot has a RaspberryPi 3 model B to run the program. The ROS master node, providing core messaging services, runs on a separate laptop to which all the robots connect. The RaspberryPi runs Raspbian OS (based on Debian Jessie) and the laptop runs Ubuntu 16.04. The ROS version is Kinetic Kame.

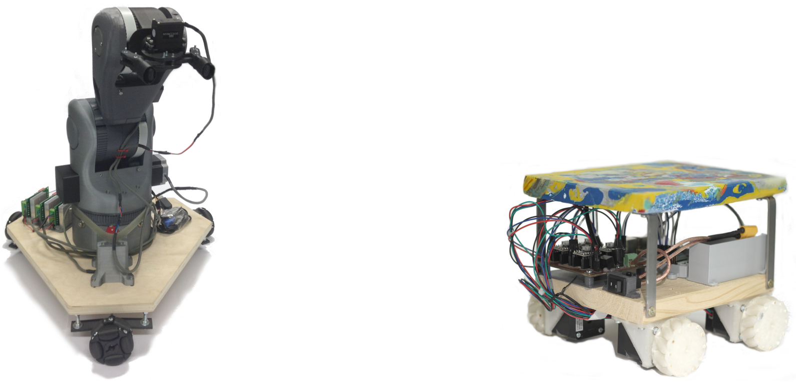
Experiments

We describe two experiments:

Handover. This experiment corresponds to our earlier example. The two carts meet before the arm takes an object placed on top of the carrier and, then, they go back to their initial position (see Figure 5a).

Underpass. First, the carrier cart brings an object to the arm which is then taken by the arm. Then, the carrier cart goes around the arm passing under an obstacle which is high enough for just the carrier alone. Finally, the arm puts the object back on the carrier on the other side of the obstacle. This can be seen in Figure 5b.

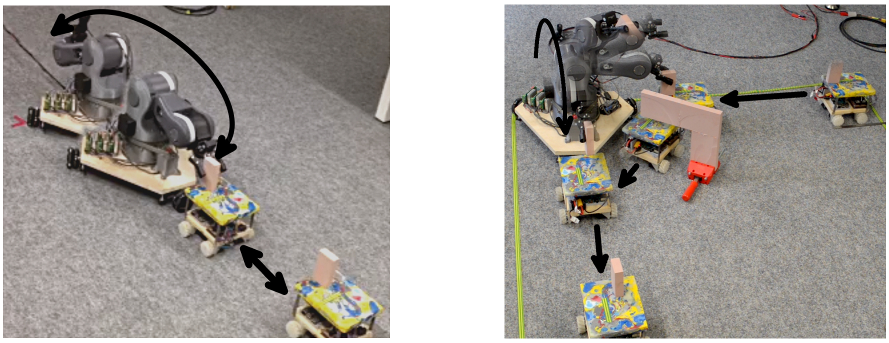
[2] `https://github.com/BCN3D/BCN3D-Moveo`



(a) The cart and arm robots attached together.

(b) The carrier robot.

Figure 4 Robots used in our experiments.



(a) Handover.

(b) Underpass.

Figure 5 Composite images of the experiments. (a) For handover, a cart containing an object moves close to the cart with the attached arm. The arm picks up the object. (b) For underpass, the carrier containing an object moves near the underpass. The arm picks up the object. The carrier moves under the underpass and moves close to the arm. The arm places the object on the carrier.

Composite images (combination of multiple frame of the video) are shown in Figure 5. The carts implement motion explicitly using the motion primitives (move straight, strafe, rotate). For instance, when going around the cart in the second experiment, the carrier executes rotate, move straight, rotate, strafe. In the model of the resources, we exclude the gripper from the footprint and we do not model the objects gripped (gripping is a collision). For the environment, we model obstacles as regions of $\mathbb{R}^3$ and also test for collision against these regions.

Table 3 shows the size of the programs in the language of PGCD (sum for all the robots) and the size of the global specifications. As part of the program we include a description of the environment which specifies the initial states of the robots and the obstacles used for additional collision checks. Finally, we show the number of verification conditions (#VCs) generated during the subtyping and the checks for joint trajectories. The total running time includes all the steps, i.e., checking the global specification, projection, typing, the existence of joint trajectories, and the absence of collision. The running time is dominated by the check on trajectories and collisions. The motion primitives (implementation and specification) are taken from PGCD without any change and represent around 1K lines of codes for all three robots.

Table 3 Programs, Specification, and Checks.

Experiment	Program	Specification	#VCs	Time
	(LoC)	(LoC)		(sec.)
Handover	22	12	141	38
Underpass	29	22	302	56

Compared to the verification results presented with PGCD [4, Section 5], we have roughly a 2× speed-up. The reason is that PGCD is used model-checking instead of global/local types. The motion session calculus makes it possible to have an abstract global specification which is easier to check.

In conclusion, our evaluation demonstrates that session types allow the specification of non-trivial co-ordination tasks between multiple robots with reasonable effort, while allowing automated and compositional verification.

7 Related Work

There is considerable interest in the robotics community on designing modular robotic components from higher-level specifications [32, 19]. However, most of this work has focused on descriptions for the physical and electronic design of components or on generating plans from higher level specifications rather than on language abstractions and types to reason about concurrency and motion. The interaction between concurrency and dynamics, and the use of automated verification techniques were considered in PGCD [4]. Our work takes PGCD as a starting point and formalises a compositional verification methodology through session types.

At the specification level, hybrid process algebras and other models of hybrid systems [1, 41, 7, 38] can model concurrent hybrid systems. However, these papers do not provide a direct path to implementation. Hybrid extensions to synchronous reactive languages [6, 5] describe programs which interact through events and control physical variables. Most existing verification methodologies for these programs rely on global model checking rather than on types. Our choice of session types is inspired by efficient type checking but also as the basis for describing interface specifications for components.

Extensions and applications of multiparty session types have been proposed in many different settings. See, e.g. [27, 3, 17]. We discuss only most related work. The work [9] extends multiparty session types with time, to enable the verification of realtime distributed systems. This extension with time allows specifications to express properties on the causalities of interactions, on the carried data types, and on the times in which interactions occur. The projected local types correspond to Communicating Timed Automata (CTA). To ensure the progress and liveness properties for projected local types, the framework requires several additional constraints on the shape of global protocols, such as feasibility condition (at any point of the protocol the current time constraint should be satisfiable for any possible past) and a limitation to the recursion where in the loop, the clock should be always reset. The approach is implemented in Python in [34] for runtime monitoring for the distributed system. Later, the work in [8] develops more relaxed conditions in CTAs, and applies them to synthesise timed global protocols. Unlike our work, no type checking for processes is studied in [8]. The main difference from [9, 34, 8] is that our approach does not rely on CTAs and is more specific to robotics applications where the verification is divided into the two layers; (1)

a simple type check for processes with motion primitives to ensure communication deadlock-freedom with global synchronisations; and (2) additional more refined checks for trajectories and resources. This two layered approach considerably simplifies our core calculus and typing system in Section 4, allowing to verify more complex scenarios for robotics interactions.

8 Conclusion

We have outlined a unifying programming model and typing discipline for communication-centric systems that sense and actuate the physical world. We work in the framework of multiparty session types [25, 26], which have proved their worth in many different scenarios relating to "pure" concurrent software systems. We show how to integrate motion primitives into a core calculus and into session types. We demonstrate how multiparty session types are used to specify correct synchronisation among multiple participants: we first provide a basic progress guarantee for communications and synchronisation by motion primitives, which is useful to extend richer verification related to trajectories.

At this point, our language is a starting point and not a panacea for robotics programming. Decoupling specifications into parallel and/or sequential tasks and using distributed controllers assumes "loosely coupled dynamics." In some examples, such as a multiple cart/arm co-ordination control, it may not be easy to assume a purely distributed control strategy based on independent motion primitives. We are thus exploring simultaneous concurrent programming and distributed controller and co-ordinator synthesis. As an example, assume that we have two cart/arm compositions which should lift one object together. In particular we can assume that lifting the object with only one arm would cause the cart/arm compositions to tilt over, which generates a strong coupling between all components during the coordinated lift of the object. Our framework allows to easily synchronise all the components. However, in any realistic scenario a robust controller would need (almost) continuous feedback between all components to fulfill the coordinated lift task. Thus, our model of loosely coupled motion primitives, one per component, may be too weak or incur too much communication and bandwidth overhead for a real implementation.

Going in this direction, we need a better way to integrate specifications of controllers (motion primitives) and their robustness. This would also enable a more realistic non-synchronous model for the communication [31] and, after checking some robustness condition on the controller, rigorously show that the synchronous idealised model is equivalent to the more realistic model, i.e., considering delay in the communication as disturbances for the motion primitives. We also plan to tackle channel passing. The challenge is that the physical world (time and space) is hard to isolate: for instance, time is an implicit synchronisation which occurs at the same time across all sessions.

Finally, robotics applications manipulate physical state and time as *resources*. An interesting open question is how resource-based reasoning techniques such as separation logics for concurrency [36, 28] can be repurposed to reason about separation of components in space and time.

References

1 R. Alur and T.A. Henzinger. Modularity for Timed and Hybrid Systems. In *CONCUR '97: Concurrency Theory*, volume 1243 of *LNCS*, pages 74–88. Springer, 1997.

2 R. Alur, T.A. Henzinger, O. Kupferman, and M.Y. Vardi. Alternating refinement relations. In *CONCUR'98 Concurrency Theory*, pages 163–178. Springer, 1998.

3 Davide Ancona, Viviana Bono, Mario Bravetti, Joana Campos, Giuseppe Castagna, Pierre-Malo Denielou, 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. *FTPL*, 3(2-3):95–230, 2016.

4 Gregor B. Banusic, Rupak Majumdar, Marcus Pirron, Anne-Kathrin Schmuck, and Damien Zufferey. PGCD: robot programming and verification with geometry, concurrency, and dynamics. In Xue Liu, Paulo Tabuada, Miroslav Pajic, and Linda Bushnell, editors, *Proceedings of the 10th ACM/IEEE International Conference on Cyber-Physical Systems, ICCPS 2019, Montreal, QC, Canada, April 16-18, 2019*, pages 57–66. ACM, 2019. doi:10.1145/3302509.3311052.

5 Kerstin Bauer and Klaus Schneider. From synchronous programs to symbolic representations of hybrid systems. In *Proceedings of the 13th ACM International Conference on Hybrid Systems: Computation and Control, HSCC 2010, Stockholm, Sweden, April 12-15, 2010*, pages 41–50. ACM, 2010. doi:10.1145/1755952.1755960.

6 Albert Benveniste, Timothy Bourke, Benoît Caillaud, Jean-Louis Colaço, Cédric Pasteur, and Marc Pouzet. Building a Hybrid Systems Modeler on Synchronous Languages Principles. *Proceedings of the IEEE*, 106(9):1568–1592, 2018. doi:10.1109/JPROC.2018.2858016.

7 J.A. Bergstra and C.A. Middelburg. Process algebra for hybrid systems. *Theoretical Computer Science*, 335(2):215–280, 2005. Process Algebra. doi:10.1016/j.tcs.2004.04.019.

8 Laura Bocchi, Julien Lange, and Nobuko Yoshida. Meeting Deadlines Together. In *26th International Conference on Concurrency Theory*, volume 42 of *LIPIcs*, pages 283–296. Schloss Dagstuhl, 2015.

9 Laura Bocchi, Weizhen Yang, and Nobuko Yoshida. Timed Multiparty Session Types. In *25th International Conference on Concurrency Theory*, volume 8704 of *LNCS*, pages 419–434. Springer, 2014.

10 Mario Coppo, Mariangiola Dezani-Ciancaglini, Luca Padovani, and Nobuko Yoshida. A Gentle Introduction to Multiparty Asynchronous Session Types. In *SFM*, volume 9104 of *LNCS*, pages 146–178. Springer, 2015.

11 Pierre-Malo Deniélou and Nobuko Yoshida. Multiparty Session Types Meet Communicating Automata. In *ESOP 2012 - European Symposium on Programming*. Springer, 2012. doi:10.1007/978-3-642-28869-2_10.

12 Pierre-Malo Deniélou, Nobuko Yoshida, Andi Bejleri, and Raymond Hu. Parameterised Multiparty Session Types. *Logical Methods in Computer Science*, 8(4), 2012. doi:10.2168/LMCS-8(4:6)2012.

13 Pierre-Malo Deniélou, Nobuko Yoshida, Andi Bejleri, and Raymond Hu. Parameterised Multiparty Session Types. *Logical Methods in Computer Science*, 8(4), 2012. doi:10.2168/LMCS-8(4:6)2012.

14 Mariangiola Dezani-Ciancaglini, Silvia Ghilezan, Svetlana Jaksic, Jovanka Pantovic, and Nobuko Yoshida. Precise subtyping for synchronous multiparty sessions. In *PLACES*, volume 203 of *EPTCS*, pages 29–43, 2015. doi:10.4204/EPTCS.203.3.

15 Mariangiola Dezani-Ciancaglini, Silvia Ghilezan, Svetlana Jaksic, Jovanka Pantovic, and Nobuko Yoshida. Denotational and Operational Preciseness of Subtyping: A Roadmap. In *Theory and Practice of Formal Methods: Essays Dedicated to Frank de Boer on the Occasion of His 60th Birthday*, volume 9660 of *LNCS*, pages 155–172. Springer, 2016.

16 Sicun Gao, Soonho Kong, and Edmund M. Clarke. dReal: An SMT solver for nonlinear theories over the reals. In Maria Paola Bonacina, editor, *Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings*, volume 7898 of *Lecture Notes in Computer Science*, pages 208–214. Springer, 2013. doi:10.1007/978-3-642-38574-2_14.

17 Simon Gay and Antonio Ravera, editors. *Behavioural Types: from Theory to Tools*. River Publishers, 2017.

18 Silvia Ghilezan, Svetlana Jaksic, Jovanka Pantovic, Alceste Scalas, and Nobuko Yoshida. Precise subtyping for synchronous multiparty sessions. *J. Log. Algebr. Meth. Program.*, 104:127–173, 2019. doi:10.1016/j.jlamp.2018.12.002.

19 Sehoon Ha, Stelian Coros, Alexander Alspach, James M. Bern, Joohyung Kim, and Katsu Yamane. Computational Design of Robotic Devices From High-Level Motion Specifications. *IEEE Trans. Robotics*, 34(5):1240–1251, 2018. doi:10.1109/TRO.2018.2830419.

20 Thomas A. Henzinger. Sooner is Safer Than Later. *Inf. Process. Lett.*, 43(3):135–141, 1992. doi:10.1016/0020-0190(92)90005-G.

21 Thomas A. Henzinger. The Theory of Hybrid Automata. In *Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science, New Brunswick, New Jersey, USA, July 27-30, 1996*, pages 278–292. IEEE Computer Society, 1996. doi:10.1109/LICS.1996.561342.

22 G.J. Holzmann. The Model Checker SPIN. *IEEE Trans. Software Eng.*, 23(5):279–295, 1997. doi:10.1109/32.588521.

23 Kohei Honda. Types for Dyadic Interaction. In *CONCUR'93*, pages 509–523, 1993.

24 Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language Primitives and Type Disciplines for Structured Communication-based Programming. In *ESOP*, volume 1381 of *LNCS*, pages 22–138. Springer, 1998. doi:10.1007/BFb0053567.

25 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty Asynchronous Session Types. In *POPL*, pages 273–284. ACM Press, 2008.

26 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty Asynchronous Session Types. *Journal of ACM*, 63:1–67, 2016.

27 Hans Hüttel, Ivan Lanese, Vasco T. Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Deniélou, Dimitris Mostrous, Luca Padovani, António Ravara, Emilio Tuosto, Hugo Torres Vieira, and Gianluigi Zavattaro. Foundations of Session Types and Behavioural Contracts. *ACM Comput. Surv.*, 49(1), 2016. doi:10.1145/2873052.

28 R. Jung, D. Swasey, F. Sieczkowski, K. Svendsen, A. Turon, L. Birkedal, and D. Dreyer. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *POPL 15*, pages 637–650. ACM, 2015.

29 Dimitrios Kouzapas and Nobuko Yoshida. Globally Governed Session Semantics. In Pedro R. D'Argenio and Hernán C. Melgratti, editors, *CONCUR*, volume 8052 of *LNCS*, pages 395–409. Springer, 2013.

30 Dimitrios Kouzapas and Nobuko Yoshida. Globally Governed Session Semantics. *Logical Methods in Computer Science*, 10(4), 2015.

31 Julien Lange, Emilio Tuosto, and Nobuko Yoshida. From communicating machines to graphical choreographies. In *42nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 221–232. ACM, 2015.

32 A.M. Mehta, N. Bezzo, P. Gebhard, B. An, V. Kumar, I. Lee, and D. Rus. A Design Environment for the Rapid Specification and Fabrication of Printable Robots. *Experimental Robotics*, pages 435–449, 2015.

33 Aaron Meurer, Christopher P. Smith, Mateusz Paprocki, Ondřej Čertík, Sergey B. Kirpichev, Matthew Rocklin, AMiT Kumar, Sergiu Ivanov, Jason K. Moore, Sartaj Singh, Thilina Rathnayake, Sean Vig, Brian E. Granger, Richard P. Muller, Francesco Bonazzi, Harsh Gupta, Shivam Vats, Fredrik Johansson, Fabian Pedregosa, Matthew J. Curry, Andy R. Terrel, Štěpán Roučka, Ashutosh Saboo, Isuru Fernando, Sumith Kulal, Robert Cimrman, and Anthony Scopatz. SymPy: symbolic computing in Python. *PeerJ Computer Science*, 3:e103, January 2017. doi:10.7717/peerj-cs.103.

34 Rumyana Neykova, Laura Bocchi, and Nobuko Yoshida. Timed runtime monitoring for multiparty conversations. *Formal Asp. Comput.*, 29(5):877–910, 2017.

35 Pierluigi Nuzzo. *Compositional Design of Cyber-Physical Systems Using Contracts*. PhD thesis, EECS Department, University of California, Berkeley, August 2015. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-189.html.

36 P.W. O'Hearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, 2007. doi:10.1016/j.tcs.2006.12.035.

37 Benjamin C. Pierce. *Types and Programming Languages.* MIT Press, 2002.

38 A. Platzer. *Logical Analysis of Hybrid Systems - Proving Theorems for Complex Dynamics.* Springer, 2010.

39 K. V. S. Prasad. A Calculus of Broadcasting Systems. *Sci. Comput. Program.*, 25(2-3):285–327, 1995. doi:10.1016/0167-6423(95)00017-8.

40 Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. ROS: an open-source Robot Operating System. In *ICRA workshop on open source software*, 2009.

41 W.C. Rounds and H. Song. The Phi-Calculus: A Language for Distributed Control of Reconfigurable Embedded Systems. In *HSCC*, pages 435–449. Springer, 2003.

42 Alceste Scalas and Nobuko Yoshida. Less is More: Multiparty Session Types Revisited. *Proc. ACM Program. Lang.*, 3(POPL):30:1–30:29, January 2019. doi:10.1145/3290343.

43 Kaku Takeuchi, Kohei Honda, and Makoto Kubo. An Interaction-based Language and its Typing System. In *PARLE'94*, volume 817 of *LNCS*, pages 398–413, 1994. doi:10.1007/3-540-58184-7_118.