

15th International Conference on Interactive Theorem Proving

ITP 2024, September 9–14, 2024, Tbilisi, Georgia

Edited by

Yves Bertot

Temur Kutsia

Michael Norrish



Editors

Yves Bertot 

Inria Research Center at University Côte d'Azur, France
Yves.Bertot@inria.fr

Temur Kutsia 

Johannes Kepler University, Linz, Austria
kutsia@risc.jku.at

Michael Norrish 

Australian National University, Canberra, Australia
Michael.Norrish@anu.edu.au

ACM Classification 2012

Theory of computation → Interactive proof systems; Theory of computation → Higher order logic; Theory of computation → Type theory; Theory of computation → Program verification; Theory of computation → Logic and verification

ISBN 978-3-95977-337-9

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/978-3-95977-337-9>.

Publication date

September, 2024

Bibliographic information published by the Deutsche Nationalbibliothek

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

License

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



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

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

The copyright is retained by the corresponding authors.

Digital Object Identifier: 10.4230/LIPIcs.ITP.2024.0

ISBN 978-3-95977-337-9

ISSN 1868-8969

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

LIPICs – Leibniz International Proceedings in Informatics

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

Editorial Board

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

ISSN 1868-8969

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

■ Contents

Preface	
<i>Yves Bertot, Temur Kutsia, and Michael Norrish</i>	0:ix
Organization	
.....	0:xi

Invited Talks


Alpha-Beta Pruning Verified	
<i>Tobias Nipkow</i>	1:1–1:4
Translating Libraries of Definitions and Theorems Between Proof Systems	
<i>Frédéric Blanqui</i>	2:1–2:1

Regular Papers

A Formal Analysis of Capacity Scaling Algorithms for Minimum Cost Flows	
<i>Mohammad Abdulaziz and Thomas Ammer</i>	3:1–3:19
Taming Differentiable Logics with Coq Formalisation	
<i>Reynald Affeldt, Alessandro Bruni, Ekaterina Komendantskaya, Natalia Ślusarz, and Kathrin Stark</i>	4:1–4:19
A Comprehensive Overview of the Lebesgue Differentiation Theorem in Coq	
<i>Reynald Affeldt and Zachary Stone</i>	5:1–5:19
Towards Solid Abelian Groups: A Formal Proof of Nöbeling’s Theorem	
<i>Dagur Asgeirsson</i>	6:1–6:17
An Operational Semantics in Isabelle/HOL-CSP	
<i>Benoît Ballenghien and Burkhart Wolff</i>	7:1–7:18
The Directed Van Kampen Theorem in Lean	
<i>Henning Basold, Peter Bruin, and Dominique Lawson</i>	8:1–8:18
Verifying Peephole Rewriting in SSA Compiler IRs	
<i>Siddharth Bhat, Alex Keizer, Chris Hughes, Andrés Goens, and Tobias Grosser</i> ..	9:1–9:20
Duper: A Proof-Producing Superposition Theorem Prover for Dependent Type Theory	
<i>Joshua Clune, Yicheng Qian, Alexander Bentkamp, and Jeremy Avigad</i>	10:1–10:20
A Formalization of the General Theory of Quaternions	
<i>Thaynara Arielly de Lima, André Luiz Galdino, Bruno Berto de Oliveira Ribeiro, and Mauricio Ayala-Rincón</i>	11:1–11:18
A Modular Formalization of Superposition in Isabelle/HOL	
<i>Martin Desharnais, Balazs Toth, Uwe Waldmann, Jasmin Blanchette, and Sophie Tourret</i>	12:1–12:20

15th International Conference on Interactive Theorem Proving (ITP 2024).

Editors: Yves Bertot, Temur Kutsia, and Michael Norrish

 Leibniz International Proceedings in Informatics

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

Completeness of Asynchronous Session Tree Subtyping in Coq <i>Burak Ekici and Nobuko Yoshida</i>	13:1–13:20
End-To-End Formal Verification of a Fast and Accurate Floating-Point Approximation <i>Florian Faissole, Paul Geneau de Lamarlière, and Guillaume Melquiond</i>	14:1–14:18
Typed Compositional Quantum Computation with Lenses <i>Jacques Garrigue and Takafumi Saikawa</i>	15:1–15:18
A Formal Proof of $R(4,5)=25$ <i>Thibault Gauthier and Chad E. Brown</i>	16:1–16:18
Verifying Software Emulation of an Unsupported Hardware Instruction <i>Samuel Gruetter, Thomas Bourgeat, and Adam Chlipala</i>	17:1–17:16
Mechanized HOL Reasoning in Set Theory <i>Simon Guilloud, Sankalp Gambhir, Andrea Gilot, and Viktor Kunčák</i>	18:1–18:18
Modular Verification of Intrusive List and Tree Data Structures in Separation Logic <i>Marc Hermes and Robbert Krebbers</i>	19:1–19:18
Formalizing the Algebraic Small Object Argument in UniMath <i>Dennis Hilhorst and Paige Randall North</i>	20:1–20:18
A Formalization of the Lévy-Prokhorov Metric in Isabelle/HOL <i>Michikazu Hirata</i>	21:1–21:18
Distributed Parallel Build for the Isabelle Archive of Formal Proofs <i>Fabian Huch and Makarius Wenzel</i>	22:1–22:19
A Generalised Union of Rely–Guarantee and Separation Logic Using Permission Algebras <i>Vincent Jackson, Toby Murray, and Christine Rizkallah</i>	23:1–23:16
An Isabelle/HOL Formalization of Narrowing and Multiset Narrowing for E -Unifiability, Reachability and Infeasibility <i>Dohan Kim</i>	24:1–24:19
Formalizing the Cholesky Factorization Theorem <i>Carl Kwan and Warren A. Hunt Jr.</i>	25:1–25:16
The Rewster: Type Preserving Rewrite Rules for the Coq Proof Assistant <i>Yann Leray, Gaëtan Gilbert, Nicolas Tabareau, and Théo Winterhalter</i>	26:1–26:18
Teaching Mathematics Using Lean and Controlled Natural Language <i>Patrick Massot</i>	27:1–27:19
Lean Formalization of Completeness Proof for Coalition Logic with Common Knowledge <i>Kai Obendrauf, Anne Baanen, Patrick Koopmann, and Vera Stebletsova</i>	28:1–28:18
Conway Normal Form: Bridging Approaches for Comprehensive Formalization of Surreal Numbers <i>Karol Pqk and Cezary Kaliszzyk</i>	29:1–29:18

A Coq Formalization of Taylor Models and Power Series for Solving Ordinary Differential Equations <i>Sewon Park and Holger Thies</i>	30:1–30:19
A Verified Earley Parser <i>Martin Rau and Tobias Nipkow</i>	31:1–31:18
Abstractions for Multi-Sorted Substitutions <i>Hannes Saffrich</i>	32:1–32:19
Correctly Compiling Proofs About Programs Without Proving Compilers Correct <i>Audrey Seo, Christopher Lam, Dan Grossman, and Talia Ringer</i>	33:1–33:20
Redex2Coq: Towards a Theory of Decidability of Redex’s Reduction Semantics <i>Mallku Soldevila, Rodrigo Ribeiro, and Beta Ziliani</i>	34:1–34:18
Formal Verification of the Empty Hexagon Number <i>Bernardo Subercaseaux, Wojciech Nawrocki, James Gallicchio, Cayden Codel, Mario Carneiro, and Marijn J. H. Heule</i>	35:1–35:19
Defining and Preserving More C Behaviors: Verified Compilation Using a Concrete Memory Model <i>Andrew Tolmach, Chris Chhak, and Sean Anderson</i>	36:1–36:20
Integrals Within Integrals: A Formalization of the Gagliardo-Nirenberg-Sobolev Inequality <i>Floris van Doorn and Heather Macbeth</i>	37:1–37:18
The Functor of Points Approach to Schemes in Cubical Agda <i>Max Zeuner and Matthias Hutzler</i>	38:1–38:18
Short Papers	
Robust Mean Estimation by All Means <i>Reynald Affeldt, Clark Barrett, Alessandro Bruni, Ieva Daukantas, Harun Khan, Takafumi Saikawa, and Carsten Schürmann</i>	39:1–39:8
Formalising Half of a Graduate Textbook on Number Theory <i>Manuel Eberl, Anthony Bordg, Lawrence C. Paulson, and Wenda Li</i>	40:1–40:7
Graphical Rewriting for Diagrammatic Reasoning in Monoidal Categories in Lean4 <i>Sam Ezeh</i>	41:1–41:8

■ Preface

The International Conference on Interactive Theorem Proving (ITP) is the main venue for the presentation of research into interactive theorem proving frameworks and their applications. It has evolved organically starting with a HOL workshop in 1988, gradually widening to include other higher-order systems and interactive theorem provers generally, as well as their applications. This year’s conference takes place in Tbilisi in Georgia.

Previous ITP conferences took place in Edinburgh 2010, Nijmegen 2011, Princeton 2012, Rennes 2013, Vienna 2014, Nanjing 2015, Nancy 2016, Brasilia 2017, Oxford 2018, Portland 2019, Paris 2020, Rome 2021, Haifa 2022, and Białystok 2023; those in 2010, 2014, 2018 and 2022 were under the umbrella organization of the Federated Logic Conference (FLoC).

This year’s conference attracted a total of 71 submissions. Each paper was systematically reviewed by at least three program committee members or appointed external reviewers and 39 papers were finally selected for presentation at the conference (36 regular papers and 3 short papers). We thank the authors of both accepted and rejected papers for their submissions, as well as the programme committee members and the external reviewers for their invaluable work.

As well as all the selected papers, we are very pleased to have invited keynote talks by Frédéric Blanqui (Centre Inria de l’Université de Lorraine, Nancy) and Tobias Nipkow (Technische Universität München). The present volume collects all the accepted papers contributed to the conference and abstracts for the two invited talks. This is the fifth time that the ITP proceedings are published by the LIPIcs series. We thank all the colleagues at Dagstuhl for their responsive feedback on all matters associated with the production of the finished proceedings.

We are grateful to all of the local organizers and thankful to the ITP Steering Committee for their guidance. This conference received partial support from the European Union COST Action CA20111 “European Research Network on Formal Proofs”, and from Inria, France.

Yves Bertot, Temur Kutsia, and Michael Norrish



■ Organization

Local Organization Committee

Matthias Baaz	Vienna University of Technology, Austria
Besik Dundua	(chair) Kutaisi International University and Institute of Applied Mathematics, Tbilisi State University, Georgia
Mariam Gamsakhurdia	Vienna University of Technology, Austria
Mikheil Rukhaia	Institute of Applied Mathematics, Tbilisi State University, Georgia

Programme Chairs

Yves Bertot	Inria Research Center at University Côte d'Azur, France
Temur Kutsia	RISC, Johannes Kepler University Linz, Austria
Michael Norrish	Australian National University, Australia

Programme Committee

Mohammad Abdulaziz	Reynald Affeldt	Nada Amin
Mauricio Ayala-Rincón	Mario Carneiro	Pierre Courtieu
Catherine Dubois	Besik Dundua	Manuel Eberl
Yannick Forster	Ruben Gamboa	Ralf Jung
Hrutvik Kanabar	Gerwin Klein	Angeliki Koutsoukou Argyraki
Ramana Kumar	Robert Lewis	Marco Maggesi
Mariano Moscato	Magnus O. Myreen	Adam Naumowicz
Karl Palmkog	Andrei Popescu	Eric Smith
Hira Syeda	Yong Kiam Tan	René Thiemann
Dmitriy Traytel	Josef Urban	Christian Urban
Niccolò Veltri	Niels Voorneveld	Freek Wiedijk
Burkhart Wolff	Floris van Doorn	

External Reviewers

Rafael Castro Gonçalves Silva	Alessandro Coglio	Luís Cruz-Filipe
Sander Dahmen	Thaynara Arielly de Lima	Flavio L. C. De Moura
Aaron Dutle	Chelsea Edmonds	Mathias Fleury
John Harrison	Chung-Kil Hur	Jan Jakubuv
Philipp Joram	Grant Jurgensen	Cezary Kaliszyk
Emin Karayel	Andy King	Dominik Kirst
Peter Koepke	András Kovács	
Anastasiya Kravchuk-Kirilyuk	Katharina Kreuzer	Wenda Li
Mircea Marin	Bhavik Mehta	Chad Nester
Lawrence Paulson	Bartosz Piotrowski	Olivier Pons
Kazuhiko Sakaguchi	Rafaello Sanna	Julia Sapiña
Maximilian Schäffeler	Anders Schlichtkrull	Joseph Slagel
Martin Suda	Laurent Théry	Adam Topaz
Xavier Urbain	Lauren White	Théo Winterhalter
Junyan Xu	Liao Zhang	Beta Ziliani

15th International Conference on Interactive Theorem Proving (ITP 2024).

Editors: Yves Bertot, Temur Kutsia, and Michael Norrish



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

Alpha-Beta Pruning Verified

Tobias Nipkow 

Department of Computer Science, Technical University of Munich, Germany

Abstract

Alpha-beta pruning is an efficient search strategy for two-player game trees. It was invented in the late 1950s and is at the heart of most implementations of combinatorial game playing programs. We have formalized and verified a number of variations of alpha-beta pruning, in particular fail-hard and fail-soft, and valuations into linear orders, distributive lattices and domains with negative values.

2012 ACM Subject Classification Software and its engineering → Formal software verification; Theory of computation → Algorithmic game theory

Keywords and phrases Verification, Algorithmic Game Theory, Isabelle

Digital Object Identifier 10.4230/LIPIcs.ITP.2024.1

Category Invited Talk

1 Introduction

In this article, we sketch the results of our formalization [9] of a number of variants of alpha-beta pruning in the proof assistant Isabelle [12, 11]. A more detailed presentation can be found in a (forthcoming) book [10]. We restrict ourselves to functional correctness and do not cover quantitative results

We consider two-player games with the usual notion of game trees. Let $val(t)$ denote the *value* of a game tree t . Alpha-beta pruning is an efficient strategy for determining the value of a tree. In this extended abstract we assume that the reader is familiar with the basic idea underlying alpha-beta pruning. We do not show any code but all our variants f of alpha-beta pruning follow the calling convention $f\ a\ b\ t$ where (a, b) is the search window and t the game tree.

2 Linear Orders

In the literature it is often assumed that values are numbers extended with $-\infty$ and ∞ . In this section we merely assume that the type of values is a bounded linear order with \perp and \top .

The first thorough mathematical analysis of alpha-beta pruning is due to Knuth and Moore [6]. They employ the relation $x \cong y\ (a, b)$ defined as follows:

$$(y \leq a \longrightarrow x \leq a) \wedge (a < y < b \longrightarrow x = y) \wedge (y \geq b \longrightarrow x \geq b)$$

The notation $x \cong y\ (a, b)$ is ours and emphasizes the fact that the relation is symmetric if $a < b$. Knuth and Moore consider the so-called *fail-hard* variant of alpha-beta pruning, which we denote by $hard\ a\ b\ t$, and prove that $val(t) \cong hard\ a\ b\ t\ (a, b)$ which implies overall correctness: $hard\ \perp\ \top\ t = val(t)$.

Fishburn [3] suggested the *fail-soft* variant and states that it searches the same part of the tree as fail-hard and satisfies the relation $soft\ a\ b\ t \leq val(t)\ (a, b)$ (the notation is again ours) where $y \leq x\ (a, b)$ is defined as follows:

$$(y \leq a \longrightarrow x \leq y) \wedge (a < y < b \longrightarrow x = y) \wedge (y \geq b \longrightarrow x \geq y)$$



© Tobias Nipkow;

licensed under Creative Commons License CC-BY 4.0

15th International Conference on Interactive Theorem Proving (ITP 2024).

Editors: Yves Bertot, Temur Kutsia, and Michael Norrish; Article No. 1; pp. 1:1–1:4

Leibniz International Proceedings in Informatics



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

1:2 Alpha-Beta Pruning Verified

We verified both that the search space is the same and that $\text{soft } a \ b \ t \leq \text{val}(t) \ (a, b)$. Because $y \leq x \ (a, b)$ implies $x \cong y \ (a, b)$ (but not the other way around), it appears that *soft* satisfies a stronger property than *hard*. However, we could also prove $\text{hard } a \ b \ t \leq \text{val}(t) \ (a, b)$, which begs the question in what sense *soft* is better than *hard*. The answer: we could also prove $\text{hard } a \ b \ t \leq \text{soft } a \ b \ t \ (a, b)$. In summary: *soft* is as least as close to *val* as *hard* (and sometimes closer).

2.1 Negative Values

In the literature, it is often assumed that values are positive and negative numbers and that the value v for one player is $-v$ for the other player. In this situation the definition of *val* and of alpha-beta pruning can be streamlined: instead of having one function for each player, now one function in total is needed. We found that this works for arbitrary linear orders with unary negation under these two assumptions:

$$-\min x \ y = \max (-x)(-y) \quad -(-x) = x$$

3 Lattices

Bird and Hughes [1] were the first to generalize alpha-beta pruning from linear orders to lattices. The generalization of the code, including game tree evaluation, is easy: simply replace *min* and *max* by \sqcap and \sqcup . It turns out that this version of alpha-beta pruning works for bounded distributive lattices (Bird and Hughes confusingly talk about Boolean algebra). In order to prove this, Bird and Hughes invent the following relation (the notation \simeq is ours)

$$x \simeq y \ (a, b) \iff a \sqcup (x \sqcap b) = a \sqcup (y \sqcap b)$$

and prove

$$bh \ a \ b \ t \simeq \text{val}(t) \ (a, b)$$

where *bh* is their variation of fail-hard. Top-level correctness $bh \ \perp \ \top \ t = \text{val}(t)$ follows immediately.

For linear orders (but not for distributive lattices) \cong and \simeq coincide if $a < b$:

$$x \cong y \ (a, b) \iff x \simeq y \ (a, b)$$

It is also possible to rephrase Fishburn's predicate \leq with *min* and *max* (for linear orders) if $a < b$:

$$y \leq x \ (a, b) \iff \min x \ b \leq y \leq \max x \ a$$

We rephrase the r.h.s. for distributive lattices and define

$$y \sqsubseteq x \ (a, b) \iff x \sqcap b \leq y \leq x \sqcup a$$

It coincides with \leq for linear orders (but not for distributive lattices) if $a < b$

$$y \leq x \ (a, b) \iff y \sqsubseteq x \ (a, b)$$

In distributive lattices \sqsubseteq implies \simeq

$$y \sqsubseteq x \ (a, b) \implies x \simeq y \ (a, b)$$

but the opposite direction does not hold.

Fail-hard and fail-soft carry over to distributive lattices (*mutadis mutandis*) and satisfy the stronger \sqsubseteq

$$f \ a \ b \ t \sqsubseteq \text{val}(t) \ (a, b)$$

where f can be fail-hard, fail-soft, and *bh* (Bird and Hughes' version),

3.1 Negative Values

The same extension to negative values that we sketched in Section 2.1 also works for distributive lattices. Now we assume that unary negation satisfies

$$-(x \sqcap y) = (-x) \sqcup (-y) \quad -(-x) = x$$

The result is a so-called *de Morgan algebra* [8]. Again, the definitions of *val* and the variants of alpha-beta pruning can be streamlined as sketched before.

4 Related Work

Ginsburg and Jaffray [4] rediscover (independently of Bird and Hughes) that pruning also works for distributive lattices but stop short of formulating and proving an actual algorithm. Li *et al.* [7] build on the work of Ginsburg and Jaffray, formulate and prove an actual algorithm correct (rediscovering the correctness notion \simeq by Bird and Hughes) and extend the algorithm to also cache and reuse earlier calls, an aspect not covered above.

In our formalization of variants of alpha-beta pruning we also considered the program that Hughes (in a much cited paper [5]) derived from a specification of *val*. Because his program looks simpler than alpha-beta pruning, we suspected that it may not prune enough. Hughes (private communication) used Haskell's Quickcheck [2] to compare the search space of his program with that of our implementation (which is the canonical alpha-beta algorithm) and confirmed our suspicion.

5 Conclusion

We have formally verified a number of variants of alpha-beta pruning, both for linear orders and distributive lattices, have clarified the relationship between different correctness notions, have expressed precisely (and proved) in what sense fail-soft is better than fail-hard, and discovered a “suboptimal” version of alpha-beta pruning in a much cited paper.

References

- 1 Richard S. Bird and John Hughes. The alpha-beta algorithm: An exercise in program transformation. *Information Processing Letters*, 24(1):53–57, 1987. doi:10.1016/0020-0190(87)90198-0.
- 2 Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of Haskell programs. In Martin Odersky and Philip Wadler, editors, *International Conference on Functional Programming (ICFP '00)*, pages 268–279. ACM, 2000. doi:10.1145/351240.351266.
- 3 John P. Fishburn. Another optimization of alpha-beta search. *SIGART Newsl.*, 84:37–38, 1983. doi:10.1145/1056623.1056628.
- 4 Matthew L. Ginsberg and Alan Jaffray. Alpha-beta pruning under partial orders. In Richard J. Nowakowski, editor, *More Games of No Chance*, volume 42 of *MSRI Publications*, pages 37–48. Cambridge University Press, 2002. URL: <http://library.msri.org/books/Book42/files/ginsberg.pdf>.
- 5 J. Hughes. Why Functional Programming Matters. *The Computer Journal*, 32(2):98–107, 1989. doi:10.1093/comjnl/32.2.98.
- 6 Donald E. Knuth and Ronald W. Moore. An analysis of alpha-beta pruning. *Artif. Intell.*, 6(4):293–326, 1975. doi:10.1016/0004-3702(75)90019-3.

- 7 Junkang Li, Bruno Zanuttini, Tristan Cazenave, and Véronique Ventos. Generalisation of alpha-beta search for AND-OR graphs with partially ordered values. In Luc De Raedt, editor, *International Joint Conference on Artificial Intelligence, IJCAI 2022*, pages 4769–4775. ijcai.org, 2022. doi:10.24963/ijcai.2022/661.
- 8 Gr. C. Moisil. Recherches sur l’algèbre de la logique. *Annales scientifiques de l’Université de Jassy*, 122:1–118, 1936.
- 9 Tobias Nipkow. Alpha-beta pruning. *Archive of Formal Proofs*, June 2024. , Formal proof development. URL: https://isa-afp.org/entries/Alpha_Beta_Pruning.html.
- 10 Tobias Nipkow, editor. *Functional Data Structures and Algorithms. A Proof Assistant Approach*. ACM Books. self-published, 2024. URL: <https://functional-algorithms-verified.org/>.
- 11 Tobias Nipkow and Gerwin Klein. *Concrete Semantics with Isabelle/HOL*. Springer, 2014. URL: <http://concrete-semantics.org>.
- 12 Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

Translating Libraries of Definitions and Theorems Between Proof Systems

Frédéric Blanqui  

INRIA, Gif-sur-Yvette, France

Abstract

There exist many proof systems, interactive or automated. However, most of them are not interoperable, which leads to an important work duplication. This is unfortunate as it slows down the formalization of more advanced mathematical results, and the democratization of proof systems in education, industry and research. This state of affairs is not just a matter of file formats. Each proof system has its own axioms and deduction rules, and those axioms and deduction rules can sometimes be incompatible. To translate a proof from one system to the other, and be able to handle so many different systems, it is important to find out a logical framework in which a logical feature used in two different systems is represented by the same construction.

Research on proof system interoperability started about 30 years ago, and received some increased attention with the formalization of Hales proof of Kepler conjecture in the years 2000, because parts of this proof were initially formalized in different systems. Then, it received some new interest in the years 2010 with the increasing use of automated theorem provers in proof assistants. At about the same time appeared a new logical framework, Dedukti, which extends Edinburgh's logical framework LF by allowing the identification of types modulo some equational theory. It has been shown that various proof systems can be nicely encoded in Dedukti, and various tools have been developed to actually represent the proofs of those systems and translate them to other systems.

In this talk, I will review some of these works and tools, and present recent efforts to translate entire libraries of definitions and theorems.

2012 ACM Subject Classification Theory of computation → Logic and verification

Keywords and phrases Logical frameworks, proof systems interoperability, type theory

Digital Object Identifier 10.4230/LIPIcs.ITP.2024.2

Category Invited Talk

Acknowledgements Based upon work from the EuroProofNet action CA20111, supported by COST (European Cooperation in Science and Technology).



© Frédéric Blanqui;

licensed under Creative Commons License CC-BY 4.0

15th International Conference on Interactive Theorem Proving (ITP 2024).

Editors: Yves Bertot, Temur Kutsia, and Michael Norrish; Article No. 2; pp. 2:1–2:1

Leibniz International Proceedings in Informatics




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

A Formal Analysis of Capacity Scaling Algorithms for Minimum Cost Flows

Mohammad Abdulaziz ✉ 🏠 

Department of Informatics, King's College London, UK

Thomas Ammer ✉ 🏠 

Department of Informatics, King's College London, UK

Abstract

We present a formalisation of the correctness of algorithms to solve minimum-cost flow problems, in Isabelle/HOL. Two of the algorithms are based on the technique of scaling, most notably Orlin's algorithm, which has the fastest running time for the problem of minimum-cost flow. Our work uncovered a number of complications in the proofs of the results we formalised, the resolution of which required significant effort. Our work is also the first to formally consider the problem of minimum-cost flows and, more generally, scaling algorithms.

2012 ACM Subject Classification Theory of computation → Network flows; Theory of computation → Invariants; Theory of computation → Program verification

Keywords and phrases Network Flows, Formal Verification, Combinatorial Optimisation

Digital Object Identifier 10.4230/LIPIcs.ITP.2024.3

Funding The authors received funding from the Hausdorff Research Institute for Mathematics as part of the Trimester Programme “Prospects for Formal Mathematics”. The second author received funding from the German foundation Cusanuswerk and is currently supported by a studentship from the Department of Informatics at King's College London.

Acknowledgements Parts of the work presented in this paper were done at the Technical University of Munich.

1 Introduction

Flow networks are some of the most important structures in combinatorial optimisation and computer science. In addition to many immediate practical applications, flow networks and problems defined on them have many connections to other important problems in computer science, most notably, the connection between maximum weight bipartite matching and the problem of maximum flow. Because of this practical and theoretical relevance, network flows have been intensely studied, leading to many important milestone results in computer science, like the Edmonds-Karp algorithm [7] for computing the maximum flow between two vertices in a network. Furthermore, flow algorithms were some of the earliest algorithms to be considered for formal analysis. The first such effort was in 2005 by Lee in the prover Mizar [19], where the Ford-Fulkerson algorithm for maximum flow was verified. Later on, Lammich and Serfidgar [17] formally analysed the same algorithm and also the Edmonds-Karp algorithm [7], which is one of its polynomial worst-case running time refinements, in Isabelle/HOL.

In this work we formalise in Isabelle/HOL the correctness of a number of algorithms for the *minimum-cost flow problem*, which is another important computational problem defined on flow networks. Given a flow network, costs per unit flow associated with every edge, and a desired flow value between a number of sources and a number of sinks, a solution to this problem is a flow achieving that value, but for the minimum-cost. This problem can be seen as a generalisation of maximum flow, and thus many problems can be reduced to it, e.g. shortest path, maximum flow, and maximum weight bipartite matching.



© Mohammad Abdulaziz and Thomas Ammer;
licensed under Creative Commons License CC-BY 4.0

15th International Conference on Interactive Theorem Proving (ITP 2024).

Editors: Yves Bertot, Temur Kutsia, and Michael Norrish; Article No. 3; pp. 3:1–3:19

Leibniz International Proceedings in Informatics



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

More specifically, we formalise 1. the problem of minimum-cost flows, 2. the main optimality criterion used to justify most algorithms for minimum-cost flow, and 3. the correctness of three algorithms to compute minimum-cost flows: a. successive shortest paths, which has an exponential worst-case running time, b. capacity scaling, which has a polynomial worst-case running time, and c. Orlin's algorithm, which has a strongly polynomial worst-case running time. A noteworthy outcome of our work is that it uncovered gaps in the correctness proof of Orlin's algorithm in most textbook expositions. For instance, an important property, namely, optimality preservation, has a gap in all combinatorial proofs of which we are aware. We cover that gap (Lemma 3) using an involved graphical argument, which was at least 15% of our effort. The presence of this gap and the complexity of proving Theorem 1 is yet another example of complications uncovered in graphical and geometric arguments when formalising them, something which was documented by prior authors [1, 24, 13, 3].

2 Background and Definitions

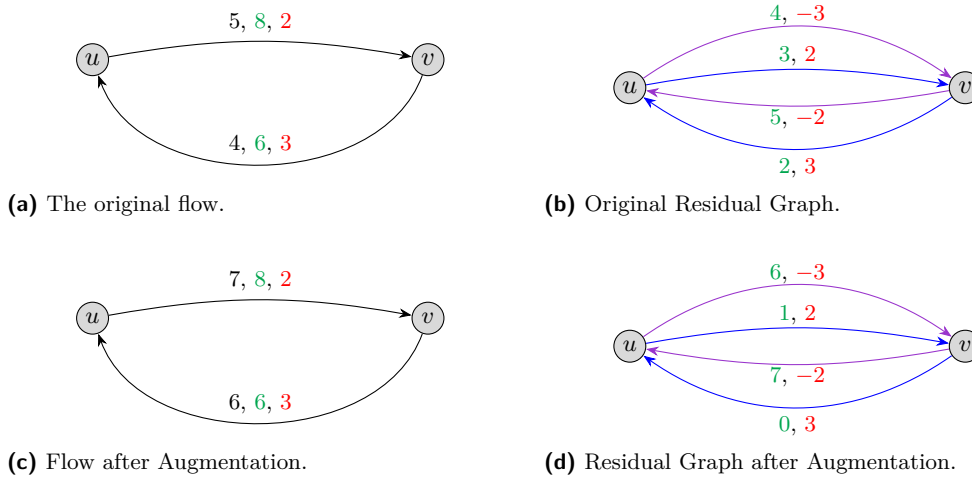
A directed graph is defined as a set of ordered pairs. A maximal set of vertices C , where there is a path between x and y or y and x for all $x, y \in C$, is a *connected component*. A *representative function* $r : \mathcal{V} \rightarrow \mathcal{V}$ maps all vertices within a component C to the same vertex $r_C \in C$. Consequently, we call $r(x)$ the representative of component C for a vertex $x \in C$.

A *flow network* consists of a directed graph over edges \mathcal{E} and vertices \mathcal{V} , and a capacity function $u : \mathcal{E} \rightarrow \mathbb{R}_0^+ \cup \{\infty\}$. If $u(e) = \infty$ for all $e \in \mathcal{E}$, the network is *uncapacitated*. The goal is to find a function, i.e. a *flow* $f : \mathcal{E} \rightarrow \mathbb{R}_0^+$ satisfying $f(e) \leq u(e)$ for any edge e . An edge is *saturated* if its flow $f(e)$ equals its capacity $u(e)$, otherwise the edge is *unsaturated*. The first vertex of an edge is called the *source* and the second one is the *target* of the edge, respectively. For a specific vertex v , the set of all edges entering or leaving this vertex is denoted by $\delta^-(v)$ or $\delta^+(v)$, respectively. The *excess* of a flow f at the vertex v , $\text{ex}_f(v)$, is the difference between ingoing and outgoing flow $\text{ex}_f(v) \stackrel{\text{def}}{=} \sum_{e \in \delta^-(v)} f(e) - \sum_{e \in \delta^+(v)} f(e)$.

Analogously to single vertices, the set of entering and leaving edges of a set of vertices X is denoted by $\Delta^+(X)$ and $\Delta^-(X)$, respectively. An ordered bipartition $(X, \mathcal{V} \setminus X)$ of the graph's vertices is called a *cut*. The (reverse) capacity of a cut X is the accumulated edge capacity of all (ingoing) outgoing edges $\text{cap}(X) \stackrel{\text{def}}{=} \sum_{e \in \Delta^+(X)} u(e)$ ($\text{acap}(X) \stackrel{\text{def}}{=} \sum_{e \in \Delta^-(X)} u(e)$).

A *minimum cost flow problem* consists of two further ingredients. We introduce *balances* $b : \mathcal{V} \rightarrow \mathbb{R}$ denoting the amount of flow that should be caught or emitted at every vertex. A flow satisfying balance and capacity constraints is called *valid*. In addition, there is a function $c : \mathcal{E} \rightarrow \mathbb{R}$ telling us about the *costs* of sending one unit of flow through an edge. A flow's *total costs* $c(f)$ are $c(f) = \sum_{e \in \mathcal{E}} f(e) \cdot c(e)$. The set of *feasible flows* is $\{f \mid \forall v \in \mathcal{V}. -\text{ex}_f(v) = b(v) \wedge \forall e \in \mathcal{E}. f(e) \leq u(e)\}$. We aim to find a *minimum cost flow* which is a feasible flow of least total costs. A network without cycles of negative total costs is called *weight-conservative*.

Given a flow f , we define the *residual network*: For any edge $(x, y) \in \mathcal{E}$ we have two *residual edges*, namely, the *forward* $F(x, y)$ and *backward* $B(y, x)$ edge pointing from x to y and from y to x , respectively. These form a pair of *reverse edges*. The reverse of a residual edge e is written as \overleftarrow{e} . We define the *residual cost* \mathbf{c} of a residual edge as $\mathbf{c}(F(x, y)) = c(x, y)$ and $\mathbf{c}(B(y, x)) = -c(x, y)$, respectively. For a flow f , we define the *residual capacities* \mathbf{u}_f . On forward edges, this is the difference between the actual capacity and the flow currently sent through this edge: $\mathbf{u}_f(F(x, y)) = u(x, y) - f(x, y)$. The capacity of a backward edge equals the flow assigned to the original edge: $\mathbf{u}_f(B(y, x)) = f(x, y)$.



■ **Figure 1** Flows, (residual) capacities and (residual) costs are black, green and red, respectively. Forward edges are blue, backward edges purple.

The residual capacity $u_f(p)$ of a path p is defined as $u_f(p) = \min\{u_f(e) \cdot e \in p\}$. The residual costs $c(p)$ are obtained by accumulating residual costs for the edges contained in p .

Note that the residual network can be considered a multi graph that has at most two copies of the same edge, e.g. $F(x, y)$ and $B(x, y)$ (see the residual network in Fig. 1b). Intuitively, a forward edge of the residual network indicates how much more could be added to the flow and a backward edge indicates how much could be removed from the flow.

► **Example 1.** Fig. 1a shows a flow network, where every edge is labelled by a flow, capacity, and a cost per unit flow. The colouring convention from the caption applies. Fig. 1b shows the residual network for the network in Fig. 1a. The residual network has, for every flow network edge (x, y) , two edges: one is the forward edge $F(x, y)$, a copy of the original edge, and the second is the backward edge $B(y, x)$, going in the opposite direction. Thus, forward edges of the residual network are labelled by the residual capacity, indicating how much more flow can still go through the network. Backward edges are also labelled by a capacity but, as stated earlier, the capacity is the flow going through the original edge, and the costs are negative, indicating that removing from the flow saves cost.

3 Towards a Simple Algorithm

Augmentation is a principal technique in combinatorial optimisation in which a candidate solution is incrementally improved until an optimal solution is found. In the context of flows, augmenting (along) a forward edge by a positive real γ means to increase the flow assigned to the original edge by γ . Augmenting along a backward edge is done by decreasing the flow value of the original edge by γ . The augmentation along a path is done by augmenting along each edge contained. We call a path of residual edges along which the residual capacities are strictly positive an *augmenting path*. Closed augmenting paths p with $c(p) < 0$ are *augmenting cycles*.

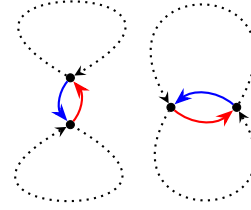
► **Example 2.** The result of augmenting our example flow from Fig. 1b is shown in Fig. 1c, where the flow is augmented along the edges (u, v) and (v, u) by 2. The resulting new residual network is shown in Fig. 1d, showing the change in capacities in forward and backward edges.

■ **Algorithm 1** *successive-shortest-path*($\mathcal{E}, \mathcal{V}, u, c, b$).

```

initialise  $b' \leftarrow b$  and  $f \leftarrow 0$ ;
while True do
  if  $\forall v \in \mathcal{V}. b'(v) = 0$  then
    return  $f$  as optimum flow;
  else
    take some  $s$  with  $b'(s) > 0$ ;
    if  $\exists t$  reachable from  $s \wedge b'(t) < 0$  then
      take such a  $t$  and
      a minimum cost augmenting path  $P$ 
      from  $s$  to  $t$ ;
       $\gamma = \min\{b(s), -b(t), u_f(P)\}$ ;
      augment along  $P$  by  $\gamma$ ;
       $b'(s) \leftarrow b'(s) - \gamma$ ;  $b'(t) \leftarrow b'(t) + \gamma$ ;
    else return infeasible;

```



■ **Figure 2 Eliminating FBPs:** Members of an *FBP* belong either to the same (left) or to two different cycles (right). When the *FBP* is dropped on the left, we obtain two new cycles. On the right, *FBP* deletion results in a single new cycle. Disjointness is preserved.

For this and all subsequent sections, we fix a weight-conservative flow network $\mathcal{G} = (\mathcal{E}, \mathcal{V}, u, c)$. Unless said otherwise, costs and capacities refer to this network. The balances b are kept generic. *Successive Shortest Path (SSP)* as given in Algorithm 1 is one of the most basic minimum cost flow algorithms. *successive-shortest-path*(\mathcal{G}, b) repeatedly selects a source s with positive balance, a target t with negative balance and a *minimum cost augmenting path* P connecting s to t , i.e. a minimum cost path in the residual network connecting s to t . Following that, it sends as much flow as possible, i.e. as much as the minimum capacity of any forward residual edge or as the balances of s and/or t allow, from s to t along P . The balances at s and t are lowered and increased by the same amount, respectively. This is done until all balances reach zero or infeasibility can be inferred from the absence of an augmenting path.

Conceptually, the algorithm is defined on *program states* consisting of variables $\mathcal{E}, \mathcal{V}, u, c$, balances b , remaining balances b' and the flow f . Invariants are predicates defined on states. If not all variables are relevant to an invariant, we say that only the involved variables satisfy the invariant.

Correctness of Algorithm 1. To prove that the algorithm is correct, we show that the following invariants hold for the states encountered during the main loop of *successive-shortest-path*(\mathcal{G}, b) (Algorithm 1):

1. The flow f is a minimum cost flow for the balance $b - b'$ ¹.
2. If capacities u and balances b are integral, then $b'(v)$ and $f(e)$ are integral for any vertex $x \in \mathcal{V}$ and $e \in \mathcal{E}$, respectively.
3. The sum of b' over all vertices v is zero: $\sum_{v \in \mathcal{V}} b'(x) = 0$.

Proving Invariant 1 was the most demanding and we dedicate most of this section to it. The other two invariants easily follow from the algorithm's structure. Correctness of all non-trivial algorithms for minimum cost flows depends on the following optimality criterion:

¹ For any v , this is defined as $(b - b')(v) = b(v) - b'(v)$.

► **Theorem 1** (Optimality Criterion [14]). *A flow f valid for balance b is optimum iff there is no augmenting cycle w.r.t f .*

Proof sketch. An augmenting cycle is a possibility to decrease costs while still meeting the balance constraints which gives one direction.

For the other direction, assume a valid flow f' with $c(f') < c(f)$. We define the flow g in the residual graph as $g(F(x, y)) = \max\{0, f'(x, y) - f(x, y)\}$ and $g(B(y, x)) = \max\{0, f(x, y) - f'(x, y)\}$. It can be shown that g is a flow in the residual graph with zero excess for every vertex, a so-called *circulation*. Moreover, $\mathbf{c}(g) = c(f') - c(f)$ and any residual edge e with $u_f(e) = 0$ has $g(e) = 0$. The circulation g can be *decomposed*, i.e. there is a set of cycles \mathcal{C} and weights $w : \mathcal{C} \rightarrow \mathbb{R}^+$ where any residual edge e has $g(e) = \sum_{C \in \mathcal{C}, e \in C} w(C)$. Since $0 > c(f') - c(f) = \mathbf{c}(g) = \sum_{C \in \mathcal{C}} w(C) \cdot \mathbf{c}(C)$, there has to be a cycle $C \in \mathcal{C}$ where $\mathbf{c}(C) < 0$. $u_f(C)$ must be positive making this an augmenting cycle w.r.t. f . ◀

Flow-decomposition [12, 8] as used in the proof of Theorem 1 is a fundamental technique in reasoning about flows. Now, pairs of residual edges where one is the reversed one of the other are called *forward-backward-pairs (FBP)*, e.g. $F(x, y)$ and $B(y, x)$. They are involved in lemmas one can use to prove preservation of the optimality invariant. Subsequently, *disjointness* of paths and cycles means their edge-disjointness.

► **Lemma 2.** *Deleting all FBPs from a set of disjoint cycles yields another set of disjoint cycles.*

Proof Sketch. Proof by induction on the number of FBPs. See cases from Fig. 2. ◀

► **Lemma 3.** *Assume an s - t -path P and some cycles \mathcal{C} where every FBP is between the path and a cycle, all items disjoint with one another. Deleting all FBPs results in an s - t -path and some cycles, again all disjoint.*

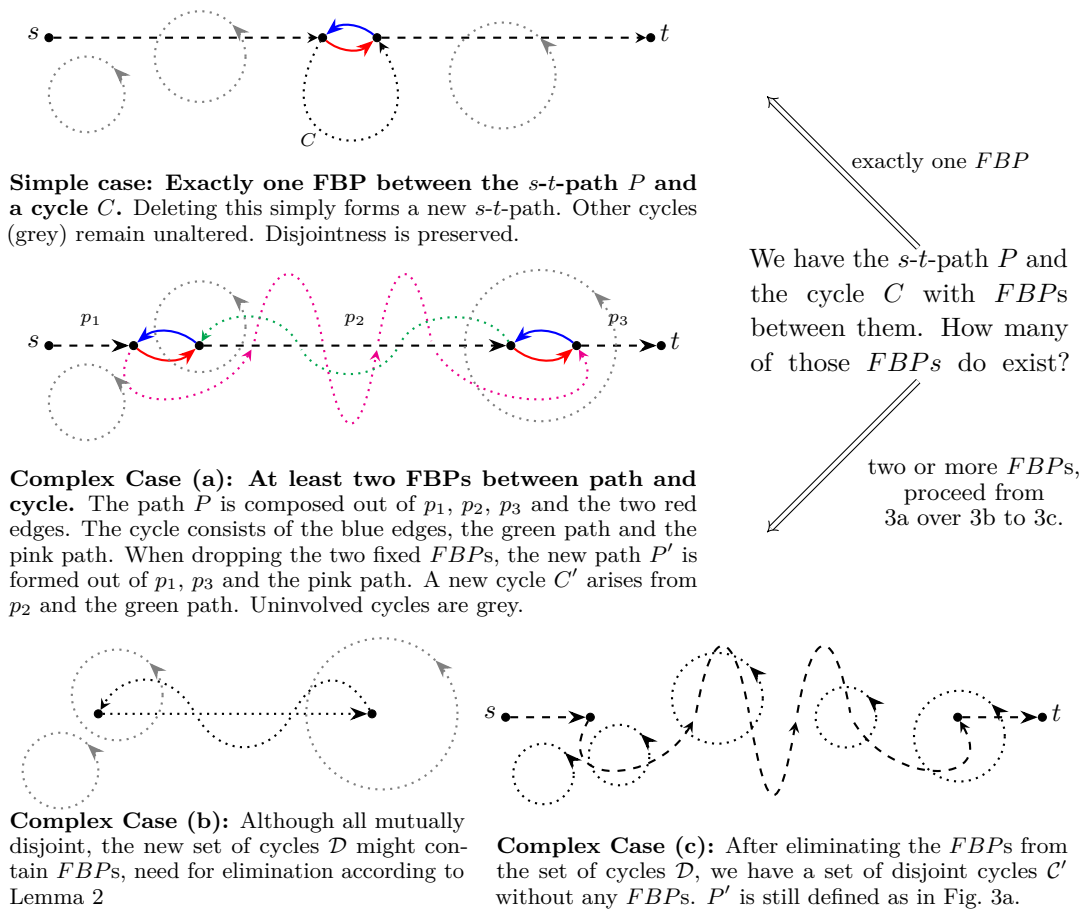
Proof Sketch. Proof by induction on the number of FBPs: Fix an arbitrary FBP which must be between the current path P and some cycle $C \in \mathcal{C}$. Now, we look at two cases.

Simple Case (Single FBP). It might be that this is the only FBP between P and C . By dropping it, we simply get a new s - t -path P' and may eliminate one cycle (Simple Case in Fig. 3). Still, there are no FBPs among or between cycles and the induction hypothesis can be applied immediately to P' and $\mathcal{C} \setminus \{C\}$.

Case (Several FBPs). We now consider the *first* and *last* FBPs between C and P according to the order given by P . By deleting those, we obtain a new s - t -path P' and a new cycle C' (Fig. 3a). Due to eliminating the first and last FBP, P' cannot have any FBPs within itself. But the set of cycles $\mathcal{D} = \mathcal{C} \setminus \{C\} \cup \{C'\}$ (Fig. 3b) may now contain FBPs, although they are still disjoint with one another. By Lemma 2, the FBPs can be deleted resulting in a set of disjoint cycles. This yields the path P' and cycles \mathcal{C}' from Fig. 3c. The number of FBPs has decreased, the substructures are disjoint and any FBP is between P' and a cycle in \mathcal{C}' to which the induction hypothesis may be applied. ◀

The following theorem implies the preservation of Invariant 1. This is perhaps not surprising since we always send the balance/flow along the cheapest augmenting paths.

► **Theorem 4** (Optimality Preservation [14]). *Let f be a minimum cost flow for balances b . Take an s - t -path P of minimum residual costs and $\gamma \leq u_f(P)$. If we augment f by γ along P then the result is still optimum for modified balances b' where $b'(s) = b(s) + \gamma$, $b'(t) = b(t) - \gamma$ and $b'(v) = b(v)$ for any other v .*



■ **Figure 3** Elimination of Forward-Backward-Pairs in the proof of Lemma 3.

Proof Sketch. P is vertex-disjoint since any cycle in P would have positive costs (Theorem 1) contradicting the optimality of P . Assume the flow f' after the augmentation were not optimum. By Theorem 1, there exists an augmenting cycle C . Wlog. C is vertex-disjoint: Otherwise split C into vertex-disjoint cycles of which one has negative residual costs.

Due to vertex-disjointness, neither P nor C can contain any $FBPs$. We can therefore apply Lemma 3 to P and C yielding another s - t -path P' and a set of cycles \mathcal{C} . Their edges have positive residual capacity w.r.t. f : For any $e \in P' \cup \mathcal{C}$ we have $u_f(e) > 0$ or $u_{f'}(e) > 0$. If only the latter holds, then $e \in C$ and $\overleftarrow{e} \in P$ which is an FBP . However, this would have been deleted. Since deleting $FBPs$ preserves costs, we have $c(P') + c(\mathcal{C}) = c(P) + c(C)$. Because $c(P') \geq c(P)$ (optimality of P) and $c(C) < 0$, there must be $D \in \mathcal{C}$ with $c(D) < 0$. This is an augmenting cycle w.r.t. f contradicting Theorem 1. ◀

Lemma 3 is a gap, which we cover with the construction above, in all published combinatorial proofs we are aware of, including the proof by Korte and Vygen [14]. The only other complete proof of Theorem 4 of which we are aware is a non-combinatorial proof by Orlin [22, 4], in which he uses advanced LP-theory.

► **Theorem 5** (Correctness of Algorithm 1). *Assume the sum of balances b over \mathcal{V} is zero. An execution of successive-shortest-path(\mathcal{G}, b) terminates, decides about the existence of a valid flow and returns one in case of existence.*

Proof Sketch. Due to weight conservativity, the zero flow is optimum for the zero balance making the optimality invariant (Invariant 1) initially true. Its preservation follows from Theorem 4. Invariants 3 and 2 also hold initially. Their preservation can be seen from the algorithm.

As the γ used for the augmentations will be a natural number, the sum of the absolute values of balances will decrease yielding a termination measure.

It remains to show that there is no valid flow if the procedure returns *infeasible*, a case for which we need some further auxiliary results. We note that f is a valid flow w.r.t. $b - b'$. We call the set of vertices X reachable from v in the residual graph its *residual cut*, denoted by $Rescut_f(v)$. It can be seen that any leaving edge must be saturated and any entering edge's flow is zero. The so-called *Flow-Value Lemma* says for any cut X , any b and any flow f valid w.r.t. b :

$$\sum_{v \in X} b(v) = \sum_{e \in \Delta^+(X)} f(e) - \sum_{e \in \Delta^-(X)} f(e)$$

Those two results yield for any b and any flow f valid w.r.t. b :

$$\sum_{x \in Rescut_f(v)} b(x) = \sum_{e \in \Delta^+(Rescut_f(v))} f(e) = \text{cap}(Rescut_f(v)) \quad (\text{Corollary 6})$$

We have $0 \leq \sum_{e \in \Delta^+(X)} f(e) \leq \text{cap}(X)$ and $0 \leq \sum_{e \in \Delta^-(X)} f(e) \leq \text{acap}(X)$ for any valid flow f and cut X . If there is a valid flow, this implies together with the Flow-Value Lemma:

$$-\text{acap}(X) \leq \sum_{v \in X} b(v) \leq \text{cap}(X) \quad (\text{Corollary 7})$$

The sum of balances is the amount of flow to be sent to or removed from the cut. This has to be within the bounds given by the capacities in both directions which is the intuition behind Corollary 7. From the algorithm's control flow we can infer that there must be an s with $b'(s) > 0$ without a reachable t where $b'(t) < 0$, i.e. any x in the rescut has a $b'(x) \geq 0$. We obtain a contradiction in case there exists a flow f' valid w.r.t. b .

$$\begin{aligned} \sum_{x \in Rescut_f(s)} b(x) &\leq \text{cap}(Rescut_f(s)) && (f' \text{ is valid flow and Corollary 7}) \\ &= \sum_{x \in Rescut_f(s)} (b - b')(x) && (\text{Corollary 6 for } f \text{ and } b - b') \\ &< \sum_{x \in Rescut_f(s)} b(x) && \blacktriangleleft \end{aligned}$$

Formalisation. We represent loops as recursive functions. The non-trivial termination argument requires the Isabelle function package [15]. We use records to model program states. The formal version of the loop in Algorithm 1 is given in Listing 1. Most notation is standard functional programming notation. The main exception is record updates, e.g. 'state(return := infeasible)' denotes 'state', but with state variable 'return' updated to 'infeasible'.

Formally, selecting reachable targets and minimum cost paths corresponds to using functions (*get-source* and *get-min-augpath*, respectively) that compute those items non-deterministically. Their existence and properties are assumed by a named context, a so-called *locale*. These allow us to fix constants and to make corresponding assumptions which are both available within the locale.

Listing 1: Recursive function formalising *SSP*

```

1  function SSP:: 'a Algo-state  $\Rightarrow$  'a Algo-state where
2    SSP state = (let b = balance state; f = current-flow state in
3    (if zero-balance b then state (return := found)
4    else (case get-source b of
5      None  $\Rightarrow$  state (return := infeasible) |
6      Some s  $\Rightarrow$  (case get-reachable-target f b s of
7        None  $\Rightarrow$  state (return := infeasible) |
8        Some t  $\Rightarrow$  (let P = get-min-augpath f s t;
9           $\gamma$  = min( min(b s) (- b t) ) (Rcap f (set P));
10         f' = augment-edges f  $\gamma$  P;
11         b' = ( $\lambda$  v. if v = s then b s -  $\gamma$  else
12           if v = t then b t +  $\gamma$  else b v)
13         in SSP (state (current-flow := f', balance := b' ))))))))

```

Listing 2: Customised simplification for *SSP*

```

1  lemma SSP-simps: assumes SSP-dom state
2    shows SSP-ret-1-cond state  $\Longrightarrow$  SSP state = (SSP-ret1 state)
3         SSP-ret-2-cond state  $\Longrightarrow$  SSP state = (SSP-ret2 state)
4         SSP-ret-3-cond state  $\Longrightarrow$  SSP state = (SSP-ret3 state)
5         SSP-call-4-cond state  $\Longrightarrow$  SSP state = SSP (SSP-upd4 state)

```

Listing 3: Customised induction rule for *SSP*

```

1  lemma SSP-induct: assumes SSP-dom state
2    assumes  $\bigwedge$ state. [[SSP-dom state;
3        SSP-call-4-cond state  $\Longrightarrow$  P (SSP-upd4 state)]]  $\Longrightarrow$  P state
4    shows P state

```

Listing 4: Single-step preservation of Optimality

```

1  lemma assumes SSP-call-4-cond state   invarOpt state
2    shows invarOpt (SSP-upd4 state)

```

We introduced definitions to specify which execution branch is taken when doing an iteration of the loop body, e.g. *SSP-call-4-cond state* indicating the recursive case from the function definition above. It has the same structure as the loop body and returns *True* for exactly one branch and *False* for all others. Similarly, the effect of a single execution branch can be modelled, e.g. *SSP-upd4 state*. We can show a simplification lemma and an induction principle for *SSP*, given in Listings 2 and 3, respectively. The preservation of the invariants is proven for single updates like the one in Listing 4 and by the simplification and induction lemmas this can be lifted to a complete execution of *SSP*. Proof automation makes this process very smooth and convenient. We follow the same formalisation methodology for all the algorithms we consider below.

4 The Capacity Scaling Algorithm

The naive Successive Shortest Path Algorithm (Algorithm 1) arbitrarily selects sources, targets and minimum cost (mincost) paths for the augmentations. This is refined to *Capacity Scaling (CS)* by selecting those triples where the residual capacities and balances are above a certain threshold that is halved from one scaling phase to another (Algorithm 2). It was proposed by Edmonds and Karp [7]. As *SSP*, *CS* works on a state consisting of \mathcal{E} , \mathcal{V} , u , c , b , b' and f . The algorithm uses two nested loops: The outer one is responsible for monitoring the scaling and determining problem infeasibility. The inner one's purpose is to process every suitable path until none are remaining. It is also responsible for terminating the execution when a solution has been found. As this refines *SSP*, the proofs for correctness and termination do not differ significantly. The same three invariants may be reapplied. Note that capacities and balances must still be integral to ensure termination.

■ **Algorithm 2** *capacity-scaling*($\mathcal{E}, \mathcal{V}, u, c, b$).

```

1 initialise  $b' \leftarrow b$ ,  $f \leftarrow 0$  and  $\gamma = 2^{\lfloor \log_2 B \rfloor}$  where  $B = \max\{1, \frac{1}{2} \sum_{v \in \mathcal{V}} b(v)\}$ ;
2 while True do
3   while True do
4     if  $\forall v \in \mathcal{V}. b'(v) = 0$  then return  $f$ ;
5     else if  $\exists s t P. P$  is  $s$ - $t$ -path,  $u_f(P) \geq \gamma$ ,  $b'(s) \geq \gamma$  and  $b'(t) \leq -\gamma$  then
6       take such  $s, t$  and  $P$ ; augment  $\gamma$  along  $P$ ;
7        $b'(s) \leftarrow b'(s) - \gamma$ ;  $b'(t) \leftarrow b'(t) + \gamma$ ;
8     else break;
9   if  $\gamma = 1$  then return infeasible;
10  else  $\gamma \leftarrow \frac{1}{2} \cdot \gamma$ ;
```

Listing 5: Formalisation of Scaling

```

1 function (domintros) SSP:: 'a Algo-state  $\Rightarrow$  'a Algo-state where
2   SSP state = (let b = balance state; f = current-flow state in
3     (if zero-balance b then state (return := success)
4     else (case get-source-target-path f b of
5       None  $\Rightarrow$  state (return := notyetterm) |
6       Some(s, t, P)  $\Rightarrow$  (let  $\gamma = \min(\min(b\ s) (-b\ t))$  (Rcap f (set P));
7         f' = augment-edges f  $\gamma$  P;
8         b' = ( $\lambda v. \text{if } v = s \text{ then } b\ s - \gamma \text{ else}$ 
9           if  $v = t$  then  $b\ t + \gamma$  else  $b\ v$ )
10        in SSP(state (current-flow := f', balance := b' )))))
11
12 definition ssp ( $\gamma :: \text{nat}$ )  $\equiv$  SSP.SSP  $\mathcal{E}\ u$  (get-source-target-path  $\gamma$ )
13
14 function (domintros) Scaling:: nat  $\Rightarrow$  'a Algo-state  $\Rightarrow$  'a Algo-state where
15   Scaling l state = (let state' = ssp (2l-1) state in
16     (case return state' of success  $\Rightarrow$  state'
17     | failure  $\Rightarrow$  state'
18     | notyetterm  $\Rightarrow$  (if l = 0 then state' (return = failure)
19     else Scaling (l-1) state')))
```

Intuitively CS behaves like SSP, but only greedily chooses large steps towards the optimal solution, thus hastening progress leading to a polynomial rather than exponential worst-case running time. Each of these steps is a minimum cost augmenting path p from a source s to a target t with a “step size” of $\min\{u_f(p), b(s), -b(t)\}$. When no paths of the right cost remain, it halves the thresholds for treatment and continues with a more fine-grained analysis.

Formalisation. As it can be seen from Listing 5, a modified version of SSP realises the inner loop. Paths returned by the selection function are assumed to have capacity above γ , which is enforced by the definition statement. In case of existence, *get-source-target-path* γ returns a source, a target and a minimum cost path with balances and capacity above γ . The outer loop works on the logarithm of γ , denoted by l . The major difference between *SSP* from Listing 5 and the one from Listing 1 is Line 5: In the modified version, the flag is set to *notyetterm* which indicates that no more suitable paths were found and the decision on infeasibility is left to the outer loop.

Most of the claims and proofs are inherited from the formalisation of SSP and therefore they are conditioned on termination for the respective input state. This can be proven from Invariant 2. The outer loop is a function on two arguments, namely, the logarithm of the threshold and the program state. Its termination follows from the decrease in γ .

For SSP, we had the sum of the balances’ absolute values as termination measure decreasing in any iteration by at least 1. This is linear in the balances and therefore exponential w.r.t. input length. On the contrary, CS halves the measure after a polynomial number of iterations

resulting in fast progress. The number of scaling phases is logarithmic w.r.t. the greatest balance and thus, linear in terms of input length. The time for finding a minimum cost path is polynomial and an augmentation is $\mathcal{O}(n)$. Provided infinite capacities, the number of augmentations per phase is at most $4n$ [14]. For an efficient path computation, CS even runs in $\mathcal{O}(n(n^2 + m) \log B)$ [7, 14], where B is the greatest absolute value of a balance. This is polynomial w.r.t. input length including the representation of balances. It is not polynomial w.r.t. only the number of vertices and edges. Such an algorithm would be *strongly polynomial*.

5 Orlin's Algorithm

Orlin's Algorithm (Algorithm 3) allows for a strongly polynomial worst-case running time of $\mathcal{O}(n \log n(m + n \log n))$ [22, 4, 14]. Similar to Algorithm 2, we have an outer loop monitoring the threshold γ and an inner loop treating all paths with a capacity above that (*augment-edges()*). After each threshold decrease, a forest that is maintained by the algorithm is updated. In the return value of *augment-edges()*, the *flag* indicates whether an optimum flow was found or infeasibility was detected. Otherwise (i.e. *flag = notyetterm*), the algorithm continues. The top loop and subprocedures work on a program state consisting of the variables $\mathcal{E}, \mathcal{V}, u, c, b, b', f, \mathcal{F}, \text{actives}, \gamma$ and r .

Here, the flow on only *active edges* and forest edges can be augmented, which is done using *augment-edges()*. All edges are active initially. Edges deleted from this set are *deactivated*. The subprocedures use a small positive constant ϵ . Its value influences the timespans between component merges. Positivity ensures termination of *augment-edges()*.

For the previous algorithms, the running time depended on B . On the contrary, Orlin's Algorithm avoids that using a continuously growing *spanning forest* \mathcal{F} of edges. The crucial observation is that this forest is used s.t. only one vertex (henceforth, the representative) per forest connected component (henceforth, *\mathcal{F} -component*) is considered as a source or as a target in searching for augmenting paths. This reduction in augmentation effort is achieved by maintaining the forest, which can be done in time polynomial in n and m .

■ **Algorithm 3** *orlins*($\mathcal{E}, \mathcal{V}, u, c, b$).

```

1 initialise  $b' \leftarrow b; f \leftarrow 0; r(v) \leftarrow v$  for any  $v; \mathcal{F} \leftarrow \emptyset; \text{actives} = \mathcal{E}; \gamma \leftarrow \max_{v \in \mathcal{V}} |b'(v)|;$ 
2 while True do
3    $(b', f, \text{flag}) \leftarrow \text{augment-edges}(\mathcal{E}, \mathcal{V}, u, c, b', f, \mathcal{F}, \text{actives}, \gamma, r);$ 
4   if flag = found then return  $f$ ;
5   if flag = infeasible then return infeasible;
6   if  $\forall e \in \text{actives}. f(e) = 0$ 
7   then  $\gamma \leftarrow \min\{\frac{\gamma}{2}, \max_{v \in \mathcal{V}} |b'(v)|\};$ 
8   else  $\gamma \leftarrow \frac{\gamma}{2};$ 
9    $(b', f, \mathcal{F}, r, \text{actives}) \leftarrow \text{maintain-forest}(\mathcal{E}, \mathcal{V}, u, c, b', f, \mathcal{F}, \text{actives}, \gamma, r);$ 

```

Partial correctness of the algorithm is shown by invariants on program states and properties of the subprocedures. Line specifications refer to the state *after* executing the respective line. Consider the following invariants about the execution of *orlins*(\mathcal{G}, b):

1. γ is strictly positive, except when $b = 0$.
2. Any active edge e outside \mathcal{F} has a flow $f(e)$ that is a non-negative integer multiple of γ .
3. Endpoints of a deactivated edge belong to the same \mathcal{F} -component.

4. Only representatives can have a non-zero balance.
5. For states in Line 3, any edge $e \in \mathcal{F}$ has $f(e) > 4 \cdot n \cdot \gamma$.
6. f is optimum for the balance $b - b'$.

The *balance potential* Φ is important for the number of augmentations during subprocedures and their termination [14]. For b' and γ it is defined as:

$$\Phi(b', \gamma) = \sum_{v \in \mathcal{V}} \left[\frac{|b'(v)|}{\gamma} - (1 - \epsilon) \right]$$

When writing $\Phi(\text{state})$, we refer to the respective b' and γ . We will later see in detail how the subprocedures work. For now, assume the subprocedures have the following properties:

- P1. For the result of *augment-edges()*, we have $|b'(x)| \leq (1 - \epsilon) \cdot \gamma$ for any x . If $\text{flag} \neq \text{found}$ there is x with $b'(x) > 0$. flag is *found* when $b' = 0$ is reached.
- P2. For any e , the change in $f(e)$ due to calling *augment-edges()* is an integer multiple of γ .
- P3. Calling *maintain-forest()* preserves Invariants 3 and 4. Calling *augment-edges()* preserves Invariant 4.
- P4. For any edge that is in \mathcal{F} after calling *maintain-forest()*, the flow reduction incurred during this subprocedure is at most $n\beta$ where $\forall v. |b'(v)| \leq \beta$ before calling *maintain-forest()*. For edges e outside \mathcal{F} after the call, there was no change in $f(e)$.
- P5. $\Phi(\text{maintain-forest}(\text{state})) \leq \Phi(\text{state}) + n$.
- P6. Provided the other invariants hold, Invariant 6 is preserved by either subprocedure.
- P7. The number of path augmentations during *augment-edges(state)* is at most $\Phi(\text{state})$. The flow decrease along any edge during *augment-edges(state)* is at most $\Phi(\text{state}) \cdot \gamma$.
- P8. Assume all invariants hold on *state*. If $\text{flag} = \text{found}$ in the result of *augment-edges(state)*, f is optimum. If $\text{flag} = \text{infeasible}$, the flow problem is indeed infeasible.

We examine how the invariants and properties yield partial correctness.

► **Theorem 8 (Partial Correctness).** *Assume the algorithm terminates on an uncapacitated instance with conservative weights. If a flow is found, it is a mincost flow, otherwise the problem is infeasible.*

Proof. $b = 0$ yields immediate termination with $f = 0$ as correct result (P1). If $b \neq 0$, we show the invariants for the last state on which *augment-edges()* is called. All invariants hold for the initialisation given in the algorithm. The pseudocode and P1 imply preservation of Invariant 1. The flow along edges outside \mathcal{F} is only changed by *augment-edges()* (P4) and the change is integral multiple of γ (P2) implying preservation of Invariant 2. The arguments for Invariants 3 and 4 are simple (P3). Preservation of Invariant 5 is more difficult. Assume it holds in Line 3. We know $|b'(x)| \leq (1 - \epsilon) \cdot \gamma$ for any x (P1). After modifying γ (Lines 6 - 8), the flow along forest edges is above $8n\gamma$, $\Phi(b', \gamma) \leq n$ and $\forall x. |b'(x)| < 2 \cdot \gamma$. Executing *maintain-forest()* can cause a decrease of $2n\gamma$ for forest edges (P4) and an increase in Φ by at most n (P5). Calling *augment-edges()* is responsible for a further flow decrease of at most $2n\gamma$ (P7). The overall decrease along forest edges was at most $4n\gamma$. By P6 we obtain Invariant 6 for the state before the last call of *augment-edges()*. P8 gives the claim. ◀

Note: we restrict ourselves to infinite edge capacities, which is insignificant as problems can be reduced to the uncapacitated setting with a linear increase in input length [14]. Our theorems here require weight-conservativity which is inherited from Algorithm 1 and Algorithm 2 as a constraint. However, we drop the restriction to integral capacities and balances.

We now state the subprocedures and show that they satisfy the aforementioned properties.

5.1 Augmenting the Flow

■ **Algorithm 4** *augment-edges*($\mathcal{E}, \mathcal{V}, u, c, b', f, \mathcal{F}, \text{actives}, \gamma$).

```

A1 while True do
A2   if  $\forall v \in \mathcal{V}. b'(v) = 0$  then return  $(b', f, \text{found})$ ;
A3   else if  $\exists s. b'(s) > (1 - \epsilon) \cdot \gamma$  then
A4     if  $\exists t. b'(t) < -\epsilon \cdot \gamma \wedge t$  is reachable from  $s$  then
A5       take such  $s, t$ , and a connecting path  $P$  with original edges from  $\text{actives} \cup \mathcal{F}$ ;
A6       augment  $f$  along  $P$  from  $s$  to  $t$  by  $\gamma$ ;
A7        $b'(s) \leftarrow b'(s) - \gamma; b'(t) \leftarrow b'(t) + \gamma$ ;
A8     else return  $(b', f, \text{infeasible})$ ;
A9   else if  $\exists t. b'(t) < -(1 - \epsilon) \cdot \gamma$  then
A10    if  $\exists s. b'(s) > \epsilon \cdot \gamma \wedge t$  is reachable from  $s$  then
A11      take such  $s, t$ , and a connecting path  $P$  with original edges from  $\text{actives} \cup \mathcal{F}$ ;
A12      augment  $f$  along  $P$  from  $s$  to  $t$  by  $\gamma$ ;
A13       $b'(s) \leftarrow b'(s) - \gamma; b'(t) \leftarrow b'(t) + \gamma$ ;
A14    else return  $(b', f, \text{infeasible})$ ;
A15  else return  $(b', f, \text{please continue})$ ;

```

We now argue why *augment-edges*() satisfies the properties stated in the previous section. P1, P2, P3, P6, P7 and P8 assert something about *augment-edges*(). P1 can be inferred from the subprocedure's structure. The only possible amount to augment is γ which yields P2. P3 (Preservation of Invariant 4) also follows from the structure.

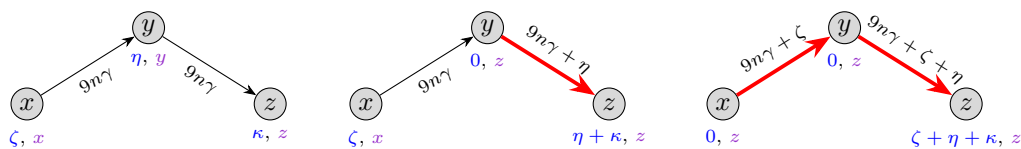
Proof Sketch for P6. We assume the invariants for hold *state* and we define $(b', f, \text{flag}) = \text{augment-edges}(\text{state})$. Then f is a minimum cost flow for balances $b - b'$ because of Theorem 4. The restriction to active and forest edges (Lines A5 and A11) is unproblematic: Simulate deactivated edges with forest paths which are of minimum costs. ◀

Proof Sketch for P7. Show that any iteration decreases Φ at least by 1, thus *augment-edges*(*state*) performs at most $\Phi(\text{state})$ iterations and the flow decrease for an edge is at most $\Phi(\text{state}) \cdot \gamma$. The proof of a strict decrease in Φ only works for $\epsilon > 0$. ◀

Proof Sketch for P8. If the algorithm found a flow, then $\forall v \in \mathcal{V}. b'(v) = 0$ (Line A2). Together with preservation of the optimality invariant, it gives the first subclaim.

If the algorithm asserts infeasibility, one can exploit information about b' from Lines A3, A4, A9 and A10. By employing residual cuts and the analogous definition where every direction is reversed, one can show infeasibility for both cases. The proof is similar to that of Theorem 5. The argument only works for $\epsilon \leq \frac{1}{n}$. ◀

By this, we have shown that *augment-edges*() satisfies all asserted properties. We also saw the restriction $0 < \epsilon \leq \frac{1}{n}$. Note: one might think that ϵ could be assigned to 0. As we shall see later on, that would make it impossible for us to derive the worst-case running time bound. In essence, we need to allow vertices to be processed if they are 'slightly' below the threshold, otherwise the algorithm take exponentially many iterations, and its running time will depend on B . Interested readers should consult sec. 5.2 [22].



■ **Figure 4** Merging forest components: Remaining balances b' , forest edges, flow values and representatives are blue, red, black and purple, respectively. Assume the balances are non-negative.

■ **Algorithm 5** *maintain-forest*($\mathcal{E}, \mathcal{V}, u, c, b', f, \mathcal{F}, \text{actives}, \gamma, r$).

```

F1 while  $\exists e = (x, y). e \in \text{actives} \wedge e \notin \mathcal{F} \wedge f(e) > 8n\gamma$  do
F2    $\mathcal{F} \leftarrow \mathcal{F} \cup \{e\}$ ;
F3   if  $|\mathcal{F}$ -component of  $y| \geq |\mathcal{F}$ -component of  $x|$  then exchange  $x$  and  $y$ ;
F4   let  $x' = r(x)$  and  $y' = r(y)$  the respective representatives;
F5   take residual path  $Q \subseteq \mathfrak{F}$  connecting  $x'$  and  $y'$ ;
F6   if  $b'(x') > 0$  then augment  $f$  along  $Q$  by  $b'(x)$  from  $x'$  to  $y'$ ;
F7   else augment  $f$  along  $\overleftarrow{Q}$  by  $-b'(x)$  from  $y'$  to  $x'$ ;
F8    $b'(y') \leftarrow b'(y') + b'(x')$ ;  $b'(x') = 0$ ;
F9   foreach  $v$  reachable from  $y'$  in  $\mathfrak{F}$  do  $r(v) \leftarrow y'$ ;
F10  foreach  $d = (u, v). d \in \text{actives} \wedge \{r(u), r(v)\} = \{y'\}$  do  $\text{actives} \leftarrow \text{actives} \setminus \{d\}$ ;
F11 return  $(b', f, \mathcal{F}, \text{actives}, r)$ ;

```

5.2 Maintaining the Forest

We now discuss the last part of the algorithm, namely, maintaining the forest. The definition of *maintain-forest*() can be seen in Algorithm 5. We add active edges with flow above $8n\gamma$ to \mathcal{F} , which inevitably changes the connected components of the forest – a new component is created for every added edge by merging the components the respective endpoints belong to. Since non-zero balance is only allowed for representatives, balances must be re-concentrated at one of the two representatives after merging two components.² Moreover, all edges between them are deactivated and the representatives must be updated. For a forest \mathcal{F} , \mathfrak{F} is the corresponding residual network consisting only of forest edges. Re-concentration is done by augmentations along paths in \mathfrak{F} . An example of how balances, flow and forest change is displayed in Fig. 4.

Only P3-P6 make assertions about *maintain-forest*() and we argue why they are indeed satisfied. P3 (preservation of Invariant 3) is easy to see since it is precondition for deactivation to have the same representatives (Line F10).

Proof Sketch for P4. Invariants F1 and F2 bound the forest edges' flow decrease:

F1. For any x , $|b'(x)|$ is bounded by the product of its \mathcal{F} -component's cardinality and β .

F2. For any $e \in \mathcal{F}$, we have $f(e) > \alpha - \beta \cdot |X|$ where X is the \mathcal{F} -component of e .

Their conjunction is preserved by *maintain-forest*(). It is important to always concentrate the balances at the larger component's representative as done in the algorithm. ◀

Any iteration merges two \mathcal{F} -components making $n - 1$ an upper bound for the number of iterations. P5 asserts $\Phi(\text{maintain-forest}(\text{state})) \leq \Phi(\text{state}) + n$. This holds because the potential cannot increase by more than 1 per iteration, as shown in the following lemma.

² Cardinalities of forest components in the pseudocode refer to the number of vertices in the component.

► **Lemma 9.** *The increase of Φ during a single iteration of `maintain-forest()` is at most 1.*

Proof. Let *state* be the program state in Line F1 and *state'* be the one in Line F9, respectively. Program variables refer to *state*. It follows:

$$\begin{aligned}
 \Phi(\text{state}') &= \sum_{v \in \mathcal{V} \setminus \{x', y'\}} \left\lceil \frac{|b'(v)|}{\gamma} - (1 - \epsilon) \right\rceil + \left\lceil \frac{0}{\gamma} - (1 - \epsilon) \right\rceil + \left\lceil \frac{|b'(y') + b'(x')|}{\gamma} - (1 - \epsilon) \right\rceil \\
 &= \sum_{v \in \mathcal{V} \setminus \{x', y'\}} \left\lceil \frac{|b'(v)|}{\gamma} - (1 - \epsilon) \right\rceil + \left\lceil \frac{|b'(y')| + |b'(x')|}{\gamma} - (1 - \epsilon) \right\rceil \\
 &\leq \sum_{v \in \mathcal{V} \setminus \{x', y'\}} \left\lceil \frac{|b'(v)|}{\gamma} - (1 - \epsilon) \right\rceil + \left\lceil \frac{|b'(x')|}{\gamma} \right\rceil + \left\lceil \frac{|b'(y')|}{\gamma} - (1 - \epsilon) \right\rceil \\
 &\leq \sum_{v \in \mathcal{V} \setminus \{x', y'\}} \left\lceil \frac{|b'(v)|}{\gamma} - (1 - \epsilon) \right\rceil + \left\lceil \frac{|b'(x')|}{\gamma} - (1 - \epsilon) \right\rceil + 1 + \left\lceil \frac{|b'(y')|}{\gamma} - (1 - \epsilon) \right\rceil \\
 &= \Phi(\text{state}) + 1
 \end{aligned}$$

Proof of P6. To apply Theorem 4 we need optimality of forest paths Q and \overleftarrow{Q} used for augmentations. Suppose this were not true. There is P with $\mathbf{c}(P) < \mathbf{c}(Q)$ connecting the same vertices. Since $\mathbf{c}(\overleftarrow{Q}) = -\mathbf{c}(Q)$, $P\overleftarrow{Q}$ is a cycle with $\mathbf{c}(P\overleftarrow{Q}) = \mathbf{c}(P) + \mathbf{c}(\overleftarrow{Q}) < 0$ and $\mathbf{u}_f(P\overleftarrow{Q}) > 0$ contradicting Theorem 1 and Invariant 6. The case for \overrightarrow{Q} is analogous. Due to Invariant 5, the flow in \mathcal{F} is positive. Infinite edge capacities imply positive residual capacities for \mathfrak{F} . ◀

This concludes our proofs that `maintain-forest()` satisfies properties P3-P6.

5.3 Termination and Running Time

For the inner loops we have termination measures decreasing in any iteration, namely, the number of components and Φ . For the outer loop, there is a maximum number of iterations until some desirable situation occurs. A vertex v is *important* iff $|b'(v)| > (1 - \epsilon) \cdot \gamma$ [14], i.e. its contribution to Φ is positive. We repeatedly wait for the occurrence of an important vertex and ensure a merge of two forest components some iterations later. We define $\ell = \lceil \log(4 \cdot m \cdot n + (1 - \epsilon)) - \log \epsilon \rceil + 1$ and $k = \lceil \log n \rceil + 3$. It can be shown that (a) if we wait for $k + 1$ iterations, a vertex has become important or there is a component merge, and that (b) for an important vertex, $\ell + 1$ iterations enforce its component being increased. There can be at most $n - 1$ such merges, which yields termination.

Concerning running time, we assume *atomic bounds* for basic parts of the algorithm. For instance, t_{FB} is an upper bound for the time consumed when executing the loop body in `maintain-forest()` and t_{FC} is the time for checking the condition (analogously t_{AC} and t_{AB} for `augment-edges()`, and t_{OC} and t_{OB} for `orlins()`). Time consumption can be bounded by

$$\begin{aligned}
 &(n \cdot (\ell + k + 2) - 1) \cdot (t_{OC} + t_{OB} + t_{AC} + t_{FC}) + \\
 &(n - 1) \cdot (t_{FC} + t_{FB} + t_{AC} + t_{AB}) + (2n - 1) \cdot (\ell + 1) \cdot (t_{BC} + t_{BB}) + t_{BC} + t_{OC}
 \end{aligned}$$

The proof involves bounding the number of iterations of the subprocedures by n and Φ , respectively, a connection between the number of important vertices and Φ , and bounding the number of occurrences of important vertices by results on so-called *Laminar Families*.

5.4 Formalisation

Listing 6: Locale to specify path selection for *augment-edges()*

```

1 locale augment-edges = algo +
2   fixes get-source-target-path :: 'a Algo-state ⇒ 'a ⇒ 'a ⇒ 'a Redge list and
3   get-vertex :: ('a ⇒ bool) ⇒ 'a
4   assumes get-source-target-path-axioms:
5     [[get-source-target-path state s t = P ; s ∈ V ; t ∈ V ; s ≠ t
6     aux-invar state ; (∀ e ∈ F state . current-flow state e > 0) ;
7     resreach(current-flow state) s t ]] ⇒
8     (Rcap(current-flow state)(set P) > 0 ∧
9     (invar-isOptflow state → is-min-path(current-flow state) s t P) ∧
10    oedge ` set P ⊆ actives state ∪ F state ∧ distinct P

```

Listing 7: Formalisation of the Top Loop of Orlin's algorithm.

```

1 function (domintros) orlins :: 'a Algo-state ⇒ 'a Algo-state where
2   orlins state = (if return state = success then state
3   else if return state = failure then state
4   else (let f = current-flow state ; b = balance state ;
5         γ = current-γ state ; E' = actives state ;
6         γ' = (if ∀ e ∈ E' . f e = 0 then min(γ / 2) (Max { | b v | | v ∈ V })
7         else (γ / 2)) ;
8         state' = maintain-forest(state (current-γ := γ' )) ;
9         in orlins (augment-edges state'))))

```

Listing 8: A function modelling the running time of *orlins()*.

```

1 function (domintros) orlinsTime :: nat ⇒ ('a, 'b, 'd) Algo-state
2   ⇒ nat × ('a, 'b, 'd) Algo-state where
3   (orlinsTime ttOC state) = (if (return state = success) then (ttOC, state)
4   else if (return state = failure) then (ttOC, state)
5   else (let f = current-flow state ; b = balance state ;
6         γ = current-γ state ; E' = actives state ;
7         γ' = (if ∀ e ∈ to-set E' . f e = 0 then
8         min(γ / 2) (Max { | b v | | v ∈ V })
9         else (γ / 2)) ;
10        state'time = maintain-forestTime(state (current-γ := γ' )) ;
11        state''time = augment-edgesTime(snd state'time)
12        in ((ttOC + tOB + fst state'time + fst state''time)
13        +++ (orlinsTime ttOC (snd state''time))))

```

General. We assume functions selecting paths non-deterministically via locales (Listing 6), which were later instantiated suitably. As above, we model program states as records and use customised simplification and induction. However, here the algorithm's complexity is more substantial (see Listings 9 and 7). Also, we note that formal proofs about paths often need pairwise distinctness of the vertices or edges which is often neglected in an informal setting.

Forest. For connected components, we reuse Abdulaziz et al.'s [2] formalisation modelling *undirected* edges as sets. Paths based on that must be transformed to paths over residual edges. Each direction is mapped to a residual edge pointing in this direction. Residual edges realising opposite directions originate from the same graph edge. This implies that converting a path over undirected edges and its reverse yields opposite costs as needed to show P6.

Termination. The termination proof reasons about a fixed number of iterations. We introduced definitions expressing the effect of a single iteration, which can then be combined to a fixed number by the function iteration *compow*. As soon as executing a single step does not change anything, the recursive version would also terminate yielding an equal result.

Running Time. We model algorithm running times as functions returning natural numbers, using an extension of Nipkow et al.'s approach. In this approach, for every function $f : \alpha \rightarrow \beta$, we devise a functional program $f_{\text{Time}} : \alpha \rightarrow \mathbb{N}$ with the same recursion and control-flow structure as the algorithm whose running time we measure. In its most basic form, for a

given input $x : \alpha$, $f_{\text{Time}}(x)$ returns the number of the recursive calls that f would perform while processing x . If f involves calls to other functions, the running times of the called functions are added to the number of recursive calls of f . To modularise the design of these running time models, we use locales to assume running times of the called functions. An example of such a running time model is shown in Listing 8, which models the running time of Orlin’s algorithm. Here, there running times of the forest and path augmentation procedures are only assumed, without explicitly specifying them, in the locale containing the definition of *orlinsTime*. Mathematically, proving the upper bound on the running time requires a basic result about *laminar families*, where the set of connected components of the forest is viewed as a laminar family.

Listing 9: Formalisation of *augment-edges()* and corresponding Induction Rule

```

1  function (domintros) augment-edges:: 'a Algo-state ⇒ 'a Algo-state  where
2    augment-edges state = (let f = current-flow state; b = balance state;
3                          γ = current-γ state
4    in (if ∀ v ∈ V. b v = 0 then state (| return=success)
5      else if ∃ s ∈ V. b s > (1 - ε) * γ then
6        (let s = get-vertex (λ s. b s > (1 - ε) * γ ∧ s ∈ V)
7          in (if ∃ t ∈ V. b t < -ε * γ ∧ resreach f s t then
8            let t = get-vertex (λ t. b t < -ε * γ ∧ resreach f s t ∧ t ∈ V);
9              P = get-source-target-path state s t;
10             f' = augment-path f γ P;
11             b' = (λ v. if v = s then b s - γ
12                  else if v = t then b t + γ else b v);
13             state' = state (| current-flow:= f', balance:= b') in
14                augment-edges state'
15             else state (| return:= failure)))
16        else if ∃ t ∈ V. b t < -(1 - ε) * γ then
17          (let t = get-vertex (λ t. b t < -(1 - ε) * γ ∧ t ∈ V)
18            in (if ∃ s ∈ V. b s > ε * γ ∧ resreach f s t then
19              let s = get-vertex (λ s. b s > ε * γ ∧ resreach f s t ∧ s ∈ V);
20                P = get-source-target-path state s t;
21                f' = augment-path f γ P;
22                b' = (λ v. if v = s then b s - γ
23                     else if v = t then b t + γ else b v);
24                state' = state (| current-flow:= f', balance:= b') in
25                   augment-edges state'
26                else state (| return:= failure)))
27          else state (| return:= notyetterm)))
28
29 lemma augment-edges-induct: assumes augment-edges-dom state
30   ∧ state. [ augment-edges-dom state ;
31             augment-edges-call1-cond state ⇒ P(augment-edges-call1-upd state);
32             augment-edges-call2-cond state ⇒ P(augment-edges-call2-upd state) ]
33   ⇒ P state

```

6 Discussion

The algorithms that we considered here share a number of features with other maximum flow algorithms that were formally analysed before, most notably the fact that they iteratively compute augmenting paths to incrementally improve a solution until an optimal solution is reached. Those algorithms also use residual graphs, which are intuitively graphs containing the remaining capacity w.r.t. the current flow maintained by the algorithm, and which have been formalised by Lammich and Sefidgar [17]. The most advanced one out of these is probably the Push-Relabel Algorithm by Goldberg and Tarjan. Another combinatorial optimisation algorithm that was also formalised is Edmonds’ blossom shrinking algorithm for maximum cardinality matching in general graphs [2].

However, our work here is different from those previously studied algorithms for maximum flow in one crucial aspect: here we cover the algorithms which use *scaling*, a technique for designing fast optimisation algorithms, including algorithms with the fastest worst-case

running times for different variants of matching and shortest path problems [6, 9, 10, 11, 23], in addition to minimum-cost flows. Our work here provides a blueprint to formalise the correctness of those other scaling algorithms. Furthermore, the running time proof here depends on properties of laminar families, making it one of the more advanced running time proofs to be formalised.

A main outcome of our work, from a mathematical perspective, is our proof of Theorem 4, which is its first complete combinatorial proof, and our simplified proof of Theorem 1. Those outcomes, especially the construction we devised for the former, highlight the potential role of formalising deep proofs in filling gaps as well as in the generation of new proofs and/or insights. Indeed, this was a recurring theme across the project, e.g. there were other non-trivial claims in the main textbook by Korte and Vygen we used as reference [14], for which no proof is given. E.g. it is claimed that the accumulated augmentations through a forest edge are below $2(n-1)\gamma$. We could not formalise the proof of this in the textbook as it also had gaps, and we devised Invariants F1 and F2 to be able to prove it. Other gaps were in the proofs of Properties P7, P6, Lemma 5, Lemma 9, and a few parts of the termination proof, but we have to refer interested readers to the formalisation due to the lack of space.

The formalisation of the algorithms presented here is around 25K lines of proof scripts. Our methodology is based on using Isabelle/HOL's *locales* to implement Wirth's notion of step-wise refinement [25], thereby compartmentalising different types of reasoning. This locale-based implementation of refinement was used earlier by many authors, e.g. by Nipkow [21], Maric [20], and Abdulaziz et al. [2]. In this approach, non-deterministic computation is handled by assuming the existence and properties of functions that compute non-deterministically, without assuming anything about the functions' implementation. Our formalisation is one further example showing that this approach scales to proving the correctness of some of the most sophisticated algorithms. We also note that our focus here is more on formalising the mathematical argument behind the algorithm and less on obtaining an executable program, which is nonetheless attainable using this locale-based approach. A notable alternative implementation of refinement is Peter Lammich's [16] framework.

We note that in our work, we have used the locale-based approach in two ways: top-down and bottom-up. Our approach to specifying *augment-edges* was top-down, where we assumed the existence of a procedure for finding shortest paths between sources and targets. On the other hand, for specifying *orlins* we went bottom-up, where we first defined and proved the correctness of *augment-edges* and *maintain-forest*, and then started defining and proving the correctness of *orlins*, which calls both *augment-edges* and *maintain-forest*. In the former case, we went top-down as we had a good a priori understanding of the functions to assume and their properties, while in the latter we went bottom-up because we had a poorer a priori understanding of the functions to assume and their properties.

Furthermore, we define procedures as recursive functions using Isabelle's function package [15], program states as records, and invariants as predicates on states. We devise automation based on manually deriving theorems characterising properties of recursive functions, and combining those theorems with Isabelle's classical reasoning and simplification. In this approach, the automation handles proofs of invariants and monotone properties, e.g. the growth of the forest or the changes in Φ . Again, other approaches can be used to model algorithms and automate reasoning about them, e.g. monads [18] or while combinators [5]. Both of those approaches allow for greater automation, which is particularly useful for reasoning about low-level implementations. However, those two approaches are problematic for manual mathematical proofs, which form the majority of our effort, as they usually add a layer of concepts between the theorem prover's basic logic and the algorithm's model.

References

- 1 Mohammad Abdulaziz and Christoph Madlener. A Formal Analysis of RANKING. In *The 14th Conference on Interactive Theorem Proving (ITP)*, 2023. doi:10.48550/arXiv.2302.13747.
- 2 Mohammad Abdulaziz, Kurt Mehlhorn, and Tobias Nipkow. Trustworthy graph algorithms (invited paper). In *The 44th International Symposium on Mathematical Foundations of Computer Science (MFCS)*, 2019. doi:10.4230/LIPIcs.MFCS.2019.1.
- 3 Mohammad Abdulaziz and Lawrence C. Paulson. An Isabelle/HOL Formalisation of Green’s Theorem. In *The 7th International Conference on Interactive Theorem Proving (ITP)*, 2016. doi:10.1007/978-3-319-43144-4_1.
- 4 Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, 1993. URL: <https://api.semanticscholar.org/CorpusID:12577796>.
- 5 Stefan Berghofer and Tobias Nipkow. Executing Higher Order Logic. In *Types for Proofs and Programs*, 2002. doi:10.1007/3-540-45842-5_2.
- 6 Ran Duan, Seth Pettie, and Hsin-Hao Su. Scaling Algorithms for Weighted Matching in General Graphs. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19, 2017*. doi:10.1137/1.9781611974782.50.
- 7 Jack Edmonds and Richard M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *J. ACM*, 19(2):248–264, April 1972. doi:10.1145/321694.321699.
- 8 L. R. FORD and D. R. FULKERSON. *Flows in Networks*. Princeton University Press, 1962. URL: <http://www.jstor.org/stable/j.ctt183q0b4>.
- 9 Harold N. Gabow. The Weighted Matching Approach to Maximum Cardinality Matching. *Fundam. Informaticae*, 2017. doi:10.3233/FI-2017-1555.
- 10 Harold N. Gabow and Robert Endre Tarjan. Faster Scaling Algorithms for Network Problems. *SIAM J. Comput.*, 1989. doi:10.1137/0218069.
- 11 Harold N. Gabow and Robert Endre Tarjan. Faster Scaling Algorithms for General Graph-Matching Problems. *J. ACM*, 1991. doi:10.1145/115234.115366.
- 12 Yon T. Gallai and Vorgelegt van G. HnjOs. Maximum-minimum sätze über graphen. *Acta Mathematica Academiae Scientiarum Hungarica*, 9:395–434, 1958. URL: <https://api.semanticscholar.org/CorpusID:123953062>.
- 13 Fabian Immler and Yong Kiam Tan. The Poincaré-Bendixson theorem in Isabelle/HOL. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020*, 2020. doi:10.1145/3372885.3373833.
- 14 Bernhard Korte and Jens Vygen. *Minimum Cost Flows*, pages 215–244. Springer Berlin Heidelberg, Berlin, Heidelberg, 2018. doi:10.1007/978-3-662-56039-6_9.
- 15 Alexander Krauss. *Automating Recursive Definitions and Termination Proofs in Higher-Order Logic*. PhD thesis, Technische Universität München, 2009.
- 16 Peter Lammich. Refinement to Imperative HOL. *Journal of Automated Reasoning*, 2019. doi:10.1007/s10817-017-9437-1.
- 17 Peter Lammich and S. Reza Sefidgar. Formalizing Network Flow Algorithms: A Refinement Approach in Isabelle/HOL. *J. Autom. Reason.*, 2019. doi:10.1007/s10817-017-9442-4.
- 18 Peter Lammich and Thomas Tuerk. Applying Data Refinement for Monadic Programs to Hopcroft’s Algorithm. In *Interactive Theorem Proving*, 2012. doi:10.1007/978-3-642-32347-8_12.
- 19 Gilbert Lee. Correctness of ford-fulkerson’s maximum flow algorithm1. *Formalized Mathematics*, 2005.
- 20 Filip Marić. Verifying Faradžev-Read Type Isomorph-Free Exhaustive Generation. In *Automated Reasoning*, 2020. doi:10.1007/978-3-030-51054-1_16.

- 21 Tobias Nipkow. Amortized Complexity Verified. In *Interactive Theorem Proving - 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015, Proceedings*, 2015. doi:10.1007/978-3-319-22102-1_21.
- 22 James Orlin. A faster strongly polynomial minimum cost flow algorithm. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing, STOC '88*, pages 377–387, New York, NY, USA, 1988. Association for Computing Machinery. doi:10.1145/62212.62249.
- 23 James B. Orlin and Ravindra K. Ahuja. New scaling algorithms for the assignment and minimum mean cycle problems. *Math. Program.*, 1992. doi:10.1007/BF01586040.
- 24 Floris van Doorn, Patrick Massot, and Oliver Nash. Formalising the h-Principle and Sphere Eversion. In *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs*, January 2023. doi:10.1145/3573105.3575688.
- 25 Niklaus Wirth. Program Development by Stepwise Refinement. *Commun. ACM*, 1971. doi:10.1145/362575.362577.

Taming Differentiable Logics with Coq Formalisation

Reynald Affeldt  

National Institute of Advanced Industrial Science and Technology (AIST), Tokyo, Japan

Alessandro Bruni  

IT-University of Copenhagen, Denmark

Ekaterina Komendantskaya  

Southampton University, UK

Heriot-Watt University, Edinburgh, UK

Natalia Ślusarz  

Heriot-Watt University, Edinburgh, UK

Kathrin Stark  

Heriot-Watt University, Edinburgh, UK

Abstract

For performance and verification in machine learning, new methods have recently been proposed that optimise learning systems to satisfy formally expressed logical properties. Among these methods, differentiable logics (DLs) are used to translate propositional or first-order formulae into loss functions deployed for optimisation in machine learning. At the same time, recent attempts to give programming language support for verification of neural networks showed that DLs can be used to compile verification properties to machine-learning backends. This situation is calling for stronger guarantees about the soundness of such compilers, the soundness and compositionality of DLs, and the differentiability and performance of the resulting loss functions. In this paper, we propose an approach to formalise existing DLs using the Mathematical Components library in the Coq proof assistant. Thanks to this formalisation, we are able to give uniform semantics to otherwise disparate DLs, give formal proofs to existing informal arguments, find errors in previous work, and provide formal proofs to missing conjectured properties. This work is meant as a stepping stone for the development of programming language support for verification of machine learning.

2012 ACM Subject Classification Theory of computation → Type theory; Theory of computation → Logic and verification; Computing methodologies → Rule learning

Keywords and phrases Machine Learning, Loss Functions, Differentiable Logics, Logic and Semantics, Interactive Theorem Proving

Digital Object Identifier 10.4230/LIPIcs.ITP.2024.4

Supplementary Material *Software (Source)*: https://github.com/ndslusarz/formal_LDL [29]
archived at `swh:1:dir:bd213b761dfc453ccfe8e785a38cffe583c98f04`

1 Introduction

This work aims to contribute to the field of formal verification of artificial intelligence, more precisely machine learning, i.e., the study of algorithms that learn statistically from data. Neural networks are the most common technical device used in machine learning. The standard learning algorithms (such as gradient descent) use a *loss function* $\mathcal{L} : \mathbb{R}^m \times \mathbb{R}^n \rightarrow \mathbb{R}$ to optimise the network's parameters (say, θ) to fit the input-output vectors given by the data in a way that the loss $\mathcal{L}(\mathbf{x}, \mathbf{y})$ is minimised. This optimisation objective is usually denoted as $\min_{\theta} \mathcal{L}(\mathbf{x}, \mathbf{y})$.



© Reynald Affeldt, Alessandro Bruni, Ekaterina Komendantskaya, Natalia Ślusarz, and Kathrin Stark; licensed under Creative Commons License CC-BY 4.0

15th International Conference on Interactive Theorem Proving (ITP 2024).

Editors: Yves Bertot, Temur Kutsia, and Michael Norrish; Article No. 4; pp. 4:1–4:19

Leibniz International Proceedings in Informatics



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

Most approaches to verification of neural networks consist of an automated procedure based on SMT solving, abstract interpretation, or branch-and-bound techniques (see, e.g., Albarghouthi’s survey [3]). Verification typically applies after training because traditional learning is purely data-driven and thus agnostic to verification properties. In contrast, *property-guided training* takes place once the verification properties are stated. More precisely, verification of neural networks consists of two parts: statement and verification of a given property, and training of the neural network that optimises the neural network’s parameters towards satisfying the given property.

However, naively or manually performed mapping of a logical property to an optimisation task results in major discrepancies (as shown by Casadio et al. [9]). This suggests the need to have tools for property-guided training. One approach is to provide programming language support for property-driven development of neural networks that involves specification, verification, and optimisation in a safe-by-construction environment. As an example, Vehicle [11, 13] provides this support in the form of a Haskell DSL, with a higher-order typed specification language, in which required neural network properties can be clearly documented, and type-driven compilation which can take care of correct-by-construction translation of properties into both the language of neural network solvers and loss functions.

To generate loss functions from a logical property, one can use *Differentiable Logics* (DLs). Well-studied *fuzzy logics* that date back to the works of Łukasiewicz and Gödel can be used as DLs [26]. Recently, both verification and machine-learning communities formulated alternative DLs such as DL2 [16] and STL [27]; the latter was shown to be more performant in optimisation tasks. These DLs are very different; for example, they do not agree on the domains of the resulting loss functions: fuzzy logics have domain $[0, 1]$, the domain of DL2 corresponds to the Lawvere quantale $(-\infty, 0]$, and STL’s domain is $(-\infty, +\infty)$ (all intervals are equipped with the usual ordering on reals). Each domain has a designated value for truth (e.g., 1 in fuzzy logics, 0 in DL2, $+\infty$ in STL) and falsity (0, $-\infty$, $-\infty$, respectively).

Vehicle uses DLs to translate logical properties into loss functions. In order to ensure the correctness of the translation, a DL needs to satisfy a number of properties:

- *Soundness*: if a property interprets as “true” in the chosen DL domain, then it is true in the boolean logic, and similarly for false.
- *Compositionality*: the translation function should preserve the structural properties, e.g., (the translation of) negation should compose with conjunction and disjunction, and (the translation of) conjunction and disjunction should satisfy the usual properties of idempotence, commutativity, and associativity.
- *Shadow-lifting*: the resulting functions should have partial derivatives that can characterise the idea of gradual improvement in training [27]. For example, a translation of a conjunction should evaluate to a higher value if the value of one of its conjuncts increases.

Unfortunately, none of the existing DLs satisfies all of these requirements [24, 27]. Therefore, future tools and compilers such as Vehicle may need to provide support for incorporating a range of DLs for different scenarios.

This conclusion brings to the forefront the need for a generic framework in which logical and geometric properties of different DLs can be formalised and proven. In this paper, we propose a unified formalisation of DLs to lay down the ground for the development of a reliable neural network verification tool. For that purpose, we will build on top of previous work that has already proposed a common presentation of DLs [24]. In order to handle the verification of translation from properties to loss functions, we use the Coq proof assistant in which numerous formalisations of logics and programming languages have been carried out. In addition, the formalisation of the properties of DLs also requires a good library

support for algebra (to handle the structural properties of DLs) as well as analysis (to handle shadow-lifting), a task for which the Mathematical Components libraries (hereafter, MATHCOMP [25]) seem well fitted.

Our contributions in this paper are as follows:

- We explain how to encode known DLs in a single generic syntax using COQ, taking advantage of dependent types and building on known techniques for logic embedding (such as intrinsic typing). The formalisation is comprehensive and extensible for future use.
- We demonstrate how to use the MATHCOMP libraries for our purpose, which includes reusable lemmas that we had to newly develop.
- As a result we are able to find and fix errors in the literature. The most prominent missing results were: soundness of STL and missing parts of the shadow-lifting proofs, both of which appear as original results in this paper.

The paper proceeds as follows. Section 2 provides further background information about property-guided training and DLs. Section 3 explains how one can define DLs in COQ using a generic encoding, including a translation function producing the semantics. Section 4 focuses on the formalisation of logical properties and soundness of DLs. In Section 5, we demonstrate the formal verification of the shadow-lifting properties of DLs. We discuss related work and conclude in Section 6.

2 Background

2.1 Property-guided training, by means of an example

Neural network properties

Given a neural network $N : \mathbb{R}^m \rightarrow \mathbb{R}^n$, the verification property usually takes the form of a Hoare triple $\forall \mathbf{x} \in \mathbb{R}^m. \mathcal{P}(\mathbf{x}) \longrightarrow \mathcal{S}(\mathbf{x})$, where \mathcal{P} and \mathcal{S} can be arbitrary properties obtained by using variables $\mathbf{x} \in \mathbb{R}^m$, constants, vector, arithmetic operations, \leq , $=$, \wedge , \vee , and \neg . Additionally, \mathcal{S} may contain the neural network N as a function.

► **Example 1** (Properties of neural networks). Given a neural network N and a vector \mathbf{v} , consider the specification that requires that for all inputs \mathbf{x} that are within ϵ distance from \mathbf{v} , the output of $N(\mathbf{x})$ should not deviate by more than δ from $N(\mathbf{v})$:

$$\forall \mathbf{x}. |\mathbf{x} - \mathbf{v}|_{L_\infty} \leq \epsilon \Rightarrow |N(\mathbf{x}) - N(\mathbf{v})|_{L_\infty} \leq \delta.$$

This property is known as ϵ - δ -robustness [9]. It can be used to avoid misclassifying images when only a few pixels are perturbed. This particular example uses the L_∞ norm: $|\mathbf{x} - \mathbf{y}|_{L_\infty} \stackrel{\text{def}}{=} \max_{i \in \{0, \dots, n-1\}} (|\mathbf{x}_i - \mathbf{y}_i|)$, where $[\mathbf{x}]_i$ stands for the i th element of \mathbf{x} .

Unfortunately, as demonstrated by Fischer et al. [16], even most accurate neural networks fail even the most natural verification properties, such as ϵ - δ -robustness. This motivated the search for better ways to train the networks.

Property-guided training

Methods for property-guided training have received considerable attention in the AI literature, as the survey [18] shows. We will only illustrate the method that was suggested by Fischer et al. [16], and refer the reader to the survey for more examples.

► **Example 2** (Generating a loss function from a logical property [16]). Recall that standard supervised learning trains a neural network N with trainable parameters θ to optimise the objective $\min_{\theta} \mathcal{L}(\mathbf{x}, \mathbf{y})$, for the loss function $\mathcal{L} : \mathbb{R}^m \times \mathbb{R}^n \rightarrow \mathbb{R}$. Generally, \mathcal{L} measures the difference between the network’s output and the given data for each input point. Examples of \mathcal{L} are cross-entropy loss or mean squared error. But now we want to train the neural network to satisfy any arbitrary property $\forall \mathbf{x}. \mathcal{P}(\mathbf{x}) \rightarrow \mathcal{S}(\mathbf{x})$. For this, we replace the above optimisation objective with

$$\min_{\theta} \left[\max_{\mathbf{x} \in \mathbb{H}_{\mathcal{P}(\mathbf{x})}} \mathcal{L}_{\mathcal{S}}(\mathbf{x}) \right]$$

where $\mathbb{H}_{\mathcal{P}(\mathbf{x})} \subseteq \mathbb{R}^m$ refines the type \mathbb{R}^m to a subset for which the property \mathcal{P} holds, and $\mathcal{L}_{\mathcal{S}} : \mathbb{R}^m \rightarrow \mathbb{R}$ is obtained by applying a suitable *interpretation function* for \mathcal{S} .

We omit the exact details of how such optimisation algorithms are defined: they are known and can be found in a suitable machine learning tutorial, for example [19]. Intuitively, the optimisation algorithm will search for $\mathbf{x} \in \mathbb{H}_{\mathcal{P}(\mathbf{x})}$ such that \mathbf{x} maximises the loss $\mathcal{L}_{\mathcal{S}}(\mathbf{x})$, in order to train the neural network parameters θ to minimise that loss. Concretely, if the property is ϵ - δ -robustness, it will look for the worst perturbation of \mathbf{v} that violates the property, and will optimise the neural network to classify that bad example correctly.

Differentiable logics for loss functions

In the above example, we did not explain how to define the interpretation function $\mathcal{L}_{\mathcal{S}}$ for an arbitrary property \mathcal{S} ; we need differentiable logics for that purpose. Fischer et al. [16] proposed one such interpretation function – called *the differentiable logic DL2*, standing for “Deep Learning with Differentiable Logics”.

► **Example 3** (Loss functions from properties in a fuzzy logic). Taking the properties from Example 1, by the Fischer et al. method we must be able to interpret the right-hand side of the implication, i.e., $|N(\mathbf{x}) - N(\mathbf{v})|_{L_{\infty}} \leq \delta$, given concrete values for ϵ , δ , a concrete vector \mathbf{v} , neural network N , and a suitable definition of the L_{∞} norm. For example, interpretation for our property in STL [27] is given by $\llbracket |N(\mathbf{x}) - N(\mathbf{v})|_{L_{\infty}} \leq \delta \rrbracket_{\text{STL}} = \delta - \llbracket N(\mathbf{x}) \rrbracket - \llbracket N(\mathbf{v}) \rrbracket_{L_{\infty}}$. On the left-hand side, the L_{∞} distance between vectors as well as N are defined in the syntax of STL; on the right-hand side, they are given by real vector arithmetic operations. Example 8 will make the relation between syntax and interpretation clear. The obtained function can be used directly for training neural networks.

We next consider our choices of DLs in details.

2.2 Differentiable logics

Ślusarz et al. [24] suggest a common syntax for all DLs, calling it the *logic of differentiable logics (LDL)*, and subsequently obtain different DLs via different interpretation functions. In the following, we summarize the syntactic and semantic features of DLs following this formulation; minor modifications will be discussed as we introduce them.

LDL syntax

LDL’s syntax consists of types and expressions (Fig. 1). Types are given by booleans, reals, vectors, indices, and a function type $\text{Fun } n \ m$; expressions are given by real numbers, vectors, vector indices, lookup operations, and functions that take real vectors as inputs. Formulae

type $\ni t ::= \text{Bool} \mid \text{Index } n \text{ for } n \in \mathbb{N} \mid \text{Real} \mid \text{Vec } n \mid \text{Fun } n \ m \text{ for } n, m \in \mathbb{N}$

$\text{exprInd} \ni i$	$::= i \in \mathbb{N}$	$\text{exprB} \ni p, p_0, \dots, p_n$	$::= \text{True} \mid \text{False}$
$\text{exprR} \ni r, r_1, r_2$	$::= r \in \mathbb{R} \mid [v]_i$		$\mid r_1 \leq r_2$
$\text{exprFun} \ni f$	$::= f \in \mathbb{R}^n \rightarrow \mathbb{R}^m$		$\mid r_1 \neq r_2$
$\text{exprVec} \ni v$	$::= v \in \mathbb{R}^n \mid f \ v$		$\mid \bigwedge_M(p_0, \dots, p_M)$
			$\mid \bigvee_M(p_0, \dots, p_M)$
			$\mid \neg p$

■ **Figure 1** Types and expressions of LDL.

are formed either via applying predicates $\leq, =$ to real expressions, by boolean values, or using logical connectives \vee, \wedge, \neg . Because STL by Varnai et al. [27] lacks associativity, conjunction and disjunction are defined as n -ary connectives so that they are the same for all DLs. Further, implication is not present in the syntax: that is due to the n -ary nature of the other connectives, which do not always allow for the implication of classical logic. Any DL with associative conjunction and disjunction will admit implication $b_1 \Rightarrow b_2$ to be defined as $\neg b_1 \vee b_2$.

We forgo the originally included quantifiers, lambda, and let expressions to obtain a simpler core language in which the three properties of interest – soundness, compositionality, and differentiability – can be studied.

Obtaining DLs via interpretation functions

To define a DL, one defines an interpretation function $\llbracket \cdot \rrbracket_{\text{DL}}$ that, given an expression in LDL, returns a function on real numbers. We introduce all DLs in a generic way and use the meta-notation $\llbracket \cdot \rrbracket_{\text{DL}}$, to refer to a range of interpretation functions, with $\text{DL} \in \{\text{Gödel}, \text{Łukasiewicz}, \text{Yager}, \text{product}, \text{DL2}, \text{STL}\}$. The boolean interpretation function $\llbracket \cdot \rrbracket_{\text{B}}$ is the obvious logical interpretation of boolean formulas, which will be useful for proving soundness later.

Table 1 shows the interpretation of all DLs. First are the four DLs based on well-known fuzzy logics: Gödel, Łukasiewicz [28], Yager, and product [26]. All fuzzy logics have the interpretation domain of $[0, 1] \subset \mathbb{R}$. Other logics have different domains: DL2 [16] has the interpretation domain $(-\infty, 0]$, and STL [27] the domain $(-\infty, +\infty)$.

The binary predicates \leq and $=$ are defined in a way that ensures that they are interpreted within the chosen real interval for the given DL. The definitions of logical connectives \bigvee_M , \bigwedge_M , and \neg are taken directly from the related papers that define the given DLs. Note that we reformulate \bigvee_M and \bigwedge_M for all DLs as n -ary connectives, however, only STL had n -ary connectives originally.

2.3 Properties of DLs

Soundness

There is no consensus in the DL literature on how or whether to state soundness: for example, STL came without any soundness statement. For the sake of generic formalisation of all DLs, we propose the following definition of soundness, which generalises soundness as defined in DL2 and fuzzy logics [16, 26].

■ **Table 1** Interpretation function $\llbracket \cdot \rrbracket_{DL}$, which extends naturally to sequences of expressions as well as interpretation function $\llbracket \cdot \rrbracket_B$, which is a structural interpretation of boolean formulas.

‡: Negation is undefined in the sense that it is not defined as a structural transformation of the syntax of formulas. It is implemented at the level of atomic comparison only, and negation on composite formulas is provided as syntactic sugar [16, Sect. 3]. For example, consider $\llbracket 3 = 3 \rrbracket_{DL2} = 0$. In DL2, $\llbracket \neg(3 = 3) \rrbracket_{DL2}$ is not defined in term of $\llbracket 3 = 3 \rrbracket_{DL2} = 0$, but DL2 provides an interpretation for the symbol \neq separately.

	$\llbracket \bigwedge_M s \rrbracket$	$\llbracket \bigvee_M s \rrbracket$	$\llbracket \neg e \rrbracket$	
Gödel	$\min \llbracket s \rrbracket_G$	$\max \llbracket s \rrbracket_G$	$1 - \llbracket e \rrbracket_G$	
Łukasiewicz	$\max \left[\sum_{a \in \llbracket s \rrbracket_L} a - s + 1, 0 \right]$	$\min \left[\sum_{a \in \llbracket s \rrbracket_L} a, 1 \right]$	$1 - \llbracket e \rrbracket_L$	
Yager	$\max \left[1 - \left(\sum_{a \in \llbracket s \rrbracket_Y} (1 - a)^p \right)^{1/p}, 0 \right]$	$\min \left[\left(\sum_{a \in \llbracket s \rrbracket_Y} a^p \right)^{1/p}, 1 \right]$	$1 - \llbracket e \rrbracket_Y$	
product	$\prod_{a \in \llbracket s \rrbracket_P} a$	$\text{fold } (\lambda x y . x + y - xy) 0 \llbracket s \rrbracket_P$	$1 - \llbracket e \rrbracket_P$	
DL2	$\sum_{a \in \llbracket s \rrbracket_{DL2}} a$	$(-1)^{ s +1} \cdot \prod_{a \in \llbracket s \rrbracket_{DL2}} a$	undefined†	
STL	$\text{and}_S \llbracket s \rrbracket_{STL}$	$\text{or}_S \llbracket s \rrbracket_{STL}$	$-\llbracket e \rrbracket_{STL}$	
Bool	$\bigwedge_M \llbracket s \rrbracket_B$	$\bigvee_M \llbracket s \rrbracket_B$	$\neg \llbracket e \rrbracket_B$	
	$\llbracket e_1 = e_2 \rrbracket$	$\llbracket e_1 \leq e_2 \rrbracket$	$\llbracket True \rrbracket$	$\llbracket False \rrbracket$
fuzzy	if $\llbracket e_1 \rrbracket = -\llbracket e_2 \rrbracket$ then $\llbracket e_1 \rrbracket = \llbracket e_2 \rrbracket$ else $\max \left[1 - \left \frac{\llbracket e_1 \rrbracket - \llbracket e_2 \rrbracket}{\llbracket e_1 \rrbracket + \llbracket e_2 \rrbracket} \right , 0 \right]$	if $\llbracket e_1 \rrbracket = -\llbracket e_2 \rrbracket$ then $\llbracket e_1 \rrbracket \leq \llbracket e_2 \rrbracket$ else $\max \left[1 - \max \left[\frac{\llbracket e_1 \rrbracket - \llbracket e_2 \rrbracket}{\llbracket e_1 \rrbracket + \llbracket e_2 \rrbracket}, 0 \right], 0 \right]$	1	0
DL2	$-\llbracket \llbracket e_2 \rrbracket_{DL2} - \llbracket e_1 \rrbracket_{DL2} \rrbracket$	$-\max \llbracket \llbracket e_1 \rrbracket_{DL2} - \llbracket e_2 \rrbracket_{DL2}, 0 \rrbracket$	0	$-\infty$
STL	$-\llbracket \llbracket e_2 \rrbracket_{STL} - \llbracket e_1 \rrbracket_{STL} \rrbracket$	$\llbracket \llbracket e_2 \rrbracket_{STL} - \llbracket e_1 \rrbracket_{STL} \rrbracket$	$+\infty$	$-\infty$
Bool	$\llbracket e_1 \rrbracket_B = \llbracket e_2 \rrbracket_B$	$\llbracket e_1 \rrbracket_B \leq \llbracket e_2 \rrbracket_B$	<i>True</i>	<i>False</i>

$$\text{and}_S [a_1, \dots, a_M] = \begin{cases} \frac{\sum_i a_{\min} e^{\tilde{a}_i} e^{\nu \tilde{a}_i}}{\sum_i e^{\nu \tilde{a}_i}} & \text{if } a_{\min} < 0 \\ \frac{\sum_i a_i e^{-\nu \tilde{a}_i}}{\sum_i e^{-\nu \tilde{a}_i}} & \text{if } a_{\min} > 0 \\ 0 & \text{if } a_{\min} = 0 \end{cases} \quad \text{where} \quad \begin{aligned} \nu &\in \mathbb{R}^+ \text{ (constant)} \\ a_{\min} &= \min [a_1, \dots, a_M] \\ \tilde{a}_i &= \frac{a_i - a_{\min}}{a_{\min}} \end{aligned}$$

or_S is analogous to and_S

► **Definition 4** (Soundness). *Given a DL, an expression e , and a boolean value $b \in \{True, False\}$, the DL is sound if*

$$\llbracket e \rrbracket_{DL} = \llbracket b \rrbracket_{DL} \implies \llbracket e \rrbracket_B = b.$$

Note that not all DLs are sound. For example, one of the oldest fuzzy logics by Łukasiewicz [28] is known to be unsound. Table 2 summarises all known soundness results. Note that prior to this paper, soundness of STL was not known. Here, we obtain the result with some restrictions, see Sect. 4.

Compositionality

We define idempotence, associativity, and commutativity of interpretation functions for \bigwedge_M and analogously \bigvee_M :

► **Definition 5** (Commutativity, idempotence, and associativity of \bigwedge_M). *Given a DL, the interpretation function of conjunction is commutative if for any permutation π of the integers $i \in \{1, \dots, M\}$ we have*

$$\left[\left[\bigwedge_M (p_0, \dots, p_M) \right] \right]_{DL} = \left[\left[\bigwedge_M (p_{\pi(0)}, \dots, p_{\pi(M)}) \right] \right]_{DL}.$$

■ **Table 2** Properties of the different DLs formalised in this paper [29]. We distinguish previously known proofs that we mechanise from previously known results published with incomplete or semi-formal proofs (yellow) and new results (orange).

†: For DL2 and STL, we prove soundness of the negation-free fragment of LDL; negation is undefined for DL2, and STL is not sound for the full fragment.

Properties:	Negation	Idempotence	Commutat.	Associativ.	Soundness	Shadow-lifting
Gödel	yes	yes	yes	yes	yes	no
Łukasiewicz	yes	no	yes	yes	no	no
Yager	yes	no	yes	yes	no	no
product	yes	no	yes	yes	yes	yes
DL2	no	no	yes	yes	yes [†]	yes
STL	yes	yes	yes	no	yes [†]	yes

It is idempotent and associative if we have

$$\begin{aligned} \llbracket \bigwedge_M (p, \dots, p) \rrbracket_{DL} &= \llbracket p \rrbracket_{DL}, \\ \llbracket \bigwedge_M (\bigwedge_M (p_0, p_1), p_2) \rrbracket_{DL} &= \llbracket \bigwedge_M (p_0, \bigwedge_M (p_1, p_2)) \rrbracket_{DL}. \end{aligned}$$

Table 2 shows which DLs satisfy which logical properties. Finally, as already illustrated in Sect. 2.1, negation can be problematic in some DLs. For example, DL2 does not give a direct interpretation for negation, as its domain is asymmetric. We will see in Sect. 4 that negation also causes problems with the soundness of STL.

Differentiability

Varnai et al. [27] introduce three properties in this category: weak smoothness, scale-invariance, and shadow-lifting. The latter is the most important as it accounts for gradual improvement in training. We only consider shadow-lifting here as it is the most complex of those properties and leave the remaining properties to future work.

► **Definition 6** (Shadow-lifting property [27]). *The DL satisfies the shadow-lifting property if, for any $\llbracket p \rrbracket_{DL} \neq 0$:*

$$\frac{\partial \llbracket \bigwedge_M (p_0, \dots, p_i, \dots, p_M) \rrbracket_{DL}}{\partial \llbracket p_i \rrbracket_{DL}} \bigg|_{p_j = p \text{ where } i \neq j} > 0$$

holds for all $0 \leq i \leq M$, where ∂ denotes partial differentiation.

Notice that classical conjunction does not satisfy the property of shadow-lifting: no matter how “true” the value of p_2 is, if p_1 is false, then $p_1 \wedge p_2$ will remain false. Likewise, all DLs that use min or max to define conjunction will fail shadow-lifting.

Shadow-lifting was originally defined for conjunction only, as STL had no disjunction. In our formalisation, we could, in principle, extend shadow-lifting to disjunction. However, we left this incremental extension for future work.

Summary of results

Table 2 summarises all properties covered in our COQ formalisation and highlights the ones for which we provide original proofs. In our development, we provided several missing results, most prominently, the soundness of STL and missing parts of the shadow-lifting proofs. Note that the formalisation further revealed some errors:

► **Example 7** (Discrepancies in pen and paper proofs). While doing the COQ formalisation, we found two sources of errors in our pen and paper proofs [24] as well as in [27]. Firstly, the work in [24] concerned completing results of Table 2 in the uniform notation of LDL. Many proofs were analogous and it was easy to overlook the rare cases when the proof could not be completed by analogy with existing proofs. For example, we tried to prove the soundness of Yager by analogy with other fuzzy logics overlooking that Yager is not sound. Indeed Yager is a generalisation of Łukasiewicz logic which in itself is not sound. The COQ formalisation revealed all such errors.

The second source of errors came from extension of fuzzy logics with comparison operators. The interpretation of comparisons needed to be scaled between 0 and 1, and it seemed obvious that the scaling was done correctly. Therefore we did not provide any proofs concerning the interval properties of these operations. In contrast the soundness proofs in COQ required us to prove that the fuzzy interpretation functions always return values within the interval $[0, 1]$.

Thirdly, while a sketch of the shadow-lifting proof was provided in [27], it was incomplete and did not mention either the existence of other cases as well as the reliance of the proof on L'Hôpital's rule.

No DL satisfies all desirable properties – for example, Gödel is sound, idempotent, associative, and commutative, but it is not shadow-lifting. On the other hand, Łukasiewicz is not sound, STL not associative, and while DL2 is sound and shadow-lifting, it fails idempotence, and its negation is not compositional because it is not structural. Varnai et al. [27] have proven that it is impossible for any DL to be idempotent, associative, and shadow-lifting at the same time.

When one has to make a choice of a DL, different considerations may influence that choice. Soundness and shadow-lifting are strictly desirable, thus Gödel, Łukasiewicz and Yager are probably less desirable than the rest, even if some of them have nice logical properties. However, given soundness and shadow-lifting, the choice between logical properties is less clear. For example, one can imagine a scenario when the specification language avoids negation, and in a style of substructural logics, treats differently p and $p \wedge p$ and thus sacrifices idempotence; in this case, DL2 may provide an ideal translation function.

3 An encoding of DLs in Coq

As discussed, LDL aims at defining all DLs in a generic and extendable way, using uniform syntactic conventions. In this section, we start by highlighting the generic features of our formalisation.

3.1 Encoding of the syntax of types and expressions

The encoding of the LDL types is the matter of declaring the following inductive type:

```
Inductive flag := def | undef.

Inductive ldl_type :=
  Bool_T of flag | Index_T of nat | Real_T | Vector_T of nat | Fun_T of nat & nat.
```

This corresponds to the informal syntax of Fig. 1 with the difference that we refine the Boolean type with a (`flag`) to signify whether negation is defined in the logic.

As for LDL expressions, their encoding is displayed in Fig. 2. It is an inductive type indexed by `ldl_type` so that the resulting syntax is intrinsically-typed: one cannot write ill-typed expressions. The COQ inductive type matches the informal syntax already explained

in Fig. 1. Real expressions (line 6) use a type `R` of type `realType` coming from MATHCOMP-ANALYSIS [1] that represents real numbers. Boolean expressions (line 7) use the native COQ type `bool`. Indices embed an *ordinal* from MATHCOMP (line 8). More specifically, `'I_n` is the type of natural number smaller than `n`. Similarly, vectors just reflect MATHCOMP tuples (line 9). For defining n -ary connectives `ldl_and` and `ldl_or`, we use polymorphic lists (of type `seq`). Yet, to ease reading, we will use notation such as `a /\ b` to denote binary conjunction in the following. For a generic definition of the syntax, we need to allow for the case of DLs in which negation is not defined (in fact, DL2). The additional argument `Bool_T_def` in the constructor `ldl_not` (line 12) signifies that the negation is defined. The constructor `ldl_cmp` (line 13) is for binary comparison operators over the real numbers; hereafter, we will use notations such as `<=` for the comparison corresponding to \leq instead of “`ldl_cmp cmp_le`” to ease reading. As for the last constructors, they are respectively for functions, their application, and lookups, as per Fig. 1.

```

1  Definition Bool_T_undef := Bool_T undef.
2  Definition Bool_T_def := Bool_T def.
3  Inductive comparison := cmp_le | cmp_eq.
4
5  Inductive expr : ldl_type -> Type :=
6  | ldl_real   : R -> expr Real_T
7  | ldl_bool   : forall p, bool -> expr (Bool_T p)
8  | ldl_idx    : forall n, 'I_n -> expr (Index_T n)
9  | ldl_vec    : forall n, n.-tuple R -> expr (Vector_T n)
10 | ldl_and    : forall x, seq (expr (Bool_T x)) -> expr (Bool_T x)
11 | ldl_or     : forall x, seq (expr (Bool_T x)) -> expr (Bool_T x)
12 | ldl_not    : expr Bool_T_def -> expr Bool_T_def
13 | ldl_cmp    : forall x, comparison -> expr Real_T -> expr (Bool_T x)
14 | ldl_fun    : forall n m, (n.-tuple R -> m.-tuple R) -> expr (Fun_T n m)
15 | ldl_app    : forall n m, expr (Fun_T n m) -> expr (Vector_T n) -> expr (Vector_T m)
16 | ldl_lookup : forall n, expr (Vector_T n) -> expr (Index_T n) -> expr Real_T.

```

■ Figure 2 LDL syntax in COQ.

3.2 Encoding of the interpretation function

We now proceed to the translation function that interprets the syntax. Types are mapped to their obvious semantics:

```

1  Definition type_translation (t : ldl_type) : Type :=
2  match t with
3  | Bool_T x => R
4  | Real_T => R
5  | Vector_T n => n.-tuple R
6  | Index_T n => 'I_n
7  | Fun_T n m => n.-tuple R -> m.-tuple R
8  end.

```

In particular, booleans are mapped to `R` of type `realType`, the type of real numbers. This translation accommodates the many interpretations of the DLs: the domain $[0, 1]$ for fuzzy logic, $(-\infty, 0]$ for DL2, and $(-\infty, +\infty)$ for STL. For DL2 and STL, we also provide an alternative translation function `ereal_type_translation` that maps booleans to real numbers extended with $-\infty$ and $+\infty$, i.e., the type `\bar{R}` of extended real numbers as provided by MATHCOMP-ANALYSIS. We use an invariant to restrict the range of the interpretation function accordingly.

4:10 Taming Differentiable Logics with Coq Formalisation

Each logic requires a separate interpretation function (as explained in Table 1). Here, we only show an excerpt of the translation function for STL, namely the cases for conjunction (constructor `ldl_and`), negation (notation `~`), and comparison (notation `<=`) of STL in Fig. 3.

```

Fixpoint stl_translation {t} (e : expr t) : type_translation t :=
  match e in expr t return type_translation t with
  | ldl_and _ (e0 :: s) => let A      := map stl_translation s in
                           let a0    := stl_translation e0 in
                           let a_min := \big[minr/a0]_(i <- A) i in
                           if a_min < 0 then stl_and_lt0 (a0 :: A) else
                           if a_min > 0 then stl_and_gt0 (a0 :: A) else
                           0
  | ~ E1                => - {[ E1 ]}
  | E1 <= E2            => {[ E2 ]} - {[ E1 ]}
  ... (* see [29] for omitted connectives *)
end where "{[ e ]}" := (stl_translation e).

```

■ **Figure 3** Excerpt of the semantics of STL: conjunction, negation, and comparison. (See Fig. 4 for intermediate definitions `stl_and_lt0` and `stl_and_gt0`.)

The case for conjunction of STL is the most complex in our formalisation because dealing formally with it requires the theories of exponentiation (`expR`), big sums (`\sum`), inverses (`^-1`), and minima (`minr`). To reduce the clutter, we define the cases for $a_{\min} > 0$ and $a_{\min} < 0$ separately as `stl_and_gt0` and `stl_and_lt0` reproduced in Fig. 4. This will allow us to state intermediate lemmas about sub-expressions.

<pre> Definition sumR a := \sum_(i <- a) i. Definition min_dev {R : realType} (x:R) (a:seq R) : R := let r := \big[minr/x]_(i <- a) i in (x - r) * r^-1. Definition stl_and_gt0 (a : seq R) := sumR (map (fun x => x * expR(-nu * min_dev x a)) a) * (sumR (map (fun x => expR(-nu * min_dev x a)) a))^-1. Definition stl_and_lt0 (a : seq R) := sumR (map (fun x => (\big[minr/x]_(i <- a) i) * expR (min_dev x a) * expR(nu * min_dev x a)) a) * (sumR (map (fun x => expR(nu * min_dev x a)) a))^-1. </pre>	$a_{\min} = \min [a_1, \dots, a_M]$ $\tilde{a}_i = \frac{a_i - a_{\min}}{a_{\min}}$ $\nu \in \mathbb{R}^+ \quad (\text{constant})$ $\frac{\sum_i a_i e^{-\nu \tilde{a}_i}}{\sum_i e^{-\nu \tilde{a}_i}} \quad (\text{case } a_{\min} > 0)$ $\frac{\sum_i a_{\min} e^{\tilde{a}_i} e^{\nu \tilde{a}_i}}{\sum_i e^{\nu \tilde{a}_i}} \quad (\text{case } a_{\min} < 0)$
---	---

■ **Figure 4** Intermediate definitions to define the conjunction of STL. (The right subfigure reproduces part of Table 1 for reading convenience.)

As a first application of our encoding, we can already formalise our running example, with the advantage of encoding it once for all DLs:

► **Example 8.** Taking the interpretation task of Example 3, we first give a suitable definition for L_∞ norm (`ldl_norm_infty`) and vector subtraction (`ldl_vec_sub`). One only needs to call the defined interpretation function to encode the loss function of Example 3:

```

Context (eps delta : @expr R Real_T) (f : @expr R (Fun_T n.+1 m.+1))
      (v : @expr R (Vector_T n.+1)) (x : @expr R (Vector_T n.+1)).

Definition eps_delta_robust : expr Bool_T_undef :=
  ldl_lookup
    (ldl_app (ldl_norm_infty m) (ldl_vec_sub (ldl_app f x) (ldl_app f v)))
    (ldl_idx ord0) <= delta.

```

4 Logical properties and soundness of DLs

4.1 Logical properties of DLs

The logical properties of DLs are idempotence, commutativity, and associativity. Not all DLs have the same properties as we saw earlier (Table 2). Proving the logical properties essentially amounts to showing that the semantic interpretation does have them. For example, the conjunction of DL2 being interpreted as addition on reals inherits its associativity directly from the properties of real numbers, and as a consequence, its proofs are one-liners, e.g.:

```
Lemma dl2_andA (e1 e2 e3 : expr Bool_T_undef) :
  [[ e1 /\ (e2 /\ e3) ]]_dl2 = [[ (e1 /\ e2) /\ e3 ]]_dl2.
Proof. by rewrite /=sumR ?big_cons ?big_nil !addr0 addrA. Qed.
```

In contrast, for Yager and STL, the proofs are more demanding. For example, the associativity for Yager, though stated analogously,

```
Theorem Yager_andA (e1 e2 e3 : expr Bool_T_def) :
  0 < p -> [[ (e1 /\ e2) /\ e3 ]]_Yager = [[ e1 /\ (e2 /\ e3) ]]_Yager.
```

consists of about 100 lines of code. This is because in this case, the interpretation relies on the power function of MATHCOMP-ANALYSIS whose properties are more technical. Yet, we could put the automatic tactics available with MATHCOMP such as `lra` [22, 23] to good use.

4.2 Soundness of DLs

We now address the topic of formalising the soundness results of Table 2.

Soundness for closed interval DLs

For fuzzy DLs and, more generally, closed interval DLs there is a clear consensus on how to define soundness: we generalised it in Definition 4. It boils down to taking the least and greatest elements in the given real interval as interpretations for *False* and *True*, respectively. In COQ, the statement of soundness for Gödel and product is as follows:

```
Lemma soundness (e : expr Bool_T_def) b :
  [[ e ]]_1 = [[ ldl_bool _ b ]]_1 -> [[ e ]]_B = b.
```

This is a direct paraphrase of the pencil-and-paper Definition 4. The proofs proceed by induction and require inversion lemmas, which we will discuss later in this section.

Soundness for DLs with open intervals

When a DL's domain of interpretation is given by an open interval, which is the case for DL2 and STL, there is no clear consensus in the literature on defining or proving soundness. We will illustrate the problems that arise using STL and following previous work [24]. The first question is how to state soundness. The easiest choice is to simply add $-\infty$ and $+\infty$ as constants to the domain, and keep the soundness statement of Definition 4. However, because no formula in the language evaluates to $-\infty$ or $+\infty$, such a soundness proof is vacuous. Note that Definition 4 did not cause this problem for fuzzy DLs because there were formulae in the language that evaluated to bottom and top values: take for example $\llbracket 3 = 3 \rrbracket_P = 1$.

Alternatively, one may keep the open interval intact and simply re-define soundness in terms of intervals: if the interpretation of the formula e is greater or equal to 0, then $\llbracket e \rrbracket_B = \text{True}$ else $\llbracket e \rrbracket_B = \text{False}$. However, this solution triggers a different problem: negation is no longer sound. Indeed, if $\llbracket 3 = 3 \rrbracket_{\text{STL}} = 0$ means the formula is true, then the same can be said about $\llbracket \neg(3 = 3) \rrbracket_{\text{STL}} = 0$.

4:12 Taming Differentiable Logics with Coq Formalisation

One could think of a solution excluding 0 from the interval $(-\infty, +\infty)$ altogether, but that complicates the interpretation of comparisons and creates a point in the interval at which the resulting function is not differentiable, which damages shadow-lifting.

Coq formalisation for logics with open intervals

For the reasons explained above, we remove negation from STL and use intervals to define the truth:

```
Definition is_stl b (x : R) := if b then x >= 0 else x < 0.
```

This results in the following soundness statement:

```
Lemma stl_soundness (e : expr Bool_T_undef) b :  
  is_stl b (nu.-[[ e ]]_stl) -> [[ e ]]_B = b.
```

The flag `Bool_T_undef` in $(e : \text{expr Bool_T_undef})$ signifies that the proof omits the case that uses negation. We write `nu.-[[e]]_stl` as notation for interpretation of STL (and similar notation for the remaining DLs).

The soundness proof proceeds by induction on the structure of the interpretation function. Because of the extensive use of dependent types in our formalisation we need a custom dependent induction principle. The most interesting cases are those for conjunction and disjunction, which need special inversion lemmas. Here is one example:

```
Lemma stl_nary_inversion_andE1 (s : seq (expr Bool_T_undef)) :  
  is_stl true (nu.-[[ ldl_and s ]]_stl) ->  
    forall i, i < size s -> is_stl true (nu.-[[ nth (ldl_bool pos false) s i ]]_stl).
```

Our formalisation faced a minor technical problem: if we are to comply with the generic DL syntax defined in Fig. 1, we need to interpret constants *True* and *False* present in the language. We therefore propose two alternative interpretations for DL2 and STL: one that works on extended reals (with added constants $-\infty, +\infty$) and maps *True* and *False* to the top and bottom elements of the respective domains, and one that resolves this discrepancy by choosing arbitrary interpretations for *True* and *False* that satisfy all the properties of interest for our study. In the latter case, for DL2 we choose to interpret *True* as 0 and *False* as -1 , and for STL we choose to interpret *True* as 1 and *False* as -1 . In all these four cases we show that the resulting logic satisfies the soundness property stated above. Adding $-\infty$ and $+\infty$ has repercussions when proving the geometric properties of the logics, as we show later in Sect. 5. If it were not for considerations of using the generic syntax for all DLs, *True* and *False* could be removed from the STL syntax altogether, without damaging the main results.

Lessons learnt

Soundness for DLs with open intervals was the first real challenge that this formalisation faced. Having no plausible solution in the field, being able to use COQ to experiment with different soundness statements and see their effect on proofs was extremely rewarding. Overall, we proved three different versions of STL soundness (one for “vacuous proofs”, which we do not present here); and we intend to use this formalisation to experiment further with STL. In particular, finding an alternative approach to negation, e.g., using “approximate 0”, is now within our reach. The currently presented approach is the first proof of soundness for any fragment of STL, it already covers formalisation of problems such as the ϵ - δ -robustness; and we attribute this intermediate success to the assistance of the COQ formalisation.

5 Differentiability: shadow-lifting

It was Varnai et al. who provided for STL the pencil-and-paper proof of shadow-lifting [27, Sect. V] (along with the definition of the STL conjunction). This section formalises this result and actually completes it since the original proof only covers one of the two non-trivial cases. The main technical aspect of the proof is high-school level mathematics: an application of L'Hôpital's rule, which was not yet available in MATHCOMP-ANALYSIS.

DL2 and the product DL also trivially enjoy shadow-lifting. In the following, we will therefore start by formalising the latter DLs, then formalising L'Hôpital's rule, and finally provide an overview of the missing part of Varnai et al.'s proof of shadow-lifting for STL. Note that the logics Gödel, Łukasiewicz, Yager fail shadow-lifting as they are not differentiable everywhere, due to their use of min or max to define conjunction.

5.1 formalisation of shadow-lifting

As seen in Sect. 2, shadow-lifting is defined in terms of partial derivatives, for which there was however no theory yet in MATHCOMP-ANALYSIS. They can be easily defined on the model of derivatives [1, file `derive.v`]. First, we define *error vectors* as row vectors (type `'rV[R]_k` where `R` is some ring) that are 0 everywhere except at one coordinate `i`:

```
Definition err_vec {R : ringType} (i : 'I_n.+1) : 'rV[R]_n.+1 :=
  \row_(j < n.+1) (i == j)%:R.
```

The notation `%:R` injects a natural number into a ring; note that here the boolean equality (notation: `==`) is implicitly coerced to a natural number. Then, given a function `f` that takes as input a row vector, we define a function `partial` that given a row vector `a` and an index `i` returns the limit $\lim_{\substack{h \rightarrow 0 \\ h \neq 0}} \frac{f(a+h\text{err_vec } i) - f(a)}{h}$. Put formally:

```
Definition partial {R} {n} (f : 'rV[R]_n.+1 -> R) (a : 'rV[R]_n.+1) i :=
  lim (h^-1 * (f (a + h * err_vec i) - f a) @[h --> 0^']).
```

In this syntax, `0^'` represents the deleted neighborhood of 0, and `lim g @ F` represents the limit of the function `g` at the filter `F` [2]. The notation `*`: represents scaling but is equivalent to the multiplication of real numbers here. Hereafter, we use the COQ notation `d f '/d i` for `partial f i`.

Using partial derivatives, the definition of shadow-lifting (Definition 6) translates directly into COQ:

```
Definition shadow_lifting {R : realType} n (f : 'rV_n.+1 -> R) :=
  forall p, p > 0 -> forall i, ('d f '/d i) (const_mx p) > 0.
```

The `const_mx` function comes from MATHCOMP's matrix theory and represents a matrix where all coefficients are the given constant; we use it to implement the restriction " $p_j = p$ " seen in Definition 6.

5.2 Shadow-lifting for DL2 and product

The proof of shadow-lifting for DL2 and product DLs provides an easy illustration of the use of the definition of the previous section (Sect. 5.1).

For DL2, the first thing to observe is that the semantics of a vector of real numbers can simply be written as an iterated sum using the notation ```_` to address elements of row-vectors:

```
Definition dl2_and {R : fieldType} {n} (v : 'rV[R]_n) := \sum_(i < n) v ``_ i.
```


Shadow-lifting for DL2 really just amounts to checking that the partial derivatives of the function $\vec{v} \mapsto \sum_{j < |\vec{v}|} \vec{v}_j$ are 1, i.e., considering vectors of size $M + 1$:

```
Lemma shadowlifting_dl2_andE (p : R) : p > 0 ->
  forall i, ('d (@dl2_and R M.+1) '/d i) (const_mx p) = 1.
```

Since the partial derivatives are all positive, DL2 satisfies the `shadow_lifting` predicate, see [24, file `dl2.v`].

Similarly, we observe for the product DL that the semantics of a vector is the function $\vec{v} \mapsto \prod_{j < |\vec{v}|} \vec{v}_j$ whose partial derivatives are p^M , which is positive:

```
Lemma shadowlifting_product_andE p : p > 0 ->
  forall i, ('d (@product_and R M.+1) '/d i) (const_mx p) = p ^+ M.
```

5.3 formalisation of L'Hôpital's rule using MathComp-Analysis

As indicated in the introduction of this section, the key technical lemma to prove shadow-lifting for STL is L'Hôpital's rule, that we show how to formalise in `MATHCOMP-ANALYSIS`. As a reminder, here follows one of L'Hôpital's rules:

► **Theorem 9** (L'Hôpital's rule). *Let $f, g : \mathbb{R} \rightarrow \mathbb{R}$ be functions differentiable on an open interval U except possibly at one point a . Suppose that $\forall x \in U, x \neq a$, we have $g'(x) \neq 0$. If it holds that $f(a) = g(a) = 0$, then if $\lim_{x \rightarrow a^+} \frac{f'(x)}{g'(x)} = l$ for some real number l , then $\lim_{x \rightarrow a^+} \frac{f(x)}{g(x)} = l$.*

It can be formally stated with `MATHCOMP-ANALYSIS` using: (a) the relation `is_derive` (between a function and its derivative at some point: the `1` appearing in the `is_derive` expression is the direction of the derivative, which is `1` for real number-valued functions) and (b) the *right filter* `a^'+` (i.e., the filter of neighborhoods of `a` intersected with $(a, +\infty)$). We slightly generalize the above statement by considering a neighborhood `U` of `a` instead of an open (line 2) and by having the derivative of `g` non-zero “near” `a` (line 6) [2, Sect. 3.2]:

```
1 Context {R : realType}.
2 Variables (f df g dg : R -> R) (a : R) (U : set R) (Ua : nbhs a U).
3 Hypotheses (fdf : forall x, x \in U -> is_derive x 1 f (df x))
4             (gdg : forall x, x \in U -> is_derive x 1 g (dg x)).
5 Hypotheses (fa0 : f a = 0) (ga0 : g a = 0)
6             (cdg : \forall x \nearrow a^', dg x != 0).
7 Lemma lhospital_right (l : R) :
8   df x / dg x @[x --> a^'+] --> l -> f x / g x @[x --> a^'+] --> l.
```

The proof is textbook, relying in particular on Cauchy's Mean Value Theorem (MVT), the proof of which can be derived from the already available MVT, see [24, file `st1.v`]. Note that we also need the variant for the left filters.

5.4 Shadow-lifting for STL

Compared with DL2 and product, the conjunction of STL (and_S in Table 1) is much more involved: it consists of two non-trivial cases (marked as $a_{\min} < 0$ and $a_{\min} > 0$ in Table 1) whose computation requires summations of exponentials of deviations. Varnai et al. provide a proof sketch for the case $a_{\min} > 0$ [27, Sect. V] which we have successfully formalised, using in particular L'Hôpital's rule from the previous section. Below we explain the formalisation of the other case $a_{\min} < 0$ that Varnai et al. did not treat.

The case $a_{\min} < 0$ actually refers to the semantics provided by the function `st1_and_lt0` already presented in Fig. 4. The positive limit we are looking for is actually $\frac{1}{M+1}$ (where $M + 1$ is the size of vectors), i.e., our goal is to prove formally (the notation \circ is for function composition):

Lemma `shadowlifting_stl_and_lt0` ($p : \mathbb{R}$) : $p > 0 \rightarrow \text{forall } i,$
 $(\text{'d } (\text{stl_and_lt0 } \backslash \circ \text{ seq_of_rV } \text{'d } i) (\text{const_mx } p) = M.+1\%:\mathbb{R}^{-1}.$

This boils down to proving the existence of the limit “from below” and “from above”. The “from below” case consists in the following convergence lemma:

Lemma `shadowlifting_stl_and_lt0_cvg_at_left` ($p : \mathbb{R}$) $i : p > 0 \rightarrow$
 $h^{-1} * (\text{stl_and_lt0 } (\text{seq_of_rV } (\text{const_mx } p + h * \text{err_vec } i)) -$
 $\text{stl_and_lt0 } (\text{seq_of_rV } (\text{const_mx } p))) @ [h \rightarrow 0^{-}] \rightarrow M.+1\%:\mathbb{R}^{-1}.$

For the sake of clarity, let us switch to standard mathematical notations and assume without loss of generality that i is actually M . By mere algebraic transformations (using MATHCOMP’s algebra theory), the goal can be turned into a sum of two limits:

$$\begin{aligned} & \lim_{h \rightarrow 0^{-}} \frac{[\bigwedge_M(p, \dots, p, p+h)]_{\text{STL}} - [\bigwedge_M(p, \dots, p)]_{\text{STL}}}{h} \\ &= \lim_{h \rightarrow 0^{-}} \frac{1}{h} \left(\frac{(p+h)M e^{\frac{-h}{p+h}} e^{\nu \frac{-h}{p+h}} + p+h}{M e^{\nu \frac{-h}{p+h}} + 1} - p \right) \quad \text{by definition (see Table 1)} \\ &= \underbrace{\lim_{h \rightarrow 0^{-}} \frac{h}{h(M + e^{\nu \frac{h}{p+h}})}}_{(a)} + \underbrace{\lim_{h \rightarrow 0^{-}} \frac{M(p+h)e^{\frac{-h}{p+h}} - pM}{h(M + e^{\nu \frac{h}{p+h}})}}_{(b)} \quad \text{by simplification} \end{aligned}$$

We can show directly that $(a) = \frac{1}{M+1}$ but the computation of (b) requires L’Hôpital’s rule:

$$\begin{aligned} (b) &= \lim_{h \rightarrow 0^{-}} \frac{\frac{hM e^{\frac{-h}{p+h}}}{p+h}}{e^{\nu \frac{-h}{p+h}} + h e^{\nu \frac{-h}{p+h}} \left(\frac{\nu}{p+h} - \frac{h\nu}{(p+h)^2} \right) + M} \\ &= \lim_{h \rightarrow 0^{-}} h \lim_{h \rightarrow 0^{-}} \frac{M e^{\frac{-h}{p+h}}}{p+h} \lim_{h \rightarrow 0^{-}} \frac{1}{e^{\nu \frac{-h}{p+h}} + h e^{\nu \frac{-h}{p+h}} \left(\frac{\nu}{p+h} - \frac{h\nu}{(p+h)^2} \right) + M} \\ &= 0 \cdot \frac{M}{p} \cdot \frac{1}{1+M} = 0 \end{aligned}$$

Barring the necessity of finding the most convenient breakdown of the limit in the two penultimate steps, this proof is arguably mathematically straightforward. The corresponding mechanised proof is however significantly less trivial than in the cases of product and DL2 (Sect. 5.2): length-wise the first tentative formal proof we wrote was an order of magnitude larger.

Proving the “from above” above consists of a simpler but similar argument:

$$\begin{aligned} & \lim_{h \rightarrow 0^{+}} \frac{[\bigwedge_M(p, \dots, p, p+h)]_{\text{STL}} - [\bigwedge_M(p, \dots, p)]_{\text{STL}}}{h} = \lim_{h \rightarrow 0^{+}} \frac{1}{h} \left(\frac{pM + e^{\frac{h}{p}} e^{\frac{\nu h}{p}}}{M + e^{\frac{h}{p}}} - p \right) \\ &= \lim_{h \rightarrow 0^{+}} \frac{1}{M + e^{\frac{\nu h}{p}}} \lim_{h \rightarrow 0^{+}} e^{\frac{\nu h}{p}} \lim_{h \rightarrow 0^{+}} \frac{e^{\frac{h}{p}} - 1}{\frac{h}{p}} = \frac{1}{M+1} \cdot 1 \cdot 1 = \frac{1}{M+1} \end{aligned}$$

Combined with the formalisation of the case $a_{\min} > 0$ sketched by Varnai et al. in [27, Sect. V], this completes the formal proof of shadow-lifting for STL.

■ **Table 3** Overview of the formalisation [29].

File	Contents	L.o.c.
<i>Additions to MATHCOMP libraries</i>		
<code>mathcomp_extra.v</code>	Lemmas iterated min/max, etc.	504
<code>analysis_extra.v</code>	L'Hôpital's rule, Cauchy's MVT (§ 5.3), etc.	820
<i>Generic logic and generic definitions of properties</i>		
<code>ldl.v</code>	LDL syntax and semantics (§ 3), shadow-lifting (§ 5.1)	417
<i>Soundness, logical and geometric properties of concrete logics</i>		
<code>dl2.v</code>	DL2: logical (§ 4), geometric (§ 5.2)	259
<code>fuzzy.v</code>	Gödel, Łukasiewicz, Yager, product: logical (§ 4), geometric (§ 5.2)	731
<code>stl.v</code>	STL: logical (§ 4), geometric (§ 5.4)	977
<i>Alternative formalisations of logical properties/ soundness using extended reals</i>		
<code>dl2_ereal.v</code>	DL2: logical (§ 4.2)	211
<code>stl_ereal.v</code>	STL: logical (§ 4.2)	362
	Total	4281

6 Conclusions, related and future work

We have presented a complete COQ formalisation of a range of existing DLs; making the following two main contributions:

1. We contribute to the DL community by *revisiting semantics of STL and DL2* in a way more amenable to formal verification. We find and fix errors in the literature.
2. We propose a *general formalisation strategy* based on dependent types and formal mathematics. The proposed formalisation is built to be easily extendable for future studies of different DLs.

Table 3 summarises the COQ implementation. Both L'Hôpital's rule and geometric properties, especially in complex cases such as STL, form a substantial part of the development. The proofs for fuzzy DLs (file `fuzzy.v`) are grouped together as thanks to their similarity, they share some of the proofs. The files `mathcomp_extra.v` and `analysis_extra.v` contain utility lemmas for the respective libraries. The file `mathcomp_extra.v` has a selection of lemmas on big operations (e.g., iterated sums, n -ary maximum), including lemmas for said operations when restricted to the domain $[0, 1]$ used by fuzzy logics. The file `analysis_extra.v` on the other hand contains proofs of L'Hôpital, Cauchy's MVT, and multiple lemmas on properties of `mine` and `maxe` (min and max for extended reals).

During our work, we completed missing parts of the shadow-lifting proof for STL, for example, the original STL proof failed to acknowledge the need for L'Hôpital's rule. We believe that understanding, let alone verifying theories that pertain to AI, without any mechanised support is difficult. Our previous attempt [24] to do this with pen and paper proofs was drowned in low-level case analysis and resulted in some errors, see Example 7. This complexity was our initial motivation to undertake the formalisation.

Regarding the concrete formalisation strategy, it was revealing that most of our formalisation was coherent with the standard MATHCOMP libraries (and standard mathematical results), and the library extensions we needed were natural (e.g., L'Hôpital's rule). This work hence demonstrates that COQ and MATHCOMP are effective working tools to formalise state-of-the-art AI results: DL2 and STL were published in recent conferences [16, 27] and this paper formalises the most significant results from both.

In the end, we have a uniform formalisation where all DLs are “tamed” which provides solid ground for formalisation of methods deployed in verification of neural networks.

Related Work

As this paper illustrates, neural network verification is a new field, and its nascent methods need validation and further refinement.

In terms of programming language support for neural network verification, a tool CAISAR [17], implemented as an OCaml DSL, puts emphasis on the smooth integration of a general specification language with many existing neural network solvers. However, CAISAR does not support property-guided training. The aspiration of languages like Vehicle and CAISAR is to accommodate compilation of specifications into both neural network solver and machine learning backends. For the former, there is an on-going work on certifying the neural network solver backends [12, 14].

On the side of machine-learning backends, DLs have been previously formalised in Agda as part of the Vehicle formalisation [5], but did not extend to shadow-lifting – the part that requires extensive mathematical libraries. Property-guided training certified via theorem proving was also proposed in [10].

Relevant work on formalisation of neural networks in ITPs includes: verification of neural networks in Isabelle/HOL [8] and Imandra [15], formalisation of piecewise affine activation functions in COQ [4], providing formal guarantees of the degree to which the trained neural network will generalise to new data in COQ [7], convergence of a single-layered perceptron in COQ [21]; and verification of neural archetypes in COQ [20]. The formalisation presented here does not directly formalise neural networks.

Future Work

We plan to consider other definitions of soundness, and other DLs, including STL with revised negation. We hypothesise that Definition 6 allows for generalisation (removing the condition “ $p_j = p$ ”) and this is left for future work. We saw in Sect. 4.2 that the choice of the interpretation domains has an impact on both soundness and shadow-lifting and this choice might deserve further investigation. The trade-off between idempotence, associativity and shadow-lifting that was conjectured in [27] is reminiscent of substructural logics and suggests investigating the connection. Establishing connection of this work with the logics of Lawvere quantale by Bacci et al. [6] might also provide new tools to study DLs.

Separately from the questions of scientific curiosity and mathematical elegance, there is a question of lacking programming language support for machine learning. As tools like Vehicle [13] and CAISAR [17] are being proposed to provide a more principled approach to verification of machine learning, in the long term, compilers of these new emerging languages will require certification. And this, in turn, will demand formalisation of results such as the ones we presented here. The formalisation of DLs would hence directly contribute to certified compilation of specification languages to machine learning libraries.

References

- 1 Reynald Affeldt, Yves Bertot, Alessandro Bruni, Cyril Cohen, Marie Kerjean, Assia Mahboubi, Damien Rouhling, Pierre Roux, Kazuhiko Sakaguchi, Zachary Stone, Pierre-Yves Strub, and Laurent Théry. MathComp-Analysis: Mathematical Components compliant analysis library. <https://github.com/math-comp/analysis>, 2024. Since 2017. Version 1.2.0.
- 2 Reynald Affeldt, Cyril Cohen, and Damien Rouhling. Formalization techniques for asymptotic reasoning in classical analysis. *J. Formaliz. Reason.*, 11(1):43–76, 2018. doi:10.6092/ISSN.1972-5787/8124.

- 3 Aws Albarghouthi. *Introduction to Neural Network Verification*. verifieddeeplearning.com, 2021. [arXiv:2109.10317](https://arxiv.org/abs/2109.10317).
- 4 Andrei Aleksandrov and Kim Völlinger. Formalizing piecewise affine activation functions of neural networks in Coq. In *15th International NASA Symposium on Formal Methods (NFM 2023)*, Houston, TX, USA, May 16–18, 2023, volume 13903 of *Lecture Notes in Computer Science*, pages 62–78. Springer, 2023. doi:10.1007/978-3-031-33170-1_4.
- 5 Robert Atkey, Matthew L. Daggitt, and Wen Kokke. Vehicle formalisation, 2024. URL: <https://github.com/vehicle-lang/vehicle-formalisation>.
- 6 Giorgio Bacci, Radu Mardare, Prakash Panangaden, and Gordon D. Plotkin. Propositional logics for the Lawvere quantale. In *39th Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXIX)*, Indiana University, Bloomington, IN, USA, June 21–23, 2023, volume 3 of *EPTICS*. EpiSciences, 2023. doi:10.46298/ENTICS.12292.
- 7 Alexander Bagnall and Gordon Stewart. Certifying the true error: Machine learning in Coq with verified generalization guarantees. In *The 33rd AAAI Conference on Artificial Intelligence (AAAI 2019)*, *The 31st Innovative Applications of Artificial Intelligence Conference (IAAI 2019)*, *The 9th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI 2019)*, Honolulu, Hawaii, USA, January 27–February 1, 2019, pages 2662–2669. AAAI Press, 2019. doi:10.1609/AAAI.V33I01.33012662.
- 8 Achim D. Brucker and Amy Stell. Verifying feedforward neural networks for classification in Isabelle/HOL. In *25th International Symposium on Formal Methods (FM 2023)*, Lübeck, Germany, March 6–10, 2023, volume 14000 of *Lecture Notes in Computer Science*, pages 427–444. Springer, 2023. doi:10.1007/978-3-031-27481-7_24.
- 9 Marco Casadio, Ekaterina Komendantskaya, Matthew L. Daggitt, Wen Kokke, Guy Katz, Guy Amir, and Idan Refaeli. Neural network robustness as a verification property: A principled case study. In *34th International Conference on Computer Aided Verification (CAV 2022)*, Haifa, Israel, August 7–10, 2022, Part I, volume 13371 of *Lecture Notes in Computer Science*, pages 219–231. Springer, 2022. doi:10.1007/978-3-031-13185-1_11.
- 10 Mark Chevallier, Matthew Whyte, and Jacques D. Fleuriot. Constrained training of neural networks via theorem proving (short paper). In *4th Workshop on Artificial Intelligence and Formal Verification, Logic, Automata, and Synthesis hosted by the 21st International Conference of the Italian Association for Artificial Intelligence (AIXIA 2022)*, Udine, Italy, November 28, 2022, volume 3311 of *CEUR Workshop Proceedings*, pages 7–12. CEUR-WS.org, 2022. URL: <https://ceur-ws.org/Vol-3311/paper2.pdf>.
- 11 Matthew Daggitt, Wen Kokke, Ekaterina Komendantskaya, Robert Atkey, Luca Arnaboldi, Natalia Slusarz, Marco Casadio, Ben Coke, and Jeonghyeon Lee. The Vehicle tutorial: Neural network verification with Vehicle. In *6th Workshop on Formal Methods for ML-Enabled Autonomous Systems*, volume 16 of *Kalpa Publications in Computing*, pages 1–5. EasyChair, 2023. doi:10.29007/5s2x.
- 12 Matthew L. Daggitt, Robert Atkey, Wen Kokke, Ekaterina Komendantskaya, and Luca Arnaboldi. Compiling higher-order specifications to SMT solvers: How to deal with rejection constructively. In *12th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2023)*, Boston, MA, USA, January 16–17, 2023, pages 102–120. ACM, 2023. doi:10.1145/3573105.3575674.
- 13 Matthew L. Daggitt, Wen Kokke, Robert Atkey, Natalia Slusarz, Luca Arnaboldi, and Ekaterina Komendantskaya. Vehicle: Bridging the embedding gap in the verification of neuro-symbolic programs, 2024. [arXiv:2401.06379](https://arxiv.org/abs/2401.06379).
- 14 Remi Desmartin, Omri Isac, Grant O. Passmore, Kathrin Stark, Ekaterina Komendantskaya, and Guy Katz. Towards a certified proof checker for deep neural network verification. In *33rd International Symposium Logic-Based Program Synthesis and Transformation (LOPSTR 2023)*, Cascais, Portugal, October 23–24, 2023, *Proceedings*, volume 14330 of *Lecture Notes in Computer Science*, pages 198–209. Springer, 2023. doi:10.1007/978-3-031-45784-5_13.

- 15 Remi Desmartin, Grant O. Passmore, Ekaterina Komendantskaya, and Matthew L. Daggitt. CheckINN: Wide range neural network verification in Imandra. In *24th International Symposium on Principles and Practice of Declarative Programming (PPDP 2022), Tbilisi, Georgia, September 20–22, 2022*, pages 3:1–3:14. ACM, 2022. doi:10.1145/3551357.3551372.
- 16 Marc Fischer, Mislav Balunovic, Dana Drachler-Cohen, Timon Gehr, Ce Zhang, and Martin T. Vechev. DL2: training and querying neural networks with logic. In *36th International Conference on Machine Learning (ICML 2019), 9–15 June 2019, Long Beach, California, USA*, volume 97 of *Proceedings of Machine Learning Research*, pages 1931–1941. PMLR, 2019. URL: <http://proceedings.mlr.press/v97/fischer19a.html>.
- 17 Julien Girard-Satabin, Michele Alberti, François Bobot, Zakaria Chihani, and Augustin Lemesle. CAISAR: A platform for characterizing artificial intelligence safety and robustness. In *The IJCAI-ECAI-22 Workshop on Artificial Intelligence Safety (AISafety 2022), July 24–25, 2022, Vienna, Austria*, CEUR-Workshop Proceedings, July 2022. URL: <https://hal.archives-ouvertes.fr/hal-03687211>.
- 18 Eleonora Giunchiglia, Mihaela Catalina Stoian, and Thomas Lukasiewicz. Deep learning with logical constraints. In *31st International Joint Conference on Artificial Intelligence (IJCAI-22)*, pages 5478–5485. International Joint Conferences on Artificial Intelligence Organization, July 2022. Survey Track. doi:10.24963/ijcai.2022/767.
- 19 Zico Kolter and Aleksander Madry. Adversarial robustness—theory and practice. NeurIPS 2018 tutorial, 2018. Available at <https://adversarial-ml-tutorial.org/>.
- 20 Elisabetta De Maria, Abdorrahim Bahrami, Thibaud L’Yvonnet, Amy P. Felty, Daniel Gaffé, Annie Ressouche, and Franck Grammont. On the use of formal methods to model and verify neuronal archetypes. *Frontiers Comput. Sci.*, 16(3):163404, 2022. doi:10.1007/S11704-020-0029-6.
- 21 Charlie Murphy, Patrick Gray, and Gordon Stewart. Verified perceptron convergence theorem. In *1st ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pages 43–50, 2017.
- 22 Kazuhiko Sakaguchi. Reflexive tactics for algebra, revisited. In *13th International Conference on Interactive Theorem Proving (ITP 2022), August 7–10, 2022, Haifa, Israel*, volume 237 of *LIPICs*, pages 29:1–29:22. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPICs.ITP.2022.29.
- 23 Kazuhiko Sakaguchi and Pierre Roux. Algebra tactics: Ring, field, lra, nra, and psatz tactics for Mathematical Components. <https://github.com/math-comp/algebra-tactics>, 2021. Last stable release: 1.2.3 (2024).
- 24 Natalia Ślusarz, Ekaterina Komendantskaya, Matthew L. Daggitt, Robert J. Stewart, and Kathrin Stark. Logic of differentiable logics: Towards a uniform semantics of DL. In *24th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR 2023), Manizales, Colombia, June 4–9, 2023*, volume 94 of *EPiC Series in Computing*, pages 473–493. EasyChair, 2023. doi:10.29007/C1NT.
- 25 Mathematical Components Team. Mathematical components library, 2007. Last stable version: 2.2.0 (2024). URL: <https://github.com/math-comp/math-comp>.
- 26 Emile van Krieken, Erman Acar, and Frank van Harmelen. Analyzing differentiable fuzzy logic operators. *Artif. Intell.*, 302:103602, 2022. doi:10.1016/J.ARTINT.2021.103602.
- 27 Péter Várnai and Dimos V. Dimarogonas. On robustness metrics for learning STL tasks. In *2020 American Control Conference (ACC 2020), Denver, CO, USA, July 1–3, 2020*, pages 5394–5399. IEEE, 2020. doi:10.23919/ACC45564.2020.9147692.
- 28 J Łukasiewicz. *O logice trójwartościowej (in Polish). English translation: On Three-Valued Logic, in Borkowski, L.(ed.) 1970. Jan Łukasiewicz: Selected Works, Amsterdam: North Holland. Ruch Filozoficzny*, 1920.
- 29 Natalia Ślusarz, Reynald Affeldt, and Alessandro Bruni. Formalisation of Differentiable Logics in Coq, 2024. Software, swhId: `swh:1:dir:bd213b761dfc453ccfe8e785a38cffe583c98f04` (visited on 2024-08-21). URL: https://github.com/ndslusarz/formal_LDL.

A Comprehensive Overview of the Lebesgue Differentiation Theorem in Coq

Reynald Affeldt  

National Institute of Advanced Industrial Science and Technology (AIST), Tokyo, Japan

Zachary Stone

The MathComp-Analysis development team, Boston, MA, USA

Abstract

Formalization of real analysis offers a chance to rebuild traditional proofs of important theorems as unambiguous theories that can be interactively explored. This paper provides a comprehensive overview of the Lebesgue Differentiation Theorem formalized in the Coq proof assistant, from which the first Fundamental Theorem of Calculus (FTC) for the Lebesgue integral is obtained as a corollary. Proving the first FTC in this way has the advantage of decomposing into loosely-coupled theories of moderate size and of independent interest that lend themselves well to incremental and collaborative development. We explain how we formalize all the topological constructs and all the standard lemmas needed to eventually relate the definitions of derivability and of Lebesgue integration of MathComp-Analysis, a formalization of analysis developed on top of the Mathematical Components library. In the course of this experiment, we substantially enrich MathComp-Analysis and even devise a new proof for Urysohn’s lemma.

2012 ACM Subject Classification Theory of computation → Logic and verification; Theory of computation → Type theory; Mathematics of computing → Mathematical analysis

Keywords and phrases Coq proof assistant, Mathematical Components library, Lebesgue integral, fundamental theorem of calculus

Digital Object Identifier 10.4230/LIPIcs.ITP.2024.5

Supplementary Material *Software (Source Code)*: <https://github.com/math-comp/analysis> [1] archived at `swh:1:dir:603335be2da03f1a37b210f027047f94aa0ed2c6`

Funding *Reynald Affeldt*: The first author acknowledges support of the JSPS KAKENHI Grant Number 22H00520.

Acknowledgements The authors are grateful to A. Bruni, C. Cohen, Y. Ishiguro, and T. Saikawa for their inputs. This work has benefited from feedback gained during the MATHCOMP-ANALYSIS development meetings. The authors would like to thank the anonymous referees of ITP 2024 for their careful and informative review, as well as the program committee of JFLA 2024 where a preliminary version of this work was presented [5].

1 Introduction

The formalization of the Fundamental Theorem of Calculus (FTC) for the Lebesgue integral in the COQ proof assistant [29] is an ongoing work as part of the development of MATHCOMP-ANALYSIS [1], a library for analysis that extends the Mathematical Components library [17] with classical axioms [3, §5]. Besides mathematics, MATHCOMP-ANALYSIS has also been used to formalize programming languages [4, 27, 31].

The first FTC for Lebesgue integration can be stated as follows: for f integrable on \mathbb{R} , $F(x) \stackrel{\text{def}}{=} \int_{t \in]-\infty, x]} f(t)(\mathbf{d}\mu)$ is differentiable and $F'(x) = f(x)$ almost-everywhere (hereafter, a.e.) relatively to μ , where μ is the Lebesgue measure. This is different from the standard statement for the Riemann integral, where f is assumed to be continuous, making for a simple proof. In comparison, connecting derivation and Lebesgue integration under an integrability



© Reynald Affeldt and Zachary Stone;

licensed under Creative Commons License CC-BY 4.0

15th International Conference on Interactive Theorem Proving (ITP 2024).

Editors: Yves Bertot, Temur Kutsia, and Michael Norrish; Article No. 5; pp. 5:1–5:19

Leibniz International Proceedings in Informatics



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

hypothesis is unwieldy, even more so in `MATHCOMP-ANALYSIS` whose formalizations of derivation [3, §4.5] and of the Lebesgue integral [2, §6.4] have been unrelated so far. They can be bridged thanks to the Lebesgue Differentiation theorem and this is appealing for two reasons. First, it is a useful theorem in itself: the first FTC is a consequence, as well as other results such as Lebesgue’s density theorem. Second, we can decompose its proof in several results: this provides a way to incrementally enrich the theories of `MATHCOMP-ANALYSIS`. We think that this approach is an instance of a more generic way to tackle formalization of mathematics: find a path through the literature to present many key results as easy consequences of a central, technical lemma with rather weak assumptions. Incidentally, such a fine-grained decomposition also provides a practical way to monitor formalization progress.

In terms of formalization in a proof assistant, our contributions are as follows:

- We provide the first formalization of the first FTC for Lebesgue integration in `COQ`.
- We bring to `COQ` several standard lemmas and theorems of measure theory: Vitali’s lemmas and theorem, a theory of Hardy-Littlewood’s operator, Urysohn’s lemma, Ergorov’s, Lusin’s, Tietze’s theorems, and the Lebesgue Differentiation theorem. In particular, among these results, Urysohn’s lemma is given an original proof. We also improve the `MATHCOMP-ANALYSIS` support for topology (lower semicontinuity, normal spaces, subspaces, etc.) and for real functions (by extending the theory for \limsup / \liminf). The formalization of the first FTC for Lebesgue integration provides a strong evidence that these pieces of formalization can indeed be combined to achieve a large result.

Another intent with this paper is to produce an informative document for potential users of the topology and measure theories of `MATHCOMP-ANALYSIS`. The whole library is still under development in the sense that not all notions are formalized as we would like them to be, often to cope with temporary limitations of the available tooling. Yet, we did observe that it is already a useful tool. For example, we could use it to revisit the proof of Urysohn’s lemma by producing an original proof. Also, we had to clarify a few proof steps that are often hand-waved in lecture notes: typically, generalizations from lemmas stated for bounded cases only. Filling such gaps is almost business-as-usual when formalizing mathematics, but we believe that it is important to document them to better anticipate similar gaps in the future. For these reasons, we think that our formalization of the FTC provides a nice milestone to document `MATHCOMP-ANALYSIS`.

Outline

Regarding mathematical proofs, we stay at the level of a bird’s-eye view and instead focus on the main aspects of the formalization. We start by recalling the basics of `MATHCOMP-ANALYSIS` in Sect. 2. We explain the formal statement of the Lebesgue Differentiation theorem in Sect. 3 and provide an overview of its proof in Sect. 4. To formalize this proof, we extend `MATHCOMP-ANALYSIS` with new topological constructs in Sect. 5 and with basic but new measure-theoretic lemmas in Sect. 6. In the particular case of Urysohn’s lemma, we explain the formalization of an original proof in Sect. 7. The main intermediate lemmas of the Lebesgue Differentiation theorem are the purpose of Sect. 8 and Sect. 9. Finally, we apply the Lebesgue Differentiation theorem to the proof of the first FTC for Lebesgue integration and to the proof of Lebesgue’s density theorem in Sect. 10. We review related work in Sect. 11 and conclude in Sect. 12.

2 Background on MathComp-Analysis

MATHCOMP-ANALYSIS is built on top of MATHCOMP and reuses several of its theories. We use in particular the following notations, which are traditionally in ASCII in MATHCOMP libraries. The successor function of natural numbers is noted `.+1`, multiplication of a natural number by 2 is noted `.*2`. A multiple conjunction is noted `[/\ P1, P2, ... & Pn]`. Function composition is noted `\o`. Point-wise equality between two functions is noted `=1`. Point-wise multiplication of two functions `f` and `g` is noted `f * g`. The notation `f ^~ y` is for the function $\lambda x.f x y$. Intervals are noted ``]a, b[`, `]a, b]`, etc. One can inject a natural number `n` into a ring with the notation `n%:R`. The inverse of a field element `r` is noted `r^-1`. The norm of `x` is noted ``|x|`. The inclusion between two lists `s` and `r` is noted `{subset s <= r}`. We write `x \in s` when an element `x` belongs to the list `s`.

MATHCOMP-ANALYSIS comes with library support for set theory. Given a type `T`, `set T` is the type of sets of elements of type `T`. The notation `[set: T]` is for the set of all the elements of type `T`; the singleton set containing `x` is noted `[set x]`; and `set0` represents the empty set. To unambiguously improve readability, we use standard L^AT_EX notations instead of ASCII for the set-theoretic operations set inclusion (\subseteq), set intersection (\cap), set union (\cup), set difference (\setminus), and the product of sets (\times). The complement of a set `A` is noted `Ac`. (If necessary, see [2, Table 2] for a list-up of the corresponding ASCII notations.) Set difference with a singleton is noted `A ` \ x`: it is a shortcut for `A \ [set x]`. The image by a function `f` of a set `A` is written `f @` A`. The set $\{f(x) \mid x \in A\}$ defined by comprehension is noted `[set f x | x in A]`. A list `s` can be turned into a set with the notation `[set ` s]`. The notation `A !=set0` means that the set `A` is not empty. A family `F` of pairwise-disjoint sets indexed by `D` is noted `trivIset D F`. The characteristic function over a set `A` is noted `\1_A`.

MATHCOMP-ANALYSIS extends the numeric types of MATHCOMP with the type `realType` for reals. When `R` is a numeric type, `\bar R` is the numeric type extended with $-\infty$ and $+\infty$, so that when `R : realType`, `\bar R` corresponds to $\overline{\mathbb{R}}$. One can inject a numeric value `r` into the corresponding type of extended numbers with the notation `r%:E` (this is actually a notation for `EFin r`). An extended number `x` can be projected to the corresponding numeric value by `fine x` (which is `0` when `x` is $\pm\infty$). The supremum of a set of extended reals `A` is noted `ereal_sup A`. In this paper, the variable `R` has type `realType` unless stated otherwise.

Like several other libraries [7, 13, 30], MATHCOMP-ANALYSIS uses filters to formalize topology. For example, we note `\oo` the filter consisting of the set of predicates over natural numbers that are eventually true; `x^-'` the *deleted neighborhood filter* of `x`, i.e., the set of neighborhoods of `x` from which `x` is excluded; `x^-'+` for *right filters*, i.e., the filters of neighborhoods of `x` intersected with $]x, +\infty[$, and similarly for the *left filters* note `x^-'-`. Filters are associated to elements of a given type upon the definition of a *filtered type*. The convergence statement $f(x) \xrightarrow{x \rightarrow a} \ell$ is noted `f x @ [x --> a] --> l`; the limit of a filter `F` is noted `lim F` [3, §2.3]. Topological spaces are built on top of filtered types and their type is `topologicalType`. In a topological space, the set of neighborhoods of `x` is `nbhs x`. Mathematical structures such as topological spaces are defined and instantiated using a COQ extension called HIERARCHY-BUILDER [9]. With this tool, interfaces are defined as so-called *factories* whose definition generates constructors to build instances. See [2, §3.1] for a quick reference to HIERARCHY-BUILDER. The predicates `open`, `closed`, `closure`, and `compact` correspond to the eponymous topological notions. The type of uniform spaces is `uniformType`. Given a uniform space `M`, entourages are objects of type `set (set (M * M))` that satisfy the axioms of uniform space for `M`. One of the axiom of uniform spaces guarantees that for each entourage `E`, there is an entourage `V` with $\{(x, z) \mid \exists y, (x, y) \in V \wedge (y, z) \in V\} \subseteq E$. We denote

this entourage with `split_ent(E)`. MATHCOMP-ANALYSIS also provides pseudometric spaces in which a ball is noted `ball`; over the real line, a ball is a centered open interval. The type of sequences (indexed by natural numbers) over `T` is noted `T^nat`.

The basics of measure theory in MATHCOMP-ANALYSIS is documented in previous work [2]. A `measurableType` is a type equipped with a structure of σ -algebra. Given a measurable type `T` and `A` of type `set T`, we write `measurable A` when the set `A` is measurable. The fact that the extended real-valued function `f` is measurable over `D` is noted `measurable_fun D f`. For a measure `mu`, the fact that `f` is integrable over `D` is noted `mu.-integrable D f`. The integral $\int_{x \in A} f(x)(d\mu)$ is noted `\int[mu]_(x in A) f x`. When `A` is negligible for a measure `mu`, we write `mu.-negligible A`. The fact that the predicate `P` holds a.e. relatively to `mu` is noted `{ae mu, forall x, P x}`.

3 Statement of the Lebesgue Differentiation theorem

Let us note $[f]_A \stackrel{\text{def}}{=} \frac{1}{\mu(A)} \int_{y \in A} |f(y)|(d\mu)$ the average of a real-valued function `f` over the set `A`. Using the existing notations of MATHCOMP-ANALYSIS, we formalize this notion as follows:

Definition `iavg (f : R -> R) (A : set R) :=`
`(fine (mu A))^-1%:E * \int[mu]_(y in A) `| (f y)%:E |.`

Recall from Sect. 2 that `%:E` injects a numeric value into its extended version and that `fine` performs a corresponding projection. We also introduce the notation $\overline{f_{B_r(x)}} \stackrel{\text{def}}{=} [\lambda y. f(y) - f(x)]_{B_r(x)}$ where `Br(x)` is a ball centered at `x` of radius `r`:

Definition `davg (f : R -> R) (x r : R) := iavg (center (f x) \o f) (ball x r).`

The `center c` function is $\lambda y. y - c$.

Given a real-valued function `f`, a *Lebesgue point* is a real number `x` s.t. $\overline{f_{B_r(x)}} \xrightarrow{r \rightarrow 0^+} 0$. Reusing the Lebesgue measure (hereafter `mu`) formalized in previous work [2, §5.2], we formally define Lebesgue points:

Definition `lebesgue_pt (f : R -> R) (x : R) :=`
`davg f x r @[r --> 0^'+] --> 0.`

Note the use of right filters (Sect. 2) to define the fact that `r` tends to 0^+ . The Lebesgue Differentiation theorem states that, for a real-valued, locally-integrable function `f` (i.e., integrable on compact subsets of its domain), we have Lebesgue points a.e.:

Lemma `lebesgue_differentiation (f : R -> R) : locally_integrable [set: R] f`
`-> {ae mu, forall x, lebesgue_pt f x}.`

Being locally integrable can be defined as the following conjunction:

Definition `locally_integrable (D : set R) (f : R -> R) :=`
`[/\ measurable_fun D f, open D & forall K, K ⊆ D -> compact K ->`
`\int[lebesgue_measure]_(x in K) `|f x|%:E < +oo].`

In fact, this definition specializes in a locally compact space such as \mathbb{R} to the conjunction of `open D` and

`forall x, D x -> exists U, open_nbhs x U /\ mu.-integrable U (EFin \o f)`

where `open_nbhs` is a predicate for open neighborhoods; yet, we stuck to the previous definition from our reference textbook [16].

4 Proof of the Lebesgue Differentiation theorem

The first step of the proof of the Lebesgue Differentiation theorem is to reduce the problem to functions $f_k \stackrel{\text{def}}{=} f \mathbb{1}_{B_k}$ with $B_k \stackrel{\text{def}}{=} B_{2^{k+1}}(0)$:

```

Lemma lebesgue_differentiation_bounded (f : R -> R) :
  let B k := ball 0 k.+1.*2%:R in let f_k := f \* \1_(B k) in
  (forall k, mu.-integrable [set: R] (EFin \o f_k)) ->
  forall k, {ae mu, forall x, B k x -> lebesgue_pt (f_k) x}.
    
```

This problem reduction is often hand-waved in lecture notes (Schwartz’s presentation is one exception [28, eqn (5.12.101)]).

Second, instead of proving for all k that we have a.e. Lebesgue points over B_k , it is sufficient to prove that the set $A_k(a) \stackrel{\text{def}}{=} B_k \cap \{x \mid a < \limsup_{r \rightarrow 0} \overline{f_{k B_r(x)}}\}$ is negligible for all $a > 0$, i.e.:

```

(* local context omitted *)
=====
mu.-negligible (B k \cap [set x \mid a%:E < (f_k)^* x])
    
```

where $h^* x$ is a (local) notation for $\limsup_{r \rightarrow 0} \overline{h_{B_r(x)}}$.

For this last step, the idea is to exhibit a sequence of continuous functions g_i such that

$$A_k(a) \subseteq \bigcap_n B_k \cap \underbrace{\{x \mid f_k(x) - g_n(x) \geq a/2\}}_{(a)} \cup \underbrace{\{x \mid \text{HL}(f_k(x) - g_n(x)) > a/2\}}_{(b)}$$

where $\text{HL}(f)(x) \stackrel{\text{def}}{=} \sup_{r>0} \{[f]_{B_r(x)}\}$ is the Hardy-Littlewood operator. We can show that the measure of the right-hand side is null. To deal with (a), we use Markov’s inequality (i.e., the fact that $\mu(\{x \mid |f(x)| \geq a\}) \leq \frac{1}{a} \|f\|_1$ for $a > 0$ and a measure μ , where $\|\cdot\|_1$ is the L^1 norm) and the fact that continuous functions are dense in L^1 , whose proof requires several standard results of measure theory (in particular, Urysohn’s lemma). To deal with (b), we need the Hardy-Littlewood maximal inequality, which in turns relies on Vitali’s covering lemma.

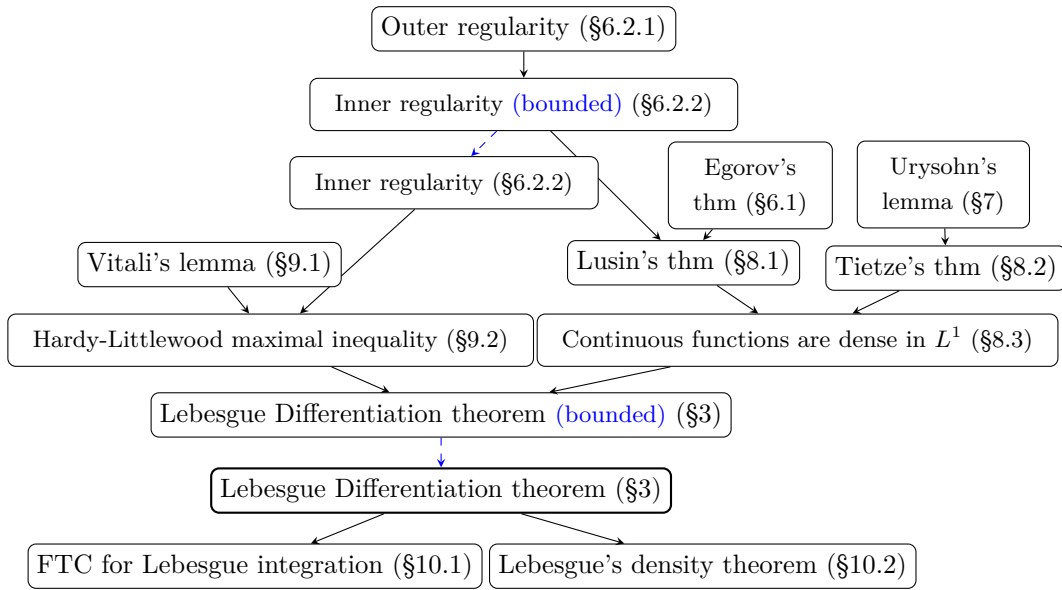
Figure 1 shows the main lemmas that we add to MATHCOMP-ANALYSIS to formalize this proof.

5 Topological constructs added to MathComp-Analysis

Before explaining the formalization of the main lemmas to prove the Lebesgue Differentiation theorem, we explain the preparatory work needed to extend the formalization of topology of MATHCOMP-ANALYSIS. In particular, as a preliminary step to be able to state Egorov’s theorem, Lusin’s theorem, and Tietze’s theorem, we extend MATHCOMP-ANALYSIS with the subspace topology and with the topology of uniform convergence.

5.1 Subspace topologies

Given a type T equipped with a topology \mathcal{T} and a set A of elements of T , $\{A \cap U \mid U \in \mathcal{T}\}$ forms a subspace topology. We formalize subspace topologies on the basis of the existing formalization of topological space of MATHCOMP-ANALYSIS and using HIERARCHY-BUILDER (see Sect. 2). First, we introduce an identifier `subspace` that serves as an alias for a type T and which is parameterized by a set A :



■ **Figure 1** Overview of the proof of the Lebesgue Differentiation theorem and derived results. (dashed arrows indicate generalizations from statements about bounded sets)

Definition `subspace {T : Type} (A : set T) := T`.

When the underlying type T is equipped with a topology (i.e., it has type `topologicalType`), we can equip the identifier `subspace` with a structure of topology relative to A . For that purpose, we start by defining a filter for a point x in the subspace topology as: (1) the restriction of the neighborhoods of x to A (below: `within A (nbhs x)`) when x belongs to A , or, otherwise, (2) the filter of the sets containing the singleton $\{x\}$ (below: `globally [set x]`):

Definition `nbhs_subspace (x : subspace A) : set_system (subspace A) := if x \in A then within A (nbhs x) else globally [set x]`.

A `set_system` is simply synonymous for a set of sets. Filters defined in this way give rise to a filtered type (Sect. 2). To attach the structure of filtered type with the identifier `subspace`, it suffices to summon `HIERARCHY-BUILDER` with the appropriate *factory*:

`HB.factory Record hasNbhs T := { nbhs : T -> set_system T }`.

A factory is represented by an interface (here, `hasNbhs`) and its definition gives rise to a constructor (here, `hasNbhs.Build`) that can be used to build instances, e.g.:

`HB.instance Definition _ := hasNbhs.Build (subspace A) nbhs_subspace`.

The filtered type associated with `subspace` can furthermore be equipped with a topology using this other factory provided by `MATHCOMP-ANALYSIS`:

`HB.factory Record Nbhs_isNbhsTopological T of Nbhs T := { nbhs_filter : forall p : T, ProperFilter (nbhs p); nbhs_singleton : forall (p : T) (A : set T), nbhs p A -> A p; nbhs_nbhs : forall (p : T) (A : set T), nbhs p A -> nbhs p (nbhs ~ A) }`.

Indeed, (1) one can show that `nbhs_subspace`'s are *proper filters* (i.e., no set in the filter is empty) [3, §3.2.2], (2) points are contained into their neighborhoods, and (3) given a

neighborhood A , the set of points of which A is a neighborhood is also a neighborhood. It suffices to build an instance of topological type by passing to the factory above the proofs of the facts (1), (2), and (3) (omitted here but that can be found in the formal development [1, file topology.v]):

```
HB.instance Definition _ := Nbhs_isNbhsTopological.Build (subspace A)
  nbhs_subspace_filter nbhs_subspace_singleton nbhs_subspace_nbhs.
```

This is the subspace topology relative to A . Note that this topology is discrete outside of A .

In particular, we use the topology of subspace to define continuity as follows. The notation `{within A, continuous f}` denotes continuity of $f : \text{subspace } A \rightarrow Y$:

```
Notation "{ 'within' A , 'continuous' f }" :=
  (continuous (f : subspace A -> _)).
```

The notation `continuous` comes from MATHCOMP-ANALYSIS [1, file topology.v] and pre-dates this work. The purpose of the notation `{within A, continuous f}` is to ensure that the continuity of f depends only on its values in A while $f(x + \text{eps})$ type-checks. If x and eps have type `subspace A`, then $f(x + \text{eps})$ is indeed well-defined. One might naively attempt to use the sigma type `{x | A x}` with the weak topology (i.e., the topology defined by the preimages of opens for a given function) induced by the function `projT1 : {x : T | A x} -> T` to define the subspace topology. However, there is no clear way to define addition for `{x | A x}` so that $f(x + \text{eps})$ is well-typed.

5.2 Uniform convergence

The methodology to formalize the topology of uniform convergence is similar to the one used to formalize subspaces.

First, let us recall how total functions are equipped with the structure of uniform space in MATHCOMP-ANALYSIS. There is an identifier `arrow_uniform_type` which is an alias for the type $U \rightarrow V$ with $U : \text{choiceType}$ and $V : \text{uniformType}$. The corresponding uniform space is generated by the entourages $\{(f, g) \mid \forall x : U, E(f(x), g(x))\}$ where E is an entourage of V (see `fct_ent` in [1, file function_spaces.v]). As a consequence of the appropriate HIERARCHY-BUILDER instantiation, `arrow_uniform_type` is given the type `uniformType`.

Now, we formalize a topology whose elements are functions from the set A to the type V . First, we introduce an identifier:

```
Definition uniform_fun {U : Type} (A : set U) (V : Type) : Type := U -> V.
```

We also introduce a notation `{uniform` A -> V}` which stands for `@uniform_fun _ A V`. We then introduce the (high-order) function `sigL_arrow` that turns a function $U \rightarrow V$ between two types into a function $A \rightarrow V$ from a set to a type:

```
Definition sigL_arrow {U : choiceType} (A : set U) (V : uniformType) :
  (U -> V) -> arrow_uniform_type A V := @sigL _ V A.
```

The function `sigL` comes from [1, file functions.v]. The identifier `uniform_fun` is then equipped with the weak topology (`weak_topology` in MATHCOMP-ANALYSIS) induced by `sigL_arrow` and eventually the desired topology is simply obtained by *copying* the structure obtained by weak topology:

```
HB.instance Definition _ (U : choiceType) (A : set U) (V : uniformType) :=
  Uniform.copy {uniform` A -> V} (weak_topology (@sigL_arrow _ A V)).
```

Copying a structure is a feature provided by HIERARCHY-BUILDER. Note that we can copy the structure using `Uniform.copy` instead of `Topological.copy` because weak topology always inherits a uniform structure, see the section `weak_uniform` in [1, file `topology.v`].

Then, the uniform convergence of a sequence of functions F towards f over a set A can be defined. It is the filter inclusion of the neighborhoods of f in $\{\text{uniform } A \rightarrow V\}$ into the filter F , and this inclusion is written `cvg_to F (nbhs (f : {uniform } A -> _))` in MATHCOMP-ANALYSIS. The notation $\{\text{uniform } A, F \dashrightarrow f\}$ is for the latter inclusion.

5.3 More support for extended real-valued functions

Besides the topological structures we explained in the previous sections, Sect. 4 also highlights the need to generalize MATHCOMP-ANALYSIS's theory of \limsup / \liminf . Before our experiment, this theory was limited to sequences (indexed by natural numbers) that were actually introduced to develop the monotone convergence theorem and its consequences [2, §6.5]. Handling extended real-valued functions over the real numbers requires the formalization of the following definition: $\limsup_{x \rightarrow a} f(x) \stackrel{\text{def}}{=} \lim_{\varepsilon \rightarrow 0^+} \sup\{f(x) \mid x \in B_\varepsilon(a) \setminus \{a\}\}$. It can be couched in formal terms by first defining the limit superior of a function f at filter F :

```
Variables (T : choiceType) (X : filteredType T) (R : realFieldType).
Implicit Types (f : X -> \bar R) (F : set (set X)).
Definition limf_esup f F := ereal_inf [set ereal_sup (f @` V) | V in F].
```

We can then specialize this definition to define the limit superior of a function over the type of real numbers by using deleted neighborhood filters:

```
Variable R : realType.
Implicit Types (f : R -> \bar R) (a : R).
Definition lime_sup f a : \bar R := limf_esup f a^'.
```

This generic definition of `lime_sup` can be shown to be equivalent to $\lim_{\varepsilon \rightarrow 0^+} \sup\{f(x) \mid x \in B_\varepsilon(a) \setminus \{a\}\}$:

```
Let sup_ball f a r := ereal_sup [set f x | x in ball a r ` \ a].
Lemma lime_sup_lim f a : lime_sup f a = lim (sup_ball f a e @[e --> 0^'+]).
```

In the course of formalizing the Lebesgue Differentiation theorem, developing the theory of \limsup / \liminf turned out to be a non-trivial intermission, which revealed some quirks (now fixed) in the automatic handling of right filters using the `near` tactics [3, §3.2] in MATHCOMP-ANALYSIS.

6 Egorov's theorem and regularity

The top part of Fig. 1 reveals the first measure-theoretic results needed to prove the Lebesgue Differentiation theorem. Lusin's theorem requires Egorov's theorem as well as the inner regularity of the Lebesgue measure, which is also used to prove the Hardy-Littlewood maximal inequality. In the following, μ is the Lebesgue measure. We give little detail about the proofs in this section because proof scripts are essentially textbook, they can be found in MATHCOMP-ANALYSIS [1, file `lebesgue_measure.v`].

6.1 Egorov's theorem

Egorov's theorem relates convergence a.e. and uniform convergence (Sect. 5.2). Let A be a bounded measurable set, f_k be a sequence of functions measurable over A , and g be a function measurable over A . Suppose that f_k converges a.e. relatively to the Lebesgue measure μ towards g . Then for any $\varepsilon > 0$, there exists a measurable set B such that $\mu(B) < \varepsilon$ and f_k converges uniformly towards g over $A \setminus B$. Formally, assuming T is a measurable type:

```
Lemma ae_pointwise_almost_uniform (f_ : (T -> R)^nat) (g : T -> R) A eps :
  (forall k, measurable_fun A (f_ k)) -> measurable_fun A g ->
  measurable A -> mu A < +oo ->
  {ae mu, forall x, A x -> f_ ^~ x @ \oo --> g x} ->
  (0 < eps)%R -> exists B, [/ \ measurable B, mu B < eps%:E &
  {uniform A \ B, f_ @ \oo --> g}].
```

The notation for uniform convergence has been explained in Sect. 5.2.

6.2 Regularity

Proving the outer regularity of the Lebesgue measure is a preliminary step before proving its inner regularity.

6.2.1 Outer regularity

The Lebesgue measure is *outer regular*, which means that it can be approximated from above by open subsets. More formally, for every bounded measurable set D and $\varepsilon > 0$, there exists an open $U \supseteq D$ such that $\mu(U \setminus D) < \varepsilon$:

```
Lemma lebesgue_regularity_outer D eps :
  measurable D -> mu D < +oo -> (0 < eps)%R ->
  exists U : set R, [/ \ open U , D \subseteq U & mu (U \ D) < eps%:E].
```

The proof is based on the definition of the Lebesgue measure as the infimum of the measures of covers, i.e., $\mu(X) = \inf_F \{ \sum_{k=0}^{\infty} \mu(F_k) \mid (\forall k, \text{measurable}(F_k)) \wedge X \subseteq \bigcup_k F_k \}$.

6.2.2 Inner regularity

Intuitively, a measure is *inner regular* when it can be approximated from within by a compact subset. The inner regularity of the Lebesgue measure states that for every (bounded) measurable set D and $\varepsilon > 0$, there exists a compact set $V \subseteq D$ such that $\mu(D \setminus V) < \varepsilon$:

```
Lemma lebesgue_regularity_inner D eps :
  measurable D -> mu D < +oo -> (0 < eps)%R ->
  exists V : set R, [/ \ compact V , V \subseteq D & mu (D \ V) < eps%:E].
```

Textbooks also resort to an alternative statement of inner regularity. Precisely, the above statement about a bounded measurable set can be generalized using the σ -finiteness of the Lebesgue measure by saying that the measure of a measurable set can be expressed as the supremum of the measure of the compact sets included inside, i.e., $\mu(D) = \sup\{\mu(K) \mid \text{compact}(K) \wedge K \subseteq D\}$:

```
Lemma lebesgue_regularity_inner_sup D : measurable D ->
  mu D = ereal_sup [set mu K | K in [set K | compact K /\ K \subseteq D]].
```

As a matter of fact, we do use both forms in our development (Fig. 1).

7 A new proof of Urysohn's lemma

The classical version of Urysohn's lemma states that a topological space T is *normal* (i.e., two disjoint closed sets have disjoint open neighborhoods) if and only if, for any closed, disjoint, non-empty subsets A and B , there is a continuous function $f : T \rightarrow \mathbb{R}$ such that $f(A) = \{0\}$ and $f(B) = \{1\}$. This result is important because it connects a purely topological property (normality) with a purely analytic property (a function into the reals). Traditionally the proof involves an induction over the rationals to explicitly construct such a function. We do not follow the traditional proof for a technique that, we believe, is more appealing from the viewpoint of formal verification. Our proof has the same pattern as proving the Lebesgue Differentiation theorem first, and then the FTC: we found that proving the right intermediate lemmas made several results, including Urysohn's much easier.

7.1 Urysohn's lemma using uniform separator

We start by stating one of our intermediate lemmas. We first introduce a new definition. For a topological space T , we define a *uniform separator* as follows:

```
Definition uniform_separator (A B : set T) :=
  exists (uT : @Uniform.axioms_ T) (E : set (T * T)),
    let UT := Uniform.Pack uT in [/\
      @entourage UT E,
      (A × B) ∩ E = set0 &
      (forall x, @nbhs UT UT x ⊆ @nbhs T T x)].
```

This says that there is a uniform structure UT on T which separates A and B , and is coarser than the T topology. This is subtly different than assuming that T is a uniform space, which would imply $\text{forall } x, \text{ @nbhs } UT \text{ } UT \text{ } x = \text{ @nbhs } T \text{ } T \text{ } x$, which is too strong for our purposes. Also, $(A \times B) \cap E = \text{set0}$ has a nice visual: since an entourage is a region around the diagonal of $T * T$, $(A \times B) \cap E = \text{set0}$ means that the region $A \times B$ is far from the diagonal.

The key result about uniform separators is:

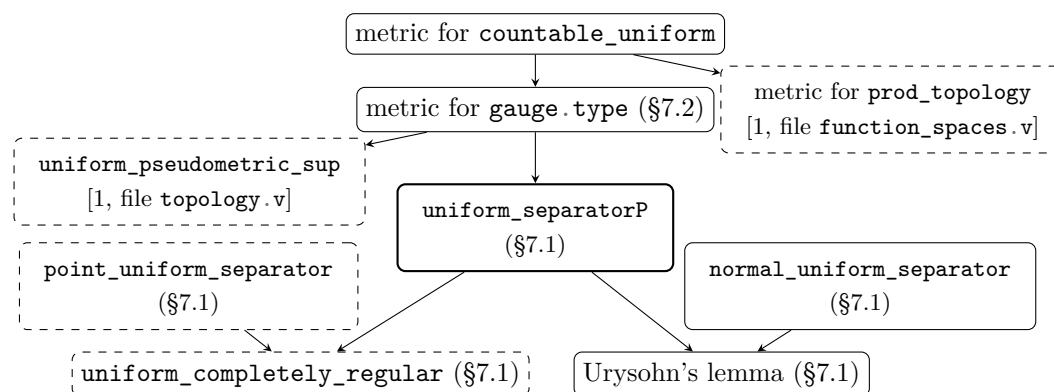
```
Lemma uniform_separatorP {T : topologicalType} {R : realType} (A B : set T) :
  uniform_separator A B <-> exists f : T -> R, [/\
    continuous f,
    range f ⊆ ` [0, 1],
    f @` A ⊆ [set 0] &
    f @` B ⊆ [set 1]].
```

For the sake of readability, we defer the explanation of the proof to the next section (Sect. 7.2).

The lemma just above is nearly Urysohn's lemma, but does not assume that T is a normal space. In fact, Urysohn's lemma follows immediately from the following lemma [1, file `normedtype.v`]:

```
Lemma normal_uniform_separator {T : topologicalType} (A : set T) (B : set T) :
  normal_space T -> closed A -> closed B -> A ∩ B = set0 ->
  uniform_separator A B.
```

The advantage of the general lemma `uniform_separatorP` is that, besides Urysohn's lemma, we can derive other results. A *completely regular space* T is a topological space where for every point x and closed set A with $x \notin A$ there is a continuous function $f : T \rightarrow \mathbb{R}$



■ **Figure 2** Overview of a novel proof of Urysohn's lemma and derived results. (dashed nodes are derived results that are not relevant to the Lebesgue Differentiation theorem)

with $f(x) = 0$ and $f(A) = \{1\}$. The classical result is that uniform spaces are completely regular (lemma `uniform_completely_regular` in `[1, file normedtype.v]`). This follows from the lemma `uniform_separatorP` and the following lemma:

Lemma `point_uniform_separator` $\{\text{uniformType } T\}$ $(x : T)$ $(B : \text{set } T) :$
`closed B -> ~ B x -> uniform_separator [set x] B.`

So, just like this paper formalizes the Lebesgue Differentiation theorem and proves the FTC among others, we find out that the lemma `uniform_separatorP` proves Urysohn's lemma and more.

7.2 Existence of uniform separators

The proof of `uniform_separatorP` from the previous section (Sect. 7.1) follows other intermediate lemmas. If T is a uniform space with a countable basis for its uniformity (i.e., the set of entourages of T is the upward closure of a countable subset of entourages, see `countable_uniformity` `[1, file topology.v]`), then T has a pseudometric [15]. The construction of the metric is rather involved, but is only done once. Then, whenever we want a function into the reals, we construct a suitable uniformity instead, and rely on this result to guarantee such a function. This has several useful consequences, such as proving countable products of metric spaces are metrizable (Fig. 2). More importantly, the metrizability of uniform spaces lets us build the so-called *gauge metric*. Given an entourage E , we get a metric for the uniform structure generated by $E_0 = E \cap E^{-1}$, $E_1 = \text{split_ent}(E_0) \cap \text{split_ent}(E_0)^{-1}$, etc., where `split_ent` was explained in Sect. 2.

The direct part of the proof of `uniform_separatorP` starts by building the gauge metric of the separator of A and B generated from E . Since the initial entourage separates A and B , we know that there is an $\varepsilon > 0$ such that for all $x \in A$ and $y \in B$, $y \notin B_\varepsilon(x)$. We define the extended real-valued function $d(x, y) \stackrel{\text{def}}{=} \inf\{r > 0 \mid y \in B_r(x)\}$ (`edist` in `[1, file normedtype.v]`) and the extended real-valued function $d'(A, z) \stackrel{\text{def}}{=} \inf\{d(z, a) \mid a \in A\}$ (`edist_inf` in `[1, file normedtype.v]`). The function d is continuous on the gauge uniform space. Since it is coarser than the topology on T , d is continuous on $T \times T$, and thus d' is continuous on T . It follows that $f(x) \stackrel{\text{def}}{=} \min(d'(A, x), \varepsilon)/\varepsilon$ is continuous. It also takes 0 on A , and 1 on B , and has range $[0, 1]$, which means that we have the separating function and thus a proof of `uniform_separatorP`. See lemma `urysohn_separation` in `[1, file normedtype.v]` for details.

5:12 A Comprehensive Overview of the Lebesgue Differentiation Theorem in Coq

We now need to show `normal_uniform_separator`. For that purpose, we need to build a suitable uniform structure on a normal space. The uniformity we construct has the following basis:

```

Let apxU (UV : set T * set T) : set (T * T) :=
  (UV.2 × UV.2) ∪ ((closure UV.1)ᶜ × (closure UV.1)ᶜ).
Let nested (UV : set T * set T) :=
  [/\ open UV.1, open UV.2, A ⊆ UV.1 & closure UV.1 ⊆ UV.2].
Let ury_base := [set apxU UV | UV in nested].

```

Most of the work is to show that this is a basis for a uniformity, and that it is coarser than the topology on T . Then given an A and a B which are closed, disjoint, and non-empty, normality guarantees an open set $U \supseteq A$ with $U \cap B = \emptyset$. Then `apxU (U, Bᶜ)` is a separator for A and B and we have our `uniform_separator`. See [1, file `normedtype.v`] for details.

8 Lusin and Tietze theorems and continuous functions are dense in L^1

As sketched in Sect. 4, one important step to prove the Lebesgue Differentiation theorem is to establish that continuous functions are dense in L^1 . As we explained in Sect. 4, we can get along with a formal proof considering only functions defined over a bounded set. As an intermediate step, we formalize Lusin's theorem and Tietze's extension theorem.

8.1 Lusin's theorem

We place ourselves in a context with $R : \text{realType}$ to represent a type of reals and where μ is the Lebesgue measure. Lusin's theorem states that, given a measurable function f over A (a measurable bounded set) and $\varepsilon > 0$, there exists a compact $K \subseteq A$ such that $\mu(K \setminus A) < \varepsilon$ and f is continuous within K [1, file `lebesgue_integral.v`]:

```

Lemma measurable_almost_continuous (f : R -> R) (A : set R) (eps : R) :
  measurable A -> mu A < +oo -> measurable_fun A f ->
  0 < eps -> exists K,
  [/\ compact K, K ⊆ A, mu (A \ K) < eps%:E & {within K, continuous f}].

```

The notation `{within _, continuous _}` was explained along the formalization of subspace topologies in Sect. 5.1. The proof also uses the following lemma that pertains to subspace topologies as well: if f and g are equal on A and f is continuous then so is g . Put formally:

```

Lemma subspace_eq_continuous {S : topologicalType} (f g : subspace A -> S) :
  {in A, f =1 g} -> continuous f -> continuous g.

```

The proof connects to results presented earlier in this paper: Egorov's theorem (Sect. 6.1) and the (bounded version of the) inner regularity of Lebesgue measurable (Sect. 6.2.2).

8.2 Tietze's extension theorem

Tietze's extension theorem states that in a normal topological space (normality being already defined in Sect. 7), a bounded, continuous, real-valued function on a closed set can be extended to a bounded, continuous function on the whole set. Although we do formalize Tietze's theorem for normal spaces, it should be noted that the normality condition is incidental to the main results of this paper; what is relevant here is that the reals are normal. Here follows the statement of Tietze's theorem in `MATHCOMP-ANALYSIS` [1, file `numfun.v`]:

Context {X : topologicalType} {R : realType}.

Hypothesis normalX : normal_space X.

Lemma continuous_bounded_extension (f : X → ℝ^o) (A : set X) M :
 closed A → {within A, continuous f} →
 0 < M → (forall x, A x → `|f x| ≤ M) →
 exists g, [/\ {in A, f =1 g}, continuous g & forall x, `|g x| ≤ M].

The notation ^o is only to help type-checking. Besides Urysohn's lemma, this proof uses the fact that uniform convergence preserves continuity (lemma `uniform_limit_continuous` in [1, file `function_spaces.v`]). The hypothesis `{within A, continuous f}` is a typical detail that does not appear in a textbook where this theorem would be assuming that the function `f` is continuous on `A`.

8.3 Continuous functions are dense in L^1

Finally, we arrive at the true goal of this section: the fact that continuous functions are dense in L^1 , i.e., that given a function f integrable over a measurable, bounded set A , there exists a sequence of continuous functions g_k , integrable over A , such that $\|f - g_k\|_1$ tends towards 0:

Lemma approximation_continuous_integrable (A : set _) (f : ℝ → ℝ) :
 measurable A → mu A < +∞ → mu.-integrable A (EFin \o f) →
 exists g_ : (ℝ → ℝ)^{nat},
 [/\ forall n, continuous (g_ n),
 forall n, mu.-integrable A (EFin \o g_ n) &
 \int[mu]_(z in A) `(f z - g_ n z)%:E| @[n --> \∞] --> 0].

The proof uses Tietze's and Lusin's theorems, see [1, file `lebesgue_integral.v`]. As we explained in Sect. 4, we use the above lemma to produce a sequence of continuous functions g_i to be used in the “bounded version” of the Lebesgue Differentiation theorem for a real-valued function restricted to some ball B_k . The desired sequence of g_i 's is obtained by the above lemma modulo the technical detail that we need to restrict them to B_k for them to connect correctly to other lemmas used in the proof of the Lebesgue Differentiation theorem.

9 Covering lemmas and the Hardy-Littlewood maximal inequality

As we explained in Sect. 4, the second important step to prove the Lebesgue Differentiation theorem is the Hardy-Littlewood maximal inequality, i.e., the fact that, for all locally integrable functions f , $\mu(\{x \mid \text{HL}(f)(x) > c\}) \leq \frac{3}{c} \|f\|_1$ for all $c > 0$. Its proof relies on a covering lemma typical of measure theory.

9.1 Vitali's covering lemma

In its finite version, the Vitali covering lemma can be stated as follows: given a finite collection of balls B_i with $i \in s$, there exists a subcollection B_j with $j \in D$ of pairwise disjoint balls such that $\bigcup_{i \in s} B_i \subseteq \bigcup_{j \in D} 3B_j$. To formalize this statement without committing to a concrete representation for collection of balls, we represent them by a function $B : I \rightarrow \text{set } \mathbb{R}$ such that each set satisfies a predicate `is_ball`, instead of representing them, say, as a function returning pairs of a center and a radius. The approach using the `is_ball` predicate gives rise to two functions `cpoint` and `radius` returning respectively a center point and a non-negative radius, when the set is indeed a ball. The finiteness of the collections is captured by using lists (respectively `s` and `D` below).

```

Context {I : eqType}.
Variable (B : I -> set R).
Hypothesis is_ballB : forall i, is_ball (B i).
Hypothesis B_set0 : forall i, B i !=set0.

Lemma vitali_lemma_finite (s : seq I) : { D : seq I | [/\ uniq D,
  {subset D <= s}, trivIset [set` D] B &
  forall i, i \in s -> exists j, [/\ j \in D, B i \cap B j !=set0,
  radius (B j) >= radius (B i) & B i \subseteq 3 *` (B j)] ] }.

```

The notation $k *`$ represents scaling of the radius of a ball, i.e., $k *` B$ is the open ball with center $\text{cpoint } B$ and radius $k * \text{radius } B$.

We also formalized the infinite version of Vitali's covering lemma [1, file `normedtype.v`] and Vitali's theorem [1, file `lebesgue_measure.v`], which are much more involved. We did not need them to prove the Lebesgue Differentiation theorem but they served as a test-bed for using the `is_ball` predicate and are anyway often mentioned in connection with proofs of the FTC.

9.2 Hardy-Littlewood maximal inequality

The Hardy-Littlewood operator is a function that transforms a real-valued function f into the function

$$\text{HL}(f)(x) \stackrel{\text{def}}{=} \sup_{r>0} \frac{1}{\mu(B_r(x))} \int_{y \in B_r(x)} |f(y)| (d\mu).$$

Its formal definition uses elements similar to the ones used when defining Lebesgue points in Sect. 3:

```

Definition HL_max (f : R -> R) (x : R^o) (r : R) : \bar{R} :=
  (fine (mu (ball x r)))^-1%:E * \int[mu]_(y in ball x r) `|(f y)%:E|.
Definition HL_maximal (f : R -> R) (x : R^o) : \bar{R} :=
  ereal_sup [set HL_max f x r | r in `]0, +oo[ ].

```

The statement of the Hardy-Littlewood maximal inequality that we explained informally at the very beginning of this section (Sect. 9) then translates directly:

```

Lemma maximal_inequality (f : R -> R) c :
  locally_integrable [set: R] f -> 0 < c ->
  mu [set x | HL_maximal f x > c%:E] <= (3%:R / c)%:E * norm1 [set: R] f.

```

The L^1 norm is formalized in the obvious way as the identifier `norm1`. The proof relies on inner regularity (Sect. 6.2.2) and Vitali's covering lemma (Sect. 9.1). To establish that the Hardy-Littlewood operator is measurable, we also need to develop a theory of lower semicontinuity, which has been added to MATHCOMP-ANALYSIS on this occasion, see [1] for details.

10 Applications of the Lebesgue Differentiation theorem

In the previous sections, we have explained the main lemmas (mainly: continuous functions are dense in L^1 and the Hardy-Littlewood maximal inequality) used to prove the Lebesgue Differentiation theorem that we sketched in Sect. 4. We are now ready to proceed to direct applications.

10.1 The first FTC for Lebesgue integration

Recall from Sect. 1 the informal statement of the first FTC for Lebesgue integration: for $f \in L^1$, $F(x) \stackrel{\text{def}}{=} \int_{t \in]-\infty, x]} f(t) (\mathbf{d}\mu)$ is differentiable and $F'(x) \stackrel{\text{a.e.}}{=} f(x)$. This can furthermore be generalized to intervals of the form $]a, x]$ and $[a, x]$ and stated as a single theorem as follows [1, file `ftc.v`]:

```
Lemma FTC1_lebesgue_pt f a : mu.-integrable [set: R] (EFin \o f) ->
  let F x := (\int[mu]_(t in [set` Interval a (BRight x)]) (f t))%R in
  forall x, a < BRight x -> lebesgue_pt f x ->
  derivable F x 1 /\ F^`() x = f x.
```

The variable `a` has the generic type of an “interval bound” and `BRight` stands for closed bounds on the right [11, file `interval.v`]. The predicate `derivable` is for derivability (1 is the direction) and the notation `^`()` is for derivatives with domain \mathbb{R} [3, §4.5]. The theorem above connects these notions with the Lebesgue integral developed in `MATHCOMP-ANALYSIS` independently [2, §6.4]. The proof is standard in that it goes through a generalization of the Lebesgue Differentiation theorem where balls are replaced with *nice* shrinking sets [16, §II.4.1]. The statement of the first FTC from Sect. 1 is an immediate corollary of the above lemma:

```
Corollary FTC1Ny f : mu.-integrable setT (EFin \o f) ->
  let F x := (\int[mu]_(t in [set` `]-oo, x]) (f t))%R in
  {ae mu, forall x, derivable F x 1 /\ F^`() x = f x}.
```

10.2 Lebesgue’s density theorem

Lebesgue’s density theorem is another direct consequence of the Lebesgue Differentiation theorem. The *density* of a point x w.r.t. a set A is defined by $\lim_{r \rightarrow 0^+} \frac{\mu(A \cap B_r(x))}{\mu(B_r(x))}$. Lebesgue’s density theorem states that almost everywhere the density is 0 or 1:

```
Lemma density (A : set R) : measurable A ->
  {ae mu, forall x, mu (A \cap ball x r) * (fine (mu (ball x r)))^-1%:E
    @[r --> 0^'+] --> (\1_A x)%:E}.
```

11 Related work

We have been using various documents to formalize the Lebesgue Differentiation theorem. In particular, the main lines are drawn from lecture notes by Bowen [8]. For the proofs of the Hardy-Littlewood maximal inequality and the proof of the Lebesgue Differentiation theorem, we used books by Li [16] and Schwartz [28]. Surely, the same contents can be found elsewhere.

Several lemmas that we discussed can also be found in `Mathlib` [30]. Of course, the proof of Urysohn’s lemma in `Mathlib` [23] is different from ours, which is original, as we explained in Sect. 7. Tietze’s extension theorem in `Mathlib` [22, class `TietzeExtension`] has a similar statement and a similar proof. The statement of the Lebesgue Differentiation theorem in `Mathlib` [19] is more general than ours: it allows the domain of the function to be an arbitrary metric space, the measure can be any locally finite measure, the codomain can be any normed abelian group, and the balls (used in `davg` in Sect. 3) can be replaced by an arbitrary Vitali family. The Lebesgue Differentiation theorem in `Mathlib` is also used to prove a generic version of Lebesgue’s density theorem [18] [26, §3.2].

■ **Table 1** Estimated lines of code for the formalization of the Lebesgue Differentiation theorem and its direct applications.

(the column l.o.c. contains the number of lines of code in proof scripts for the main proof and intermediate lemmas (including statements); these numbers are approximations because, among other reasons, where one draws the line between intermediate lemmas and the supporting theories can be arbitrary)

<i>Supporting theories</i>	Section	l.o.c.		file in [1]
Subspaces	Sect. 5.1	N.A.		<code>topology.v</code>
Uniform convergence	Sect. 5.2	N.A.		<code>function_spaces.v</code>
<i>Main lemmas</i>	Section	l.o.c.		file in [1]
Egorov's thm	Sect. 6.1	≈ 87	(2 lemmas)	<code>lebesgue_measure.v</code>
Outer regularity	Sect. 6.2.1	≈ 61	(1 lemma)	<code>lebesgue_measure.v</code>
Inner regularity	Sect. 6.2.2	≈ 118	(4 lemmas)	<code>lebesgue_measure.v</code>
Lusin's thm	Sect. 8.1	≈ 108	(3 lemmas)	<code>lebesgue_integral.v</code>
Tietze's extension thm	Sect. 8.2	≈ 108	(3 lemmas)	<code>numfun.v</code>
Density of cont. functions	Sect. 8.3	≈ 118	(3 lemmas)	<code>lebesgue_integral.v</code>
Finite Vitali's covering lem.	Sect. 9.1	≈ 75	(2 lemmas)	<code>normedtype.v</code>
Hardy-Littlewood max. ineq.	Sect. 9.2	≈ 180	(6 lemmas)	<code>lebesgue_integral.v</code>
Urysohn's lemma	Sect. 7.1	≈ 165	(11 lemmas)	<code>normedtype.v</code>
Lebesgue Differentiation thm	Sect. 4	≈ 143	(3 lemmas)	<code>lebesgue_integral.v</code>
First FTC	Sect. 10.1	≈ 265	(4 lemmas)	<code>ftc.v</code>
Lebesgue's density thm	Sect. 10.2	≈ 69	(1 lemma)	<code>lebesgue_integral.v</code>
Total (<i>Main lemmas</i>)		≈ 1,497		

The FTC has already been the target of several formalizations in proof assistants. It can be found in COQ but for the Riemann integral in a constructive setting [10, §6]. NASALib does not feature the first FTC for Lebesgue integration but an elementary version (for a C^1 function) of the second FTC [25, file `lebesgue_fundamental.pvs`], which can actually be obtained from the first FTC for Lebesgue integration as a corollary. Isabelle/HOL features the first FTC for Lebesgue integration but for continuous functions whereas we prove it for integrable functions [6, §3.7]. Mathlib features several variants of the first FTC; many require integrability and continuity at the endpoints but establish strict differentiability [20]. They stem from a lemma analogous to a strengthening of the Lebesgue Differentiation theorem with nicely shrinking sets [21]. In other words, we are able to match our lemmas with Mathlib lemmas but statements and proofs are organized in a different way. However, it must be said that Mathlib's statements are admittedly more general than ours in many respects. One reason is that MATHCOMP-ANALYSIS has started to use HIERARCHY-BUILDER pervasively only recently. Before that, mathematical structures were manually encoded with *packed classes* [12]: this was making modifications very difficult in practice. With HIERARCHY-BUILDER, we believe that introduction of, say, Banach spaces, should be a matter of engineering because most of our proofs are textbook, because we do not abuse the fact that we are working on the real line, and because our development is short enough to be refactored (see Table 1 for a concrete size estimation).

12 Conclusions

In this paper, we provided a comprehensive overview of the Lebesgue Differentiation theorem. We started with a formal statement and a proof overview (in Sect. 3 and in Sect. 4) to plan the whole development (as summarized in Fig. 1). Before being able to even state the first

intermediate lemmas, we needed to extend MATHCOMP-ANALYSIS with in particular new topological constructs in Sect. 5. This made it possible to formalize the needed basic lemmas from measure theory in Sect. 6. Among the needed lemma, we formalized in particular a novel proof of Urysohn’s lemma in Sect. 7. We used all this material to formalize the main steps of the proof of the Lebesgue Differentiation theorem: the density of continuous functions in L^1 in Sect. 8 and Hardy-Littlewood maximal inequality in Sect. 9. Our formalization of the Lebesgue Differentiation theorem was completed by two applications in Sect. 10, which include the first proof of the first FTC for Lebesgue integration for the COQ proof assistant.

In the end, we provide in a single document a complete overview of an important theorem. We believe that this experiment also concretely illustrates an important aspect of formalization of mathematics: the Lebesgue Differentiation theorem, like the uniform separators of Sect. 7.1, are examples of results whose formalization should be prioritized because, though technical, they are generic intermediate results from which important results can be obtained as corollaries (here, the first FTC and Urysohn’s lemma). More pragmatically, we hope that this overview also contributes to documenting formalization of real analysis with MATHCOMP-ANALYSIS, for example by explaining the use of HIERARCHY-BUILDER to develop topology. We think that we demonstrated that MATHCOMP-ANALYSIS is already a rich library and also a useful tool to formalize mathematics. As a matter of fact, we could use it to revisit Urysohn’s lemma by producing an original proof.

As for future work, we are now working on the second FTC for Lebesgue integration whose most general form deals with *absolutely continuous functions* [24], using as the main ingredient the Radon-Nikodým theorem already available in MATHCOMP-ANALYSIS [14] and the recently developed theory of bounded and total variation [1, file `realfun.v`].

References

- 1 Reynald Affeldt, Yves Bertot, Alessandro Bruni, Cyril Cohen, Marie Kerjean, Assia Mahboubi, Damien Rouhling, Pierre Roux, Kazuhiko Sakaguchi, Zachary Stone, Pierre-Yves Strub, and Laurent Théry. MathComp-Analysis: Mathematical Components compliant analysis library. <https://github.com/math-comp/analysis>, 2017. Last stable version: 1.2.0 (2024).
- 2 Reynald Affeldt and Cyril Cohen. Measure construction by extension in dependent type theory with application to integration. *J. Autom. Reason.*, 67(3):28:1–28:27, 2023.
- 3 Reynald Affeldt, Cyril Cohen, and Damien Rouhling. Formalization techniques for asymptotic reasoning in classical analysis. *J. Formaliz. Reason.*, 11(1):43–76, 2018. doi:10.6092/issn.1972-5787/8124.
- 4 Reynald Affeldt, Cyril Cohen, and Ayumu Saito. Semantics of probabilistic programs using s-finite kernels in Coq. In *12th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2023), Boston, MA, USA, January 16–17, 2023*, pages 3–16. ACM, 2023. doi:10.1145/3573105.3575691.
- 5 Reynald Affeldt and Zachary Stone. Towards the fundamental theorem of calculus for the Lebesgue integral in Coq. In *35ème Journées Francophones des Langages Applicatifs (JFLA 2024), Saint-Jacut-de-la-Mer, France, January 30–February 2, January 2024*. open access.
- 6 Jeremy Avigad, Johannes Hölzl, and Luke Serafin. A formally verified proof of the central limit theorem. *J. Autom. Reason.*, 59(4):389–423, 2017. doi:10.1007/s10817-017-9404-x.
- 7 Sylvie Boldo, Catherine Lelay, and Guillaume Melquiond. Coquelicot: A user-friendly library of real analysis for Coq. *Math. Comput. Sci.*, 9(1):41–62, 2015. doi:10.1007/S11786-014-0181-1.
- 8 Lewis Bowen. Lecture notes in real analysis. Available at <https://web.ma.utexas.edu/users/lpbowen/m381c/lecture-notes.pdf>, December 2014. University of Texas at Austin.
- 9 Cyril Cohen, Kazuhiko Sakaguchi, and Enrico Tassi. Hierarchy builder: Algebraic hierarchies made easy in Coq with Elpi (system description). In *5th International Conference on Formal*

- Structures for Computation and Deduction (FSCD 2020)*, June 29–July 6, 2020, Paris, France (Virtual Conference), volume 167 of *LIPICs*, pages 34:1–34:21. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPICs.FSCD.2020.34.
- 10 Luís Cruz-Filipe. A constructive formalization of the fundamental theorem of calculus. In *Selected Papers of the 2nd International Workshop on Types for Proofs and Programs (TYPES 2002)*, Berg en Dal, The Netherlands, April 24–28, 2002, volume 2646 of *Lecture Notes in Computer Science*, pages 108–126. Springer, 2002. doi:10.1007/3-540-39185-1_7.
 - 11 The MathComp development team. Mathematical components. <https://github.com/math-comp/math-comp>, 2005. Last stable version: 2.2.0 (2024).
 - 12 François Garillot, Georges Gonthier, Assia Mahboubi, and Laurence Rideau. Packaging mathematical structures. In *22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2009)*, Munich, Germany, August 17–20, 2009, volume 5674 of *Lecture Notes in Computer Science*, pages 327–342. Springer, 2009. doi:10.1007/978-3-642-03359-9_23.
 - 13 Johannes Hölzl, Fabian Immler, and Brian Huffman. Type classes and filters for mathematical analysis in Isabelle/HOL. In *4th International Conference on Interactive Theorem Proving (ITP 2013)*, Rennes, France, July 22–26, 2013, volume 7998 of *Lecture Notes in Computer Science*, pages 279–294. Springer, 2013. doi:10.1007/978-3-642-39634-2_21.
 - 14 Yoshihiro Ishiguro and Reynald Affeldt. The Radon-Nikodým theorem and the Lebesgue-Stieltjes measure in Coq. *Computer Software*, 41(2), 2024. Japan Society for Software Science and Technology.
 - 15 Samuel Leland Lesseig. Metrization of uniform spaces. Master’s thesis, Department of Mathematics, Kansas State University, Manhattan, Kansas, 1963.
 - 16 Daniel Li. *Notions fondamentales d’analyse réelle et complexe—Espaces de Hardy et interpolation, avec exercices corrigés*. Ellipses, 2022.
 - 17 Assia Mahboubi and Enrico Tassi. *Mathematical Components*. Zenodo, January 2021. doi:10.5281/zenodo.4457887.
 - 18 Mathlib 4. File `MeasureTheory/Covering/DensityTheorem.lean`. [url](#), June 2024.
 - 19 Mathlib 4. File `MeasureTheory/Covering/Differentiation.lean`. [url](#), June 2024.
 - 20 Mathlib 4. File `MeasureTheory/Integral/FundThmCalculus.lean`. [url](#), June 2024.
 - 21 Mathlib 4. File `MeasureTheory/Integral/SetIntegral.lean`. [url](#), June 2024.
 - 22 Mathlib 4. File `Topology/TietzeExtension.lean`. [url](#), June 2024.
 - 23 Mathlib 4. File `Topology/UrysohnsLemma.lean`. [url](#), June 2024.
 - 24 Maran Mohanarangan. The fundamental theorem of calculus for Lebesgue integration. Technical report, ETH Zürich, 2021. Exercise Class 13, Measure and Integration, Spring 2021, available at https://metaphor.ethz.ch/x/2021/fs/401-2284-00L/sc/notes_exclass13.pdf.
 - 25 NASALib. NASA PVS library of formal developments. Current version: 7.1.1. Available at <https://github.com/nasa/pvslib>, 2023.
 - 26 Oliver Nash. A Formalisation of Gallagher’s Ergodic Theorem. In *14th International Conference on Interactive Theorem Proving (ITP 2023)*, volume 268 of *Leibniz International Proceedings in Informatics (LIPICs)*, pages 23:1–23:16, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPICs.ITP.2023.23.
 - 27 Ayumu Saito and Reynald Affeldt. Experimenting with an intrinsically-typed probabilistic programming language in Coq. In *21st Asian Symposium on Programming Languages and Systems (APLAS 2023)*, Taipei, Taiwan, November 26–29, 2023, volume 14405, pages 182–202. Springer, 2023. doi:10.1007/978-981-99-8311-7_9.
 - 28 Laurent Schwartz. *Analyse III: Calcul intégral*. Hermann, 1997.
 - 29 The Coq Development Team. *The Coq Proof Assistant Reference Manual*. Inria, 2024. Available at <https://coq.inria.fr>. Version 8.19.2.
 - 30 The mathlib Community. The lean mathematical library. In *9th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2020)*, New Orleans, LA, USA, January 20–21, 2020, pages 367–381. ACM, 2020. doi:10.1145/3372885.3373824.

- 31 Li Zhou, Gilles Barthe, Pierre-Yves Strub, Junyi Liu, and Mingsheng Ying. CoqQ: Foundational verification of quantum programs. *Proc. ACM Program. Lang.*, 7(POPL):833–865, 2023. doi:10.1145/3571222.

Towards Solid Abelian Groups: A Formal Proof of Nöbeling’s Theorem

Dagur Asgeirsson   

University of Copenhagen, Denmark

Abstract

Condensed mathematics, developed by Clausen and Scholze over the last few years, is a new way of studying the interplay between algebra and geometry. It replaces the concept of a topological space by a more sophisticated but better-behaved idea, namely that of a condensed set. Central to the theory are solid abelian groups and liquid vector spaces, analogues of complete topological groups.

Nöbeling’s theorem, a surprising result from the 1960s about the structure of the abelian group of continuous maps from a profinite space to the integers, is a crucial ingredient in the theory of solid abelian groups; without it one cannot give any nonzero examples of solid abelian groups. We discuss a recently completed formalisation of this result in the Lean theorem prover, and give a more detailed proof than those previously available in the literature. The proof is somewhat unusual in that it requires induction over ordinals – a technique which has not previously been used to a great extent in formalised mathematics.

2012 ACM Subject Classification General and reference → Verification; Computing methodologies → Representation of mathematical objects; Mathematics of computing → Mathematical software

Keywords and phrases Condensed mathematics, Nöbeling’s theorem, Lean, Mathlib, Interactive theorem proving

Digital Object Identifier 10.4230/LIPIcs.ITP.2024.6

Supplementary Material Software: <https://github.com/leanprover-community/mathlib4/blob/ba9f2e5baab51310883778e1ea3b48772581521c/Mathlib/Topology/Category/Profinite/Nobeling.lean> archived at `swh:1:cnt:2fb2985994d43409a52761d0e853d37deeabdc74`

Funding *Dagur Asgeirsson*: The author was supported by the Danish National Research Foundation (DNRF) through the “Copenhagen Center for Geometry and Topology” under grant no. DNRF151.

Acknowledgements First and foremost, I would like to thank Johan Commelin for encouraging me to start seriously working on this project when we were both in Banff attending the workshop on formalisation of cohomology theories last year. I had useful discussions related to this work with Johan, Kevin Buzzard, Adam Topaz, and Dustin Clausen. I am indebted to all four of them for providing helpful feedback on earlier drafts of this paper. Any project formalising serious mathematics in Lean depends on Mathlib, and this one is no exception. I am grateful to the Lean community as a whole for building and maintaining such a useful mathematical library, as well as providing an excellent forum for Lean-related discussions through the Zulip chat. Finally, I would like to thank the anonymous referees for helpful feedback.

1 Introduction

Nöbeling’s theorem says that the abelian group $C(S, \mathbb{Z})$ of continuous maps from a profinite space S to the integers, is a free abelian group. In fact, the original statement [14, Satz 1] is the more general result that bounded maps from any set to the integers form a free abelian group, but this special case has recently been applied [16, Theorem 5.4] in the new field of condensed mathematics (see also [15, 5, 3]).

We report on a recently completed formalisation of this theorem using the Lean 4 theorem prover [12], building on its *Mathlib* library of formalised mathematics (which was recently ported from Lean 3, see [11, 10]). The proof uses the well-ordering principle and a tricky



© Dagur Asgeirsson;

licensed under Creative Commons License CC-BY 4.0

15th International Conference on Interactive Theorem Proving (ITP 2024).

Editors: Yves Bertot, Temur Kutsia, and Michael Norrish; Article No. 6; pp. 6:1–6:17

Leibniz International Proceedings in Informatics




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

induction over ordinals. This is the first use of the induction principle for ordinals in Mathlib outside the directory containing the theory of ordinals. Often, one can replace such transfinite constructions by appeals to Zorn’s lemma. The author is not aware of any proof of Nöbeling’s theorem that does this, or otherwise avoids induction over ordinals¹.

When formalising a nontrivial proof, one inevitably makes an effort to organise the argument carefully. One purpose of this paper is to give a well-organised and detailed proof of Nöbeling’s theorem, written in conventional mathematical language, which is essentially a by-product of the formalisation effort. This will hopefully be a more accessible proof than those that already exist in the literature; the one in [14] is in German, while the proofs of the result in [7, 16] are the same argument as the one presented here, but in significantly less detail. This is the content of section 4; some mathematical prerequisites are found in section 3.

In section 2 we give more details about the connection to condensed mathematics and in sections 5 and 6 we discuss the formalisation process and the integration into Mathlib.

Throughout the text, we use the symbol “” for external links, usually directly to the source code for the corresponding theorems and definitions in Mathlib. In order for the links to stay usable, they are all to a fixed commit to the master branch (the most recent one at the time of writing).

Mathlib is a growing library of mathematics formalised in Lean. All material is maintained continuously by a team of experts. There is a big emphasis on unity, meaning that there is *one* official definition of every concept, and it is the job of contributors to provide proofs that alternative definitions are equivalent. All the code in this project has been integrated into Mathlib; a process that took quite some time, as high standards are demanded of code that enters the library. However, it is an important part of formalisation to get the code into Mathlib, because doing so means that it stays usable to others in the future.

2 Motivation

Condensed mathematics [16, 15, 5] is a new theory developed by Clausen and Scholze (and independently by Barwick and Haine, who called the theory *pyknotic sets* [3]). It has the purpose of generalising topology in a way that gives better categorical properties, which is desirable e.g. when the objects have both a topological and an algebraic structure. Condensed objects² can be described as sheaves on a certain site of profinite spaces. A topological abelian group A can be regarded as a condensed abelian group with S -valued points $C(S, A)$ for profinite spaces S . Discrete abelian groups such as \mathbb{Z} are important examples of topological abelian groups. There is a useful characterisation of discrete condensed sets (which leads to the same characterisation for more general condensed objects such as condensed abelian groups), which has been formalised in Lean 3 by the author in [1].

The discreteness characterisation can be stated somewhat informally as follows: A condensed set X is discrete if and only if for every profinite space $S = \varprojlim_i S_i$ (written as a cofiltered limit of finite discrete spaces), the natural map

$$\varinjlim_i X(S_i) \rightarrow X(S)$$

is an isomorphism.

¹ The proof does not use any ordinal arithmetic. However, it crucially uses the principle of induction over ordinals with a case split between limit ordinals and successor ordinals.

² This notion was first formalised in the *Liquid Tensor Experiment* [6, 17], see section 6 for a more detailed discussion.

There is a notion of completeness of condensed abelian groups, called being *solid* [16, Definition 5.1]. For the convenience of the reader, we give the informal definition here in Definition 1. First, we need to recall two facts about condensed abelian groups:

- The category of condensed abelian groups has all limits.
- The forgetful functor from condensed abelian groups to condensed sets has a left adjoint, denoted by $\mathbb{Z}[-]$ (adopted from the analogous relationship between the category of sets and the category of abelian groups).

► **Definition 1.** Let $S = \varprojlim_i S_i$ be a profinite space and define a condensed abelian group as follows:

$$\mathbb{Z}[S]^{\blacksquare} := \varprojlim_i \mathbb{Z}[S_i]$$

There is a natural map $\mathbb{Z}[S] \rightarrow \mathbb{Z}[S]^{\blacksquare}$, and we say that a condensed abelian group A is solid if for every profinite space S and every morphism $f : \mathbb{Z}[S] \rightarrow A$ of condensed abelian groups, there is a unique morphism $g : \mathbb{Z}[S]^{\blacksquare} \rightarrow A$ making the obvious triangle commute³.

Using the discreteness characterisation and Nöbeling's theorem, one can prove that for every profinite space S , there is a set I and an isomorphism of condensed abelian groups

$$\mathbb{Z}[S]^{\blacksquare} \cong \prod_{i \in I} \mathbb{Z}.$$

This structural result is essential to developing the theory of solid abelian groups. Without it one cannot even prove the existence of a nontrivial solid abelian group.

Since the proof of Nöbeling's theorem has nothing to do with condensed mathematics, people studying the theory might be tempted to skip the proof and use Nöbeling's theorem as a black box. Now that it has been formalised, they can do this with a better conscience. On the other hand, people interested in understanding the proof might want to turn to sections 3 and 4 of this paper for a more detailed account.

3 Preliminaries

For ease of reference, we collect in this section some prerequisites for the proof of Nöbeling's theorem. Most of them were already in Mathlib.

3.1 Order theory

► **Definition 2.** \square Let I and X be sets and let r be a binary relation on X . An I -indexed family (x_i) in X is directed if for all $i, j \in I$, there exists $k \in I$ such that $r(x_i, x_k)$ and $r(x_j, x_k)$.

► **Lemma 3.** \square A monotone map on a poset with a join operation (i.e. a least upper bound of two elements) is directed.

► **Remark 4.** Taking the union of two sets is an example of a join operation.

► **Definition 5.** \square A category \mathcal{C} is filtered if it satisfies the following three conditions
(i) \mathcal{C} is nonempty.

³ This definition has also been formalised by the author in Lean 3 in [2]

6:4 Towards Solid Abelian Groups: A Formal Proof of Nöbeling's Theorem

- (ii) For all objects X, Y , there exists an object Z and morphisms $f : X \rightarrow Z$ and $g : Y \rightarrow Z$.
- (iii) For all objects X, Y and all morphisms $f, g : X \rightarrow Y$, there exists an object Z and a morphism $h : Y \rightarrow Z$ such that $h \circ f = h \circ g$.

A category is cofiltered if the opposite category is filtered.

- Remark 6. A poset is filtered if and only if it is nonempty and directed.
- Remark 7. The poset of finite subsets of a given set is filtered.

3.2 Linear Independence

► **Lemma 8.** \square If (X_i) is a family of linearly independent subsets of a module over a ring R , which is directed with respect to the subset relation, then its union is linearly independent.

► **Lemma 9.** \square Suppose we have a commutative diagram

$$\begin{array}{ccccccc}
 0 & \longrightarrow & N & \xrightarrow{f} & M & \xrightarrow{g} & P \\
 & & \uparrow v & & \uparrow u & & \uparrow w \\
 & & I & \longleftarrow & I \sqcup J & \longrightarrow & J
 \end{array}$$

where N, M, P are modules over a ring R , the top row is exact, and the bottom maps are the inclusion maps. If v and w are linearly independent, then u is linearly independent.

3.3 Cantor's intersection theorem

► **Theorem 10.** \square Cantor's intersection theorem. If $(Z_i)_{i \in I}$ is a nonempty family of nonempty, closed and compact subsets of a topological space X , which is directed with respect to the superset relation $(V, W) \mapsto V \supseteq W$, then the intersection $\bigcap_{i \in I} Z_i$ is nonempty.

► Remark 11. Cantor's intersection theorem is often stated only for the special case of decreasing nested sequences of nonempty compact, closed subsets. The generalisation above can be proved by slightly modifying the standard proof of that special case.

3.4 Cofiltered limits of profinite spaces

► **Definition 12.** \square A profinite space is a totally disconnected compact Hausdorff space.

► **Lemma 13.** \square Every profinite space has a basis of clopen subsets.

► **Lemma 14.** \square Every profinite space is totally separated, i.e. any two distinct points can be separated by clopen neighbourhoods.

► Remark 15. A topological space is profinite if and only if it can be written as a cofiltered limit of finite discrete spaces. See section 6 for a further discussion.

► **Lemma 16.** \square Any continuous map from a cofiltered limit of profinite spaces to a discrete space factors through one of the components.


► Remark 17. In particular, a continuous map from a profinite space

$$S = \varprojlim_i S_i$$

to a discrete space factors through one of the finite quotients S_i .

4 The theorem


This section is devoted to proving

► **Theorem 18.**  (Nöbeling's theorem). *Let S be a profinite space. Then the abelian group $C(S, \mathbb{Z})$ of continuous maps from S to \mathbb{Z} is free.*

We can immediately reduce this to proving Lemma 19 below as follows: Let I denote the set of clopen subsets of S . Then the map

$$S \rightarrow \prod_{i \in I} \{0, 1\}$$

whose i -th projection is given by the indicator function of the clopen subset i is a closed embedding.

► **Lemma 19.**  *Let I be a set and let S be a closed subset of $\prod_{i \in I} \{0, 1\}$. Then $C(S, \mathbb{Z})$ is a free abelian group.*

To prove Lemma 19, we need to construct a basis of $C(S, \mathbb{Z})$. Our proposed basis is defined as follows:

- Choose a well-ordering on I .
- Let $e_{S,i} \in C(S, \mathbb{Z})$ denote the composition

$$S \hookrightarrow \prod_{i \in I} \{0, 1\} \xrightarrow{p_i} \{0, 1\} \hookrightarrow \mathbb{Z}$$

where p_i denotes the i -th projection map, and the other two maps are the obvious inclusions.

- Let P denote the set of finite, strictly decreasing sequences in I . Order these lexicographically.
- Let $\text{ev}_S : P \rightarrow C(S, \mathbb{Z})$ denote the map

$$(i_1, \dots, i_r) \mapsto e_{S,i_1} \cdots e_{S,i_r}.$$

- For $p \in P$, let $\Sigma_S(p)$ denote the span in $C(S, \mathbb{Z})$ of the set

$$\text{ev}_S(\{q \in P \mid q < p\}).$$

- Let $E(S)$ denote the subset of P consisting of those elements whose evaluation cannot be written as a linear combination of evaluations of smaller elements of P , i.e.

$$E(S) := \{p \in P \mid \text{ev}_S(p) \notin \Sigma_S(p)\}.$$

In Subsection 4.2 we prove that the set $\text{ev}_S(E(S))$ spans $C(S, \mathbb{Z})$, and in Subsection 4.3 we prove that the family

$$\text{ev}_S : E(S) \rightarrow C(S, \mathbb{Z})$$

is linearly independent, concluding the proof of Nöbeling's theorem. Subsection 4.1 defines some notation which will be convenient for bookkeeping in the subsequent proof.

4.1 Notation and generalities

For a subset J of I we denote by

$$\pi_J : \prod_{i \in I} \{0, 1\} \rightarrow \prod_{i \in I} \{0, 1\}$$

the map whose i -th projection is p_i if $i \in J$, and 0 otherwise. These maps are continuous, and since source and target are compact Hausdorff spaces, they are also closed. Given a subset $S \subseteq \prod_{i \in I} \{0, 1\}$, we let

$$S_J := \pi_J(S).$$

We can regard I with its well-ordering as an ordinal. Then I is the set of all strictly smaller ordinals. Given an ordinal μ , we let

$$\pi_\mu := \pi_{\{i \in I \mid i < \mu\}}$$

and


$$S_\mu := S_{\{i \in I \mid i < \mu\}}.$$

These maps induce injective \mathbb{Z} -linear maps

$$\pi_J^* : C(S_J, \mathbb{Z}) \rightarrow C(S, \mathbb{Z})$$


by precomposition.

Recall that we have defined P as the set of finite, strictly decreasing sequences in I , ordered lexicographically. We will use this notation throughout the proof of Nöbeling's theorem.

► **Lemma 20.**  For $p \in P$ and $x \in S$, we have

$$\text{ev}_S(p)(x) = \begin{cases} 1 & \text{if } \forall i \in p, x_i = 1 \\ 0 & \text{otherwise.} \end{cases}$$


Proof. Obvious. ◀


► **Lemma 21.**  Let J be a subset of I and let $p \in P$ be such that $i \in p$ implies $i \in J$. Then $\pi_J^*(\text{ev}_{S_J}(p)) = \text{ev}_S(p)$.

Proof. Since $i \in p$ implies $i \in J$, we have

$$x_i = \pi_J^*(x)_i$$

for all $x \in S$ and $i \in p$. The result now follows from Lemma 20. ◀

► **Remark 22.**  The hypothesis in Lemma 21 holds in particular if $p \in E(S_J)$. Indeed, suppose $i \in p$, then if $i \notin J$, we have $\text{ev}_{S_J}(p) = 0$.

► **Lemma 23.**  If μ', μ are ordinals satisfying $\mu' < \mu$, then $E(S_{\mu'}) \subseteq E(S_\mu)$.

Proof. Let $p \in E(S_{\mu'})$. Then every entry of p is $< \mu'$, and it suffices to show that if

$$\text{ev}_{S_\mu}(p) = \pi_{\mu'}^*(\text{ev}_{S_{\mu'}}(p))$$

is in the span of

$$\text{ev}_{S_\mu}(\{q \in P \mid q < p\}) = \pi_{\mu'}^*(\text{ev}_{S_{\mu'}}(\{q \in P \mid q < p\}))$$



then $\text{ev}_{S_\mu}(p)$ is in the span of $\text{ev}_{S_{\mu'}}(\{q \in P \mid q < p\})$. This follows by injectivity of $\pi_{\mu'}^*$. ◀


4.2 Span

The following series of lemmas proves that $\text{ev}_S(E(S))$ spans $C(S, \mathbb{Z})$.


► **Lemma 24.** *The set P is well-ordered.*

Proof sketch. Suppose not. Take a strictly decreasing sequence (p_n) in P . Let a_n denote first term of p_n . Then (a_n) is a decreasing sequence in I and hence eventually constant. Denote its limit by a . Let $q_n = p_n \setminus a_n$. Then there exists an N such that $(q_n)_{n \geq N}$ is a strictly decreasing sequence in P and we can repeat the process of taking the indices of the first factors, get a decreasing sequence in I whose limit is strictly smaller than a . Continuing this way, we get a strictly decreasing sequence in I , a contradiction. ◀

► **Remark 25.** The proof sketch of Lemma 24 above is ill-suited for formalisation. Kim Morrison gave a formalised proof , following similar ideas to those above, which used close to 300 lines of code. A few days later, Junyan Xu found a proof  that was ten times shorter, directly using the inductive datatype `WellFounded`. This is the only result whose proof indicated in this paper differs significantly from the one used in the formalisation.

► **Lemma 26.**  *If $\text{ev}_S(P)$ spans $C(S, \mathbb{Z})$, then $\text{ev}_S(E(S))$ spans $C(S, \mathbb{Z})$.*

Proof. It suffices to show that $\text{ev}_S(P)$ is contained in the span of $\text{ev}_S(E(S))$. Suppose it is not, and let p be the smallest element of P whose evaluation is not in the span of $\text{ev}_S(E(S))$ (this p exists by Lemma 24). Write $\text{ev}_S(p)$ as a linear combination of evaluations of strictly smaller elements of P . By minimality of p , each term of the linear combination is in the span of $\text{ev}_S(E(S))$, implying that p is as well, a contradiction. ◀

► **Lemma 27.**  *Let F denote the contravariant functor from the (filtered) poset of finite subsets of I to the category of profinite spaces, which sends J to S_J . Then S is homeomorphic to the limit of F .*

Proof sketch. Since S is compact and the limit is Hausdorff, it suffices to show that the natural map from S to the limit of F induced by the projection maps $\pi_J : S \rightarrow S_J$ is bijective.

For injectivity, let $a, b \in S$ such that $\pi_J(a) = \pi_J(b)$ for all finite subsets J of I . For all $i \in I$ we have $a_i = \pi_{\{i\}}(a) = \pi_{\{i\}}(b) = b_i$, hence $a = b$.

For surjectivity, let $b \in \lim F$. Denote by

$$f_J : \lim F \rightarrow S_J$$

the projection maps. We need to construct an element a of C such that $\pi_J(a) = f_J(b)$ for all J . In other words, we need to show that the intersection

$$\bigcap_J \pi_J^{-1}\{f_J(b)\},$$

where J runs over all finite subsets of I , is nonempty. By Cantor's intersection theorem 10, it suffices to show that this family is directed (all the fibres are closed by continuity of the π_J , and closed subsets of a compact Hausdorff space are compact). To show that it is directed, it suffices to show that for $J \subseteq K$, we have

$$\pi_K^{-1}\{f_K(b)\} \subseteq \pi_J^{-1}\{f_J(b)\}$$

(by Lemma 3). This follows easily because the transition maps in the limits are just restrictions of the π_J . ◀

► **Lemma 28.** \square *Let J be a finite subset of I . Then $\text{ev}_{S_J}(E(S_J))$ spans $C(S_J, \mathbb{Z})$.*

Proof. By lemma 26, it suffices to show that $\text{ev}_{S_J}(P)$ spans. For $x \in S_J$, denote by f_x the map $S_J \rightarrow \mathbb{Z}$ given by the Kronecker delta $f_x(y) = \delta_{xy}$.

Since S_J is finite, the set of continuous maps is actually the set of all maps, and the maps f_x span $C(S_J, \mathbb{Z})$.

Now let $j_1 > \dots > j_r$ be a decreasing enumeration of the elements of J . Let $x \in S_J$ and let e_i denote e_{S_J, j_i} if $x_{j_i} = 1$ and $(1 - e_{S_J, j_i})$ if $x_{j_i} = 0$. Then

$$f_{j_i} = \prod_{i=1}^r e_i$$

is in the span of $P(S_J)$, as desired. \blacktriangleleft

► **Lemma 29.** \square *P spans $C(S, \mathbb{Z})$.*

Proof. Let $f \in C(S, \mathbb{Z})$. Then by Lemmas 27 and 16, there is a $g \in C(S_J, \mathbb{Z})$ such that $f = \pi_J^*(g)$. Writing this g as a linear combination of elements of $E(S_J)$, by Lemma 21 we see that f is a linear combination of elements of P as desired. \blacktriangleleft

4.3 Linear independence

► **Notation 30.** *Regard I with its well-ordering as an ordinal. Let Q denote the following predicate on an ordinal $\mu \leq I$:*

For all closed subsets S of $\prod_{i \in I} \{0, 1\}$, such that for all $x \in S$ and $i \in I$, $x_i = 1$ implies $i < \mu$, $E(S)$ is linearly independent in $C(S, \mathbb{Z})$.

We want to prove the statement $Q(I)$. We prove by induction on ordinals that $Q(\mu)$ holds for all ordinals $\mu \leq I$.

► **Lemma 31.** \square *The base case of the induction, $Q(0)$, holds.*

Proof. In this case, S is empty or a singleton. If S is empty, the result is trivial. Suppose S is a singleton. We want to show that $E(S)$ consists of only the empty list, which evaluates to 1 and is linearly independent in $C(S, \mathbb{Z}) \cong \mathbb{Z}$. Let $p \in P$ and suppose p is nonempty. Then it is strictly larger than the empty list. But the evaluation of the empty list is 1, which spans $C(S, \mathbb{Z}) \cong \mathbb{Z}$, and thus $\text{ev}_S(p)$ is in the span of strictly smaller products, i.e. not in $E(S)$. \blacktriangleleft

4.3.1 Limit case

Let μ be a limit ordinal, S a closed subset such that for all $x \in S$ and $i \in I$, $x_i = 1$ implies $i < \mu$. In other words, $S = S_\mu$. Suppose $Q(\mu')$ holds for all $\mu' < \mu$. Then in particular $E(S_{\mu'})$ is linearly independent

► **Lemma 32.** \square *Let $\mu' < \mu$ and $p \in P$ whose entries are all $< \mu'$. Then*

$$\pi_{\mu'}^* \left(\text{ev}_{S_{\mu'}}(\{q \in P \mid q < p\}) \right) = \text{ev}_{S_\mu}(\{q \in P \mid q < p\}).$$

Proof. If $q < p$, then every element of q is also $< \mu'$. Thus, by lemma 21,

$$\pi_{\mu'}^* \left(\text{ev}_{S_{\mu'}}(q) \right) = \text{ev}_{S_\mu}(q),$$

as desired. \blacktriangleleft

► **Lemma 33.** 

$$E(S_\mu) = \bigcup_{\mu' < \mu} E(S_{\mu'})$$

Proof. The inclusion from right to left follows from Lemma 23, so we only need to show that if $p \in E(S_\mu)$ then there exists $\mu' < \mu$ such that $p \in E(S_{\mu'})$. Take μ' to be the supremum of the set $\{i + 1 \mid i \in p\}$. Then $\mu' = 0$ if p is empty, and of the form $i + 1$ for an ordinal $i < \mu$ if p is nonempty. In either case, $\mu' < \mu$.

Since every $i \in p$ satisfies $i < \mu' < \mu$, we have

$$\text{ev}_{S_\mu}(p) = \pi_{\mu'}^*(\text{ev}_{S_{\mu'}}(p))$$

and

$$\text{ev}_{S_\mu}(\{q \in P \mid q < p\}) = \pi_{\mu'}^*(\text{ev}_{S_{\mu'}}(\{q \in P \mid q < p\}))$$

so if $\text{ev}_{S_{\mu'}}(p)$ is in the span of

$$\text{ev}_{S_{\mu'}}(\{q \in P \mid q < p\}),$$

then $\text{ev}_{S_\mu}(p)$ is in the span of


$$\text{ev}_{S_\mu}(\{q \in P \mid q < p\}),$$

contradicting the fact that $p \in E(S_\mu)$. ◀

► **Lemma 34.** 

$$\text{ev}_{S_\mu}(E(S_\mu)) = \bigcup_{\mu' < \mu} \pi_{\mu'}^*(\text{ev}_{S_{\mu'}}(E(S_{\mu'})))$$

Proof. This follows from a combination of Lemmas 21 and 33. ◀

The family of subsets in the union in Lemma 34 is directed with respect to the subset relation (this follows from Lemmas 3, 23, and 21). The sets $\text{ev}_{S_{\mu'}}(E(S_{\mu'}))$ are all linearly independent by the inductive hypothesis, and by injectivity of $\pi_{\mu'}^*$, their images under that map are as well. Thus, by Lemma 8, the union is linearly independent, and we are done. 

4.3.2 Successor case

Let μ be an ordinal, S a closed subset such that for all $x \in S$ and $i \in I$, $x_i = 1$ implies $i < \mu + 1$. In other words, $S = S_{\mu+1}$. Suppose $Q(\mu)$ holds. Then in particular $\text{ev}_{S_\mu} : E(S_\mu) \rightarrow C(S_\mu, \mathbb{Z})$ is linearly independent.

To prove the inductive step in the successor case, we construct a closed subset S' of $\prod_{i \in I} \{0, 1\}$ such that for all $x \in S'$, $x_i = 1$ implies $i < \mu$, and a commutative diagram

$$\begin{array}{ccccc} 0 & \longrightarrow & C(S_\mu, \mathbb{Z}) & \xrightarrow{\pi_\mu^*} & C(S, \mathbb{Z}) & \xrightarrow{g} & C(S', \mathbb{Z}) \\ & & \text{ev}_{S_\mu} \uparrow & & \text{ev}_S \uparrow & & \\ & & E(S_\mu) & \longleftarrow & E(S) & \longleftarrow & E'(S) \end{array} \quad (1)$$

where the top row is exact and $E'(S)$ is the subset of $E(S)$ consisting of those p with $\mu \in p$ (note that p necessarily starts with μ). For $p \in P$, we denote by $p^t \in P$ the sequence obtained by removing the first element of p (t stands for *tail*). The linear map g has the property that $g(\text{ev}_S(p)) = \text{ev}_{S'}(p^t)$ and $p^t \in E(S')$. Given such a construction, the successor step in the induction follows from lemma 9.

6:10 Towards Solid Abelian Groups: A Formal Proof of Nöbeling's Theorem

► **Construction 35.**  Let


$$S_0 = \{x \in S \mid x_\mu = 0\},$$

$$S_1 = \{x \in S \mid x_\mu = 1\},$$

and


$$S' = S_0 \cap \pi_\mu(S_1).$$

Then S' satisfies the inductive hypothesis.

► **Construction 36.**  Let $g_0 : S' \rightarrow S$ denote the inclusion map, and let $g_1 : S' \rightarrow S$ denote the map that swaps the μ -th coordinate to 1 (since $S' \subseteq \pi_\mu(S_1)$, this map lands in S). These maps are both continuous, and we obtain a linear map

$$g_1^* - g_0^* : C(S, \mathbb{Z}) \rightarrow C(S', \mathbb{Z}),$$

which we denote by g .

► **Lemma 37.**  The top row in diagram (1) is exact.

Proof. We already know that π_μ^* is injective. Also, since $\pi_\mu \circ g_1 = \pi_\mu \circ g_0$, we have

$$g \circ \pi_\mu^* = 0.$$

Now suppose we have

$$f \in C(S, \mathbb{Z}) \text{ with } g(f) = 0.$$

We want to find an

$$f_\mu \in C(S_\mu, \mathbb{Z}) \text{ with } f_\mu \circ \pi_\mu = f.$$

Denote by

$$\pi'_\mu : \pi_\mu(S_1) \rightarrow S_1$$


the map that swaps the μ -th coordinate to 1. Since $g(f) = 0$, we have

$$f \circ g_1 = f \circ g_0$$

and hence the two continuous maps $f|_{S_0}$ and $f|_{S_1} \circ \pi'_\mu$ agree on the intersection


$$S' = S_0 \cap \pi_\mu(S_1)$$

Together, they define the desired continuous map f_μ on all of $S_0 \cup \pi_\mu(S_1) = S_\mu$. ◀

► **Lemma 38.**  If $p \in P$ starts with μ , then $g(\text{ev}_S(p)) = \text{ev}_{S'}(p^t)$.

Proof. This follows from considering all the cases given by Lemma 20. We omit the proof here and refer to the Lean proof linked above. ◀

► **Remark 39.** If $p \in E(S)$ and $\mu \in p$, then p satisfies the hypotheses of Lemma 38.

► **Lemma 40.**  If $p \in E(S)$ and $\mu \in p$, then $p^t \in E(S')$.

Proof. Contraposing the statement, it suffices to show that if

$$\text{ev}_{S'}(p^t) \in \text{Span}(\text{ev}_{S'}(\{q \mid q < p^t\})),$$

then

$$\text{ev}_S(p) \in \text{Span}(\text{ev}_S(\{q \mid q < p\})).$$


Given a $q \in P$ such that $i \in q$ implies $i < \mu$, we denote by $q^\mu \in P$ the sequence obtained by adding μ at the front. Write

$$g(\text{ev}_S(p)) = \text{ev}_{S'}(p^t) = \sum_{q < p^t} n_q \text{ev}_{S'}(q) = \sum_{q < p^t} n_q g(\text{ev}_S(q^\mu)).$$

Then by Lemma 37, there exists an $n \in C(S_\mu, \mathbb{Z})$ such that

$$\text{ev}_S(p) = \pi_\mu^*(n) + \sum_{q < p^t} n_q (\text{ev}_S(q^\mu)).$$

Now it suffices to show that each of the two terms in the sum above is in the span of $\{\text{ev}_S(q) \mid q < p\}$. The latter term is because $q < p^t$ implies $q^\mu < p$. The former term is because we can write n as a linear combination indexed by $E(S_\mu)$, and for $q \in E(S_\mu)$ we have $\pi_\mu^*(\text{ev}_{S_\mu}(q)) = \text{ev}_S(q)$ and $\mu \notin q$ so $q < p$. ◀

► **Lemma 41.**  *The set $E(S)$ is the disjoint union of $E(S_\mu)$ and*

$$E'(S) = \{p \in E(S) \mid \mu \in p\}.$$

Proof. We already know by Lemma 23 that $E(S_\mu) \subseteq E(S)$. Also, as noted in Remark 22, if $p \in E(S_\mu)$ then all elements of p are $< \mu$ and hence $p \notin E'(S)$. Now it suffices to show that if $p \in E(S) \setminus E'(S)$, then $p \in E(S_\mu)$.

Since $p \in E(S) \setminus E'(S)$, every $i \in p$ satisfies $i < \mu$. We have

$$\text{ev}_S(p) = \pi_\mu^*(\text{ev}_{S_\mu}(p))$$

and

$$\text{ev}_S(\{q \in P \mid q < p\}) = \pi_\mu^*(\text{ev}_{S_\mu}(\{q \in P \mid q < p\}))$$


so if $\text{ev}_{S_\mu}(p)$ is in the span of

$$\text{ev}_{S_\mu}(\{q \in P \mid q < p\}),$$

then $\text{ev}_S(p)$ is in the span of

$$\text{ev}_S(\{q \in P \mid q < p\}),$$

contradicting the fact that $p \in E(S)$. ◀

The above lemmas prove all the claims made at the beginning of this section, concluding the inductive proof. 

5 The formalisation

First a note on terminology: in the mathematical exposition of the proof in section 4, we have talked about continuous maps from S to \mathbb{Z} . Since \mathbb{Z} is discrete, these are the same as the locally constant maps. The statement we have formalised is Listing 1.

```
instance LocallyConstant.freeOfProfinite (S : Profinite.{u}) :
  Module.Free ℤ (LocallyConstant S ℤ)
```

Listing 1 Nöbeling’s theorem

which says that the \mathbb{Z} -module of locally constant maps from S to \mathbb{Z} is free. When talking about locally constant maps, one does not have to specify a topology on the target, which is slightly more convenient when working in a proof assistant.

The actual proof is about closed subsets of the product $\prod_{i \in I} \{0, 1\}$, which is of course the same thing as the space of functions $I \rightarrow \{0, 1\}$. We implement it as the type $\mathbb{I} \rightarrow \text{Bool}$, where Bool is the type with two elements called `true` and `false`. This is the canonical choice for a two-element discrete topological space in `Mathlib`.

5.1 The implementation of P and $E(S)$

We implemented the set P as the type `Products I` defined as

```
def Products (I : Type*) [LinearOrder I] := {l : List I // l.Chain' (>.)}
```

The predicate `l.Chain' (>.)` means that adjacent elements of the list `l` are related by “>”. We define the evaluation ev_S of products as

```
def Products.eval (S : Set (I → Bool)) (l : Products I) :
  LocallyConstant S ℤ := (l.val.map (e S)).prod
```

where `l.val.map (e S)` is the list of $e_{S,i}$ for i in the list `l.val`, and `List.prod` is the product of the elements of a list.

We define a predicate on `Products`

```
def Products.isGood (S : Set (I → Bool)) (l : Products I) : Prop :=
  l.eval S ∉ Submodule.span ℤ ((Products.eval S) '' {m | m < 1})
```

and then the set $E(S)$ becomes

```
def GoodProducts (S : Set (I → Bool)) : Set (Products I) :=
  {l : Products I | l.isGood S}
```

It is slightly painful to prove completely trivial lemmas like 20 and its corollary 21 in Lean. Indeed, these results are not mentioned in the proof of [16, Theorem 5.4]. Although trivial, they are used often in the proof of the theorem and hence very important to making the proof work. Reading an informal proof of this theorem, one might never realise that these trivialities are used. This is an example of a useful by-product of formalisation; more clarity of exposition.

5.2 Ordinal induction

When formalising an inductive proof of any kind, one has to be very precise about what statement one wants to prove by induction. This is almost never the case in traditional mathematics texts. For example, the proof of [16, Theorem 5.4] claims to be proving by

induction that $E(S)$ is a basis of $C(S, \mathbb{Z})$, not just that it is linearly independent. Furthermore, the set I is not fixed throughout the inductive proof which makes it somewhat unclear what the inductive hypothesis actually says. Working inside the topological space $\prod_{i \in I} \{0, 1\}$ for a fixed set I throughout the proof was convenient in the successor step. This avoided problems that are solved by abuse of notation in informal texts, such as regarding a set as the same thing as its image under a continuous embedding.

The statement of the induction principle for ordinals in Mathlib is the following⁴:

```
def Ordinal.limitRecOn {Q : Ordinal → Sort _} (o : Ordinal)
  (H1 : Q 0)
  (H2 : ∀ o, Q o → Q (succ o))
  (H3 : ∀ o, IsLimit o → (∀ o' < o, Q o') → Q o) :
  Q o
```

In our setting, given a map $Q : \text{Ordinal} \rightarrow \text{Prop}$ (in other words, a *predicate on ordinals*)⁵, we can prove $Q(\mu)$ for any ordinal μ if three things hold:

- The zero case: $Q(0)$ holds.
- The successor case: for all ordinals λ , $Q(\lambda)$ implies $Q(\lambda + 1)$.
- The limit case: for every limit ordinal λ , if $Q(\lambda')$ holds for every $\lambda' < \lambda$, then $Q(\lambda)$ holds.

Finding the correct predicate Q on ordinals was essential to the success of this project:

```
def Q (I : Type*) [LinearOrder I] [IsWellOrder I (<·)] (o : Ordinal) : Prop :=
  o ≤ Ordinal.type (<· : I → I → Prop) →
  (∀ (S : Set (I → Bool)), IsClosed S → contained S o →
    LinearIndependent ℤ (GoodProducts.eval S))
```

The inequality

```
o ≤ Ordinal.type (<· : I → I → Prop)
```

means that $o \leq I$ when I is considered as an ordinal, and the proposition `contained S o` is defined as

```
def contained {I : Type*} [LinearOrder I] [IsWellOrder I (<·)]
  (S : Set (I → Bool)) (o : Ordinal) : Prop :=
  ∀ f, f ∈ S → ∀ (i : I), f i = true → ord I i < o
```

and `ord I i` is an abbreviation for

```
Ordinal.typein (<· : I → I → Prop) i
```

i.e. the element $i \in I$ considered as an ordinal. The conclusion

```
LinearIndependent ℤ (GoodProducts.eval S)
```

means that the map $ev_S : E(S) \rightarrow C(S, \mathbb{Z})$ is linearly independent.

As is often the case, this is quite an involved statement that we are proving by induction, and when writing informally, mathematicians wouldn't bother to specify the map $Q : \text{Ordinal} \rightarrow \text{Prop}$ explicitly.

⁴ We have altered the notation slightly to match the notation in this paper.

⁵ `Prop` is `Sort 0`

5.3 Piecewise defined locally constant maps

In the proof of Lemma 37, we defined a locally constant map $S_\mu \rightarrow \mathbb{Z}$ by giving locally constant maps from S_0 and $\pi_\mu(S_1)$ that agreed on the intersection, and noting that this gives a locally constant map from the union which is equal to S_μ . To do this in Lean, the following definition was added to Mathlib [↗](#):

```
def LocallyConstant.piecewise {X Z : Type*} [TopologicalSpace X] {C₁ C₂ : Set X}
  (h₁ : IsClosed C₁) (h₂ : IsClosed C₂) (h : C₁ ∪ C₂ = Set.univ)
  (f : LocallyConstant C₁ Z) (g : LocallyConstant C₂ Z)
  (hfg : ∀ (x : X) (hx : x ∈ C₁ ∩ C₂), f ⟨x, hx.1⟩ = g ⟨x, hx.2⟩)
  [∀ j, Decidable (j ∈ C₁)] : LocallyConstant X Z where
toFun i := if hi : i ∈ C₁ then f ⟨i, hi⟩
  else g ⟨i, (compl_subset_iff_union.mpr h) hi⟩
isLocallyConstant := omitted
```

It says that given locally constant maps f and g defined respectively on closed subsets C_1 and C_2 which together cover the space X , such that f and g agree on $C_1 \cap C_2$, we get a locally constant map defined on all of X . This seems like exactly what we need in the above-mentioned proof. However, there is a subtlety, in that because of how the rest of the inductive proof is structured, we want the sets S_0 , $\pi_\mu(S_1)$ and S_μ all to be considered as subsets of the underlying topological space $\prod_{i \in I} \{0, 1\}$. To use `LocallyConstant.piecewise`, we would have to consider S_μ as the underlying topological space and S_0 and $\pi_\mu(S_1)$ as subsets of it. This is possible and is what was done initially, but a cleaner solution is to define a variant of `LocallyConstant.piecewise`:

```
def LocallyConstant.piecewise' {X Z : Type*} [TopologicalSpace X]
  {C₀ C₁ C₂ : Set X}
  (h₀ : C₀ ⊆ C₁ ∪ C₂) (h₁ : IsClosed C₁) (h₂ : IsClosed C₂)
  (f₁ : LocallyConstant C₁ Z) (f₂ : LocallyConstant C₂ Z)
  [DecidablePred (· ∈ C₁)]
  (hf : ∀ x (hx : x ∈ C₁ ∩ C₂), f₁ ⟨x, hx.1⟩ = f₂ ⟨x, hx.2⟩) :
  LocallyConstant C₀ Z
```

which satisfies the equations

```
lemma LocallyConstant.piecewise'_apply_left {X Z : Type*} [TopologicalSpace X]
  {C₀ C₁ C₂ : Set X} (h₀ : C₀ ⊆ C₁ ∪ C₂)
  (h₁ : IsClosed C₁) (h₂ : IsClosed C₂)
  (f₁ : LocallyConstant C₁ Z) (f₂ : LocallyConstant C₂ Z)
  [DecidablePred (· ∈ C₁)]
  (hf : ∀ x (hx : x ∈ C₁ ∩ C₂), f₁ ⟨x, hx.1⟩ = f₂ ⟨x, hx.2⟩)
  (x : C₀) (hx : x.val ∈ C₁) :
  piecewise' h₀ h₁ h₂ f₁ f₂ hf x = f₁ ⟨x.val, hx⟩
```

and

```
lemma LocallyConstant.piecewise'_apply_right {X Z : Type*} [TopologicalSpace X]
  {C₀ C₁ C₂ : Set X} (h₀ : C₀ ⊆ C₁ ∪ C₂)
  (h₁ : IsClosed C₁) (h₂ : IsClosed C₂)
  (f₁ : LocallyConstant C₁ Z) (f₂ : LocallyConstant C₂ Z)
  [DecidablePred (· ∈ C₁)]
  (hf : ∀ x (hx : x ∈ C₁ ∩ C₂), f₁ ⟨x, hx.1⟩ = f₂ ⟨x, hx.2⟩)
  (x : C₀) (hx : x.val ∈ C₂) :
  piecewise' h₀ h₁ h₂ f₁ f₂ hf x = f₂ ⟨x.val, hx⟩
```


Here C_0, C_1 , and C_2 are subsets of the same underlying topological space X ; C_1 and C_2 are closed sets covering C_0 , and f_1 and f_2 are locally constant maps defined on C_1 and C_2 respectively, such that f_1 and f_2 agree on the intersection. This fits the application perfectly and shortened the proof of Lemma 37 considerably. Subtleties like this come up frequently, and can stall the formalisation process, especially when formalising general topology. When formalising Gleason’s theorem [↗](#) (another result in general topology relevant to condensed mathematics, see [16, Definition 2.4] and [8]), similar subtleties arose about changing the “underlying topological space” to a subset of the previous underlying topological space.

The phenomenon that it is sometimes more convenient to formalise the definition of an object rather as a subobject of some bigger object is, of course, well known. It was noted in the context of group theory by Gonthier et al. during the formalisation of the odd order theorem, see [9, Section 3.3].

5.4 Reflections on the proof

The informal proof in [16] is about half a page; 21 lines of text. Depending on how one counts (i.e. what parts of the code count as part of the proof and not just prerequisites), the formalised proof is somewhere between 1500 and 3000 lines of Lean code. A big part of the difference is because of omissions in the proof in [16].

A more fair comparison would be with the entirety of section 4 in this paper, which is an account of all the mathematical contents of the formalised proof. Still, there is quite a big difference, which is mostly explained by the pedantry of proof assistants, as discussed in subsections 5.1, 5.2, and 5.3.

5.5 Mathlib integration

As discussed in recent papers by Nash [13] and Best et al. [4], when formalising mathematics in Lean, it is desirable to develop as much as possible directly against Mathlib. Otherwise, the code risks going stale and unusable, while if integrated into Mathlib it becomes part of a library that is continuously maintained.

The development of this project took place on a branch of Mathlib, all code being written in new files. This was a good workflow to get the formalisation done as quickly as possible, because if new code is put in the “correct places” immediately, one has to rebuild part of Mathlib to be able to use that code in other places, which can be a slow process if changes are made deep in the import hierarchy.

The proof of Nöbeling’s theorem described in this paper has now been fully integrated into Mathlib. An unusually large portion of the code was of no independent interest, which resulted in a pull request adding one huge file, which Johan Commelin and Kevin Buzzard kindly reviewed in great detail, improving both the style and performance of the code.

6 Towards condensed mathematics in Mathlib

The history of condensed mathematics in Lean started with the *Liquid Tensor Experiment* (LTE) [6, 17]. This is an example of a formalisation project that was in some sense too big to be integrated into Mathlib. Nevertheless, it was a big success in that it demonstrated the capabilities of Lean and its community by fully formalising the complicated proof of a highly nontrivial theorem about so-called liquid modules. Moreover, it provided a setting in which to experiment with condensed mathematics and find the best way to do homological algebra in Lean. As mentioned above, the goal of LTE was to formalise one specialised theorem.

This is somewhat orthogonal to the goal of Mathlib which is to build a coherent, unified library of formalised mathematics. It is thus understandable that the contributors of LTE chose to focus on completing the task at hand instead of spending time on moving some parts of the code to Mathlib. Now that both LTE and the port of Mathlib to Lean 4 have been completed, we are seeing some important parts of LTE being integrated into Mathlib.

The definition [↗](#) of a condensed object was recently added to Mathlib. During a masterclass on formalisation of condensed mathematics organised in Copenhagen in June 2023, participants collaborated, under the guidance of Kevin Buzzard and Adam Topaz, on formalising as much condensed mathematics as possible in one week (all development took place in Lean 4 and the goal was to write material for Mathlib). The code can be found in the masterclass GitHub repository [↗](#) and much of it has already made it into Mathlib.

Profinite spaces form a rich category of topological spaces and there is more work other than Nöbeling’s theorem to be done in Mathlib. Being the building blocks of condensed sets, it is important to develop a good API for profinite spaces in Mathlib. There, profinite spaces are defined as totally disconnected compact Hausdorff spaces. It is proved [↗](#) that every profinite space can be expressed as a cofiltered limit (more precisely, over the poset of its discrete quotients). It is also proved [↗](#) that the category of profinite spaces has all limits and that the forgetful functor to topological spaces preserves them [↗](#). From this we can extract the following useful theorem:

► **Theorem 42.** *A topological space is profinite if and only if it can be written as a cofiltered limit of finite discrete spaces.*

The story about profinite spaces as limits does not end there, though. Sometimes it is not enough to know just that some profinite space *can* be written as *a* limit, but rather that there is a specific limit formula for it. Lemma 27 gives one specific way of writing a compact subset of a product as a cofiltered limit, which can be useful. Another example can be extracted from [1]. This is the fact that the identity functor on the category of profinite spaces is right Kan extended from the inclusion functor from finite sets to profinite spaces along itself. This gives another limit formula for profinite spaces, coming from the limit formula for right Kan extensions, and is useful when formalising the definition of solid abelian groups [2].

It can also be useful to regard the category of profinite spaces as the pro-category of the category of finite sets. The definition of pro-categories and this equivalence of categories would make for a nice formalisation project and be a welcome contribution to Mathlib.

7 Conclusion and future work

By formalising Nöbeling’s theorem, we have illustrated that the induction principle for ordinals in Mathlib can be used to prove nontrivial theorems outside the theory ordinals themselves. Another contribution is the detailed proof given in section 4, and of course as mentioned before, it is an important step for the formalisation of condensed mathematics to continue.

A natural next step in the formalisation of the theory of solid abelian groups is to port the code in [1, 2] to Lean 4 and get it into Mathlib. Then one can put together the discreteness characterisation and Nöbeling’s theorem to prove the structural results about $\mathbb{Z}[S]^\blacksquare$, which would lead us one step closer to an example of a nontrivial solid abelian group in Mathlib.

More broadly, it is important to continue moving as much as possible of the existing Lean code about condensed mathematics (from the LTE and the Copenhagen masterclass) into Mathlib.

References

- 1 Dagur Asgeirsson. Formalising discrete condensed sets. <https://github.com/dagurtomas/lean-solid/tree/discrete>, 2023.
- 2 Dagur Asgeirsson. Formalising solid abelian groups. <https://github.com/dagurtomas/lean-solid/>, 2023.
- 3 Clark Barwick and Peter Haine. Pyknotic objects, i. basic notions, 2019. [arXiv:1904.09966](https://arxiv.org/abs/1904.09966).
- 4 Alex J. Best, Christopher Birkbeck, Riccardo Brasca, and Eric Rodriguez Boidi. Fermat’s Last Theorem for Regular Primes. In Adam Naumowicz and René Thiemann, editors, *14th International Conference on Interactive Theorem Proving (ITP 2023)*, volume 268 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 36:1–36:8, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. [doi:10.4230/LIPIcs.ITP.2023.36](https://doi.org/10.4230/LIPIcs.ITP.2023.36).
- 5 Dustin Clausen and Peter Scholze. Condensed mathematics and complex geometry. <https://people.mpim-bonn.mpg.de/scholze/Complex.pdf>, 2022.
- 6 Johan Commelin, Adam Topaz et al. Liquid tensor experiment. <https://github.com/leanprover-community/lean-liquid>, 2022.
- 7 László Fuchs. *Infinite Abelian Groups*. ISSN. Elsevier Science, 1970.
- 8 Andrew M. Gleason. Projective topological spaces. *Illinois Journal of Mathematics*, 2(4A):482–489, 1958. [doi:10.1215/ijm/1255454110](https://doi.org/10.1215/ijm/1255454110).
- 9 Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O’Connor, Sidi Ould Biha, Ioana Pasca, Laurence Rideau, Alexey Solovyev, Enrico Tassi, and Laurent Théry. A machine-checked proof of the odd order theorem. In Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie, editors, *Interactive Theorem Proving*, pages 163–179, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.
- 10 The mathlib Community. The lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020*, pages 367–381, New York, NY, USA, 2020. Association for Computing Machinery. [doi:10.1145/3372885.3373824](https://doi.org/10.1145/3372885.3373824).
- 11 Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The lean theorem prover (system description). In Amy P. Felty and Aart Middeldorp, editors, *Automated Deduction - CADE-25*, pages 378–388, Cham, 2015. Springer International Publishing.
- 12 Leonardo de Moura and Sebastian Ullrich. The lean 4 theorem prover and programming language. In André Platzer and Geoff Sutcliffe, editors, *Automated Deduction – CADE 28*, pages 625–635, Cham, 2021. Springer International Publishing.
- 13 Oliver Nash. A Formalisation of Gallagher’s Ergodic Theorem. In Adam Naumowicz and René Thiemann, editors, *14th International Conference on Interactive Theorem Proving (ITP 2023)*, volume 268 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 23:1–23:16, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. [doi:10.4230/LIPIcs.ITP.2023.23](https://doi.org/10.4230/LIPIcs.ITP.2023.23).
- 14 Georg Nöbeling. Verallgemeinerung eines Satzes von Herrn E. Specker. *Invent. Math.*, 6:41–55, 1968. [doi:10.1007/BF01389832](https://doi.org/10.1007/BF01389832).
- 15 Peter Scholze. Lectures on analytic geometry. <http://www.math.uni-bonn.de/people/scholze/Analytic.pdf>, 2019.
- 16 Peter Scholze. Lectures on condensed mathematics. <https://www.math.uni-bonn.de/people/scholze/Condensed.pdf>, 2019.
- 17 Peter Scholze. Liquid tensor experiment. *Experimental Mathematics*, 31(2):349–354, 2022. [doi:10.1080/10586458.2021.1926016](https://doi.org/10.1080/10586458.2021.1926016).

An Operational Semantics in Isabelle/HOL-CSP

Benoît Ballenghien  

Laboratoire Méthodes Formelles, University Paris-Saclay, France

Burkhart Wolff¹  

Laboratoire Méthodes Formelles, University Paris-Saclay, France

Abstract

The theory of Communicating Sequential Processes going back to Hoare and Roscoe is still today a reference model for concurrency. In the fairly rich literature, several versions of operational semantics have been discussed, which should be consistent with the denotational one.

This work is based on Isabelle/HOL-CSP 2.0, a shallow embedding of the failure-divergence model of denotational semantics proposed by Hoare, Roscoe and Brookes in the eighties. In several ways, HOL-CSP is actually an extension of the original setting in the sense that it admits higher-order processes and infinite alphabets.

In this paper, we present a construction and formal equivalence proofs between operational CSP semantics and the underlying denotational failure-divergence semantics. The construction is based on a *definition* of the operational transition operator $P \rightsquigarrow_e P'$ basically via the *After* operator and the classical failure-divergence refinement.

Several choices are discussed to formally derive the operational semantics leading to subtle differences. The derived operational semantics for symbolic Labelled Transition Systems (LTSs) can be potentially used for certifications of model-checker logs as well as combined proof techniques.

2012 ACM Subject Classification Theory of computation → Higher order logic; Theory of computation → Semantics and reasoning; General and reference → Verification

Keywords and phrases Process-Algebra, Semantics, Concurrency, Computational Models, Theorem Proving, Isabelle/HOL

Digital Object Identifier 10.4230/LIPIcs.ITP.2024.7

Supplementary Material *Software (Source Code)*: <https://gitlab.lisn.upsaclay.fr/burkhart.wolff/hol-csp2.0> archived at [swh:1:dir:89601b34c12af4812a99c58054a091d5afbc3e42](https://sw.h1.dir.89601b34c12af4812a99c58054a091d5afbc3e42)

Funding *Benoît Ballenghien*: Labex DigiCosme.

Acknowledgements We would like to thank Catherine Dubois for her remarks on earliest versions of this paper. The present version has been generated and deeply checked with Isabelle/DOF [9, 8].

1 Introduction

Communicating Sequential Processes (CSP) is a language to specify and verify patterns of interaction of concurrent systems. Together with CCS and LOTOS, it belongs to the family of *process algebras*. CSP's rich theory comprises denotational, operational and algebraic semantic facets and has influenced programming languages such as Limbo, Crystal, Clojure and most notably Golang [14]. CSP has been applied in industry as a tool for specifying and verifying the concurrent aspects of hardware systems, such as the T9000 transputer [5].

The theory of CSP was first described in 1978 by Tony Hoare [15], but has since evolved substantially [6, 7, 27]. The denotational semantics of CSP is described by a fully abstract model of behaviour designed to be *compositional*: a process P encompasses all possible behaviours, i. e. sets of *traces* annotated by additional information that allow to reason over

¹ corresponding author



- deadlocks (the resulting semantic domain is called *failure semantics F*)
- and additionally livelocks (the *failure-divergence semantics FD*).

Several attempts have been undertaken to formalize this fairly complex theory, notably [10, 34, 19, 23, 16]. The arguably (see Section 7) most comprehensive one is HOL-CSP [34, 33, 31, 3, 4], which is in several ways not only a formalization in a proof assistant, but a generalization of the original setting:

- the set of traces $'\alpha$ *trace* is constructed over an arbitrary type $'\alpha$ in HOL, paving the way for dense time, vector spaces, etc.²,
- more generally speaking, HOL-CSP attempts to remove finiteness-restrictions, and
- the semantic domain is encapsulated in the type $'\alpha$ *process* belonging to the cpo type class (see Subsection 2.4). Thus, process patterns can be expressed and analyzed.

In this paper, we present the formal theory of the operational semantics of HOL-CSP which is consistent with the denotational one by construction. To this end, we proceed by defining the operational transition operator $P \rightsquigarrow_e P'$ in terms of the *After* operator and the classical failure-divergence refinement. In the literature on process algebras like CSP ([15, 6, 7, 27]) or Circus ([35, 36]), several versions of operational semantics have been presented, but a formal proof of equivalence for a substantial part of the language has never been undertaken. Our proof architecture foresees an Isabelle locale (a parameterized theory) whose instances represent several of these versions, thus shedding some light on their relationships. The operational rules for small-step and big-steps pave the way for symbolic execution of processes and the combination of model-checking with theorem proving.

We proceed as follows: after an introduction to “classic” CSP and our extensions HOL-CSP and HOL-CSPM as an Isabelle framework in Section 2, we present in Section 3 the core construction of this paper, the *bridge*-definition linking the denotational semantics to operational one(s), resulting in the formally proven laws written in Section 4. This is generalized to big steps semantics in Section 5 and possible variants are discussed in Section 6. Note that HOL-CSP[33], HOL-CSPM[3] as well as the novel contribution HOL-CSP_OpSem [4] containing the proofs discussed here are published in the Archive of Formal Proofs AFP.

2 Background

2.1 Classic CSP Syntax

At a glance, the syntax of the classical CSP core language reads as follows:

$$P ::= SKIP \mid STOP \mid P \square P' \mid P \sqcap P' \mid P \llbracket A \rrbracket P' \mid P ; P' \mid P \setminus A \\ \mid a \rightarrow P \mid \square a \in A \rightarrow P(a) \mid \sqcap a \in A \rightarrow P(a) \mid \text{Renaming } P \ g \mid \mu X. f(X)$$

SKIP signals termination, *STOP* denotes a deadlock. Two choice operators are distinguished:

1. the *external choice* \square , which forces a process “to follow” whatever its context requires,
2. the *internal choice* \sqcap , which imposes on the context of a process “to follow” the non-deterministic choices made.

With the prefix operator $a \rightarrow P$ which signals a and continues with P (where a is an element of a set Σ of *events*), generalized choices of the form $\square a \in A \rightarrow P(a)$ resp. $\sqcap a \in A \rightarrow P(a)$ are constructed (A is originally a finite set). When events are tagged with *channels*, i. e. $\Sigma = CHANNELS \times DATA$, syntactic sugar like $c?x \in A \rightarrow P(x)$ or $c!x \in A \rightarrow P(x)$ is added; the former reads intuitively as “ x is read from channel c ” while the latter means “ x is arbitrarily chosen from A and sent into c ” (where $c \in CHANNELS$ and $x \in DATA$).

² or even differential equations as in cyber-physical system models [12]

The sequential composition $P ; P'$ behaves first like P and, once it has successfully terminated, like P' . $P \setminus A$ consists in hiding the events of the set A . *Renaming* $P g$ results in a process in which each event e of P is renamed in $g(e)$. The *Sliding* operator $P \triangleright P'$ is defined as $(P \square P') \square P'$. The fixed point $\mu X. f(X)$ operator satisfies $(\mu X. f(X)) = f(\mu X. f(X))$ (but requires precautions, see Subsection 2.4).

CSP describes all communication with one single primitive: the synchronized product written $P \llbracket A \rrbracket P'$. Note that interleaving $P \parallel P'$ stands for $P \llbracket \{\} \rrbracket P'$, whereas the parallel operator $P \parallel P'$ is a shortcut for $P \llbracket UNIV \rrbracket P'$ ($UNIV$ is the universal set).

2.2 Classic CSP Semantics

The denotational semantics (following [27]) comes in three layers: the *trace model*, the (*stable*) *failures model* and the *failure-divergence model*.

In the trace semantics model, the behaviour of a process P is denoted by a prefix-closed set of traces, denoted $\mathcal{T} P$, similar to the well-known concept of a “language of an automata”. Since traces are finite lists and infinite behaviour is therefore represented via the set of approximations, an additional element *tick* (written \checkmark) is used to represent explicit termination signaled by *SKIP*. Note that, obviously, *tick* should only appear at the end of a trace (traces should be *front-tickFree*).

It is impossible to distinguish external and internal non-determinism in the trace model since the traces of both operators are just the union of their argument traces. To be more discriminant, [6] proposed the failure semantics model, where traces were annotated with a set of *refusals*, i. e. sets of events a process can *not* engage in. This leads to the notion of a *failure* $(t, X) \in \mathcal{F} P$ which is a pair of a trace t and a set of refusals X . Consider for example the process $P = (a \rightarrow SKIP) \square (a \rightarrow STOP)$. The traces $\mathcal{T} P$ will non-deterministically lead to a situation where the process accepts termination (but refuses everything else) or just refuses everything. So, if we assume $\Sigma = \{a, \checkmark\}$, then the traces $\mathcal{T} P$ will be $\{\[], [a]\}$. The failures $\mathcal{F} P$ are then $\{(\[], \{\{\checkmark\}\}), ([a], \{\Sigma, \{a\}\})\}$ (plus all subsets of the respective refusal sets, which is required for the recursion ordering discussed in Subsection 2.4).

Finally, [6] enriched the semantic domain of CSP with one more element, the set of *divergences* (written $\mathcal{D} P$), in order to distinguish deadlocks from livelocks³. In the failure divergence model, the semantic domain consists of a pair of failures and divergences, where the latter are traces to situations where livelocks may occur.

In contrast to Hoare Logics and its Hoare Triples, which is a framework to reason over terminating calculations, CSP and process refinement are designed to reason over non-terminating calculations. Three classic refinement notions are considered:

1. the trace refinement: $P \sqsubseteq_T Q \equiv \mathcal{T} P \supseteq \mathcal{T} Q$,
2. the failure refinement: $P \sqsubseteq_F Q \equiv \mathcal{F} P \supseteq \mathcal{F} Q$, and
3. the failure-divergence refinement $P \sqsubseteq_{FD} Q \equiv \mathcal{F} P \supseteq \mathcal{F} Q \wedge \mathcal{D} P \supseteq \mathcal{D} Q$.

It turns out that beyond common protocol refinement proofs and test-problems, many properties such as deadlock or livelock freeness can be expressed via a refinement statement.

2.3 Theories and Locales in Isabelle and HOL

Isabelle is a major interactive proof assistant implementing higher-order logic (HOL). As an LCF style theorem prover, it is based on a small logical core (kernel) to increase the trustworthiness of proofs without requiring – yet supporting – explicit proof objects.

³ also called *infinite internal chatter* as occurring in processes like $\mu x. a \rightarrow x \setminus \{a\}$

The Isabelle distribution comes with a number of library theories constructed solely from definitional axioms; among them basic data-types for sets, lists, arithmetics, a substantial part of analysis, and – particularly relevant here – Scott domain theory (HOLCF) [22] providing a particular type class $'\alpha::pcpo$, i.e. the class of types $'\alpha$ for which a least element \perp and a complete partial order \sqsubseteq is defined.

HOLCF provides the concept of *continuity*, the concept of *admissibility*, the fixed point operator $\mu x. f x$ as well as the fixed point induction for admissible predicates. Isabelle’s type inference can automatically infer, for example, that if $'\alpha::pcpo$, then $('\beta \Rightarrow '\alpha)::pcpo$.

A distinguishing feature of Isabelle is the **locale** mechanism, i.e. theories that may be parameterized by types, constant-symbols and local hypotheses over them. Since **locales** may inherit from other **locales**, they represent a powerful structuring mechanism for orders and algebraic structures very similar to dependent types available in other systems.

2.4 Isabelle/HOL-CSP

Isabelle/HOL-CSP is a shallow embedding of CSP in HOL based on the traditional semantic domain described by 9 “axioms” over the three semantic functions $\mathcal{T} :: '\alpha \text{ process} \Rightarrow '\alpha \text{ trace set}$, $\mathcal{F} :: '\alpha \text{ process} \Rightarrow '\alpha \text{ failure set}$ and $\mathcal{D} :: '\alpha \text{ process} \Rightarrow '\alpha \text{ trace set}$:

- the empty trace is always the initial trace and any trace is *front-tickFree*;
- traces of a process are *prefix-closed* and a process can refuse all subsets of refusals;
- any event refused by a process after a trace s must be in a refusal set associated to s ;
- the *tick* accepted after a trace s implies that all other events are refused;
- a divergence trace with any suffix is itself a divergence one
- once a process has diverged, it can engage in or refuse any sequence of events.
- a *tick-ending* divergence trace has a *tickFree* divergence trace prefix of maximal length.

More formally, a process P of the type $\Sigma \text{ process}$ should have the following properties:

$$\begin{aligned} & (\square \in \mathcal{T} P \wedge (\forall s X. (s, X) \in \mathcal{F} P \longrightarrow \text{front-tickFree } s) \wedge \\ & (\forall s t. s @ t \in \mathcal{T} P \longrightarrow s \in \mathcal{T} P) \wedge (\forall s X Y. (s, Y) \in \mathcal{F} P \wedge X \subseteq Y \longrightarrow (s, X) \in \mathcal{F} P) \wedge \\ & (\forall s X Y. (s, X) \in \mathcal{F} P \wedge (\forall c. c \in Y \longrightarrow s @ [c] \notin \mathcal{T} P) \longrightarrow (s, X \cup Y) \in \mathcal{F} P) \wedge \\ & (\forall s X. s @ [\checkmark] \in \mathcal{T} P \longrightarrow (s, X - \{\checkmark\}) \in \mathcal{F} P) \wedge \\ & (\forall s t. s \in \mathcal{D} P \wedge \text{tickFree } s \wedge \text{front-tickFree } t \longrightarrow s @ t \in \mathcal{D} P) \wedge \\ & (\forall s X. s \in \mathcal{D} P \longrightarrow (s, X) \in \mathcal{F} P) \wedge (\forall s. s @ [\checkmark] \in \mathcal{D} P \longrightarrow s \in \mathcal{D} P)) \end{aligned}$$

The core of HOL-CSP is to encapsulate this wishlist into a type definition. This is achieved by 1) defining the pair of failures and divergences $\Sigma \text{ process}_0$ via $(\Sigma^\checkmark \text{ list} \times (\Sigma^\checkmark) \text{ set}) \text{ set} \times (\Sigma^\checkmark) \text{ set}$ (where $\Sigma^\checkmark = \Sigma \uplus \{\checkmark\}$), 2) by turning the above wishlist into a data-constraint *is-process* of type $\Sigma \text{ process}_0 \Rightarrow \text{bool}$, and 3) by establishing an isomorphism between the subset of $\Sigma \text{ process}_0$ ’es satisfying *is-process* via the-specification construct:

$$\text{typedef } '\alpha \text{ process} = \{P :: '\alpha \text{ process}_0 . \text{is-process } P\}$$

Subsequently, we provide definitions for each CSP operator in terms of $\Sigma \text{ process}_0$; these definitions formalize directly the textbook [27]. Finally, we prove that each operator preserves the *is-process*-invariant. The preservation even holds for arbitrary (possibly infinite) sets A in the generalisations $\square x \in A \rightarrow P(x)$ resp. $\square x \in A \rightarrow P(x)$. Note that both use higher-order abstract syntax and have the type $'\alpha \text{ set} \Rightarrow (''\alpha \Rightarrow '\alpha \text{ process}) \Rightarrow '\alpha \text{ process}$.

A major problem prevails: how to give semantics to the fixed point operator?

This is achieved by turning the denotational domain of CSP into a Scott complete partial order (cpo) [29], which provides semantics for the fixed point operator $\mu x. f(x)$ under the condition that f is continuous wrt. this partial ordering. Since the natural ordering \sqsubseteq_{FD} -

is too weak for this purpose, Roscoe and Brookes [24] proposed a complete *process ordering* $P \sqsubseteq Q$ which is stronger, i. e. $P \sqsubseteq Q \implies P \sqsubseteq_{FD} Q$, and yet ensures completeness at least for general read and write operations.

It is based on the concept *refusals after* a trace s ($\mathcal{R}_a P s \equiv \{X \mid (s, X) \in \mathcal{F} P\}$):

$$P \sqsubseteq Q \equiv \mathcal{D} Q \subseteq \mathcal{D} P \wedge (\forall s. s \notin \mathcal{D} P \implies \mathcal{R}_a P s = \mathcal{R}_a Q s) \wedge \text{min-elems}(\mathcal{D} P) \subseteq \mathcal{T} Q$$

► **Theorem 1** (Continuity). *Almost all HOL-CSP operators \otimes are continuous wrt. (\sqsubseteq) , i. e.:*

$$\text{cont } f \implies \text{cont } g \implies \text{cont}(\lambda x. (f x) \otimes (g x))$$

► **Theorem 2** (Fixed-point Inductions). *Since (\sqsubseteq_{FD}) is admissible, when f is continuous we have an induction rule of the following form:*

$$C(\perp) \sqsubseteq_{FD} Q \implies (\bigwedge x. C(x) \sqsubseteq_{FD} Q \implies C(fx) \sqsubseteq_{FD} Q) \implies C(\mu X. f X) \sqsubseteq_{FD} Q$$

► **Proposition 3** (CSP-Algebra). *HOL-CSP provides about 200 rules derived from the denotational semantics, be it monotonicities or equalities, which were called the “axioms” in the literature. We show here only the small collection used in the subsequent example proof:*

$$\begin{aligned} (\forall y. c y \in S) &\implies c?x \rightarrow P x \llbracket S \rrbracket c?x \rightarrow Q x = c?x \rightarrow (P x \llbracket S \rrbracket Q x) \\ (\forall y. c y \in S) &\implies \text{inj } c \implies c!a \rightarrow P \llbracket S \rrbracket c?x \rightarrow Q x = c!a \rightarrow (P \llbracket S \rrbracket Q a) \\ d a \notin S &\implies (\bigwedge y. c y \in S) \implies d!a \rightarrow P \llbracket S \rrbracket c?x \rightarrow Q x = d!a \rightarrow (P \llbracket S \rrbracket c?x \rightarrow Q x) \\ d \in S &\implies (\bigwedge y. c y \notin S) \implies d \rightarrow P \llbracket S \rrbracket c?x \rightarrow Q x = c?x \rightarrow (d \rightarrow P \llbracket S \rrbracket Q x) \\ d a \notin C &\implies c \in C \implies c \rightarrow Q \llbracket C \rrbracket d!a \rightarrow P = d!a \rightarrow (c \rightarrow Q \llbracket C \rrbracket P) \\ \forall y. c y \notin B &\implies c?x \rightarrow P x \setminus B = c?x \rightarrow (P x \setminus B) \\ \forall y. c y \notin B &\implies c!a \rightarrow P \setminus B = c!a \rightarrow (P \setminus B) \\ c a \in B &\implies c!a \rightarrow P \setminus B = P \setminus B \end{aligned} \quad \text{etc.}$$

The theories HOL-CSP and HOL-CSPM [3] also add a number of extensions of the original language. This includes for the binary operators $P \llbracket A \rrbracket P'$, $P ; P'$, $P \square P'$, $P \sqcap P'$, the generalizations $\llbracket S \rrbracket i \in \# M. P(i)$, $\llbracket \llbracket i \in \# M. P(i) \rrbracket$, etc. Roscoe’s operators *Interrupt* $P \triangle P'$ and *Throw* (exception handler) $P \Theta a \in A. P'(a)$ have also been included since they come in handy in some of the more general constructions. Finally, [31] proposed another refinement ordering, the trace-divergence ordering $P \sqsubseteq_{DT} Q \equiv P \sqsubseteq_T Q \wedge \mathcal{D} Q \subseteq \mathcal{D} P$, which has surprisingly benign properties wrt. operational semantics and which is relevant for applications [12].

2.5 A Model and Sample Proof in HOL-CSP

Of course, proving refinements is not done by unfolding the definitions in the denotational semantics. Instead, the predominant proof technique is merely fixed point induction via Theorem 2, application of the algebraic rules of Proposition 3 as well as the monotonicity rules which are a consequence of Theorem 1. We demonstrate this with the paradigmatic *CopyBuffer* example, where we model a protocol *COPY* (“received data on channel *left* will eventually be copied into channel *right*”) and an implementing *SYS* which transfers the data from some *SEND*-component into some *REC*-component using an internal channel *mid* where *REC* acknowledges each data-package via a signal on the internal *ack*-channel.

The formalisation of these model-elements proceeds as follows. The events were defined by the inductive data-type introducing the channels:

$$\text{datatype } 'a \text{ channel} = \text{left } 'a \mid \text{right } 'a \mid \text{mid } 'a \mid \text{ack}$$

Note that this definition leaves open what data is actually transmitted. A synchronisation set SYN is defined via $\{e \mid \exists x. e = mid\ x\} \cup \{ack\}$. The process $COPY$ of type ' α channel process' is defined by $\mu x. left?xa \rightarrow right!xa \rightarrow x$, the process $SEND$ by $\mu x. left?xa \rightarrow mid!xa \rightarrow ack \rightarrow x$ and the process REC by $\mu x. mid?xa \rightarrow right!xa \rightarrow ack \rightarrow x$. The latter two are wired together to the process SYS via $SYS \equiv SEND \llbracket SYN \rrbracket REC \setminus SYN$.

Now we ask the question: *does SYS implement the protocol $COPY$?* This can be rewritten as the following refinement problem : $COPY \sqsubseteq_{FD} SYS$.

Unfolding $COPY$ and applying Theorem 2 yields the two subgoals:

1. $\perp \sqsubseteq_{FD} (SEND \llbracket SYN \rrbracket REC \setminus SYN)$
2. $\bigwedge x. x \sqsubseteq_{FD} (SEND \llbracket SYN \rrbracket REC \setminus SYN) \implies left?a \rightarrow right!a \rightarrow x \sqsubseteq_{FD} (SEND \llbracket SYN \rrbracket REC \setminus SYN)$

where the former is trivial and the latter represents the induction step. If we unfold once $SEND$ and REC and apply the reduction rules of Proposition 3, this results in:

$$left?a \rightarrow right!a \rightarrow x \sqsubseteq_{FD} left?a \rightarrow right!a \rightarrow (SEND \llbracket SYN \rrbracket REC \setminus SYN)$$

Applying the monotonicity rules resulting from Theorem 1 we can reduce this goal to the induction hypothesis $x \sqsubseteq_{FD} (SEND \llbracket SYN \rrbracket REC \setminus SYN)$.

Furthermore this proof can be highly automated (reduces to a few lines in Isabelle/Isar). No assumption is made over ' α ', this construction is therefore truly parametric over data, which is in stark contrast to model-checkers for CSP such as [1, 30]. Using the fact that functions over processes are continuous, we can specify and analyse, e.g., global variables by $VAR\ \sigma_{init} \equiv (\mu x. (\lambda\sigma. (Read!\sigma \rightarrow x\ \sigma) \square (Update?\sigma' \rightarrow x\ \sigma')))\ \sigma_{init}$ and other building blocks of concurrent programs like buffers, semaphores and monitors.

3 Small Steps Semantics

Operational semantics of CSP involve two kinds of transitions that we need to define:

- the τ transition, denoted $P \rightsquigarrow_{\tau} Q$, (*internal transition*)
- and the transition with an observable event (*external transition*) where we distinguish the two cases resulting from the type sum of “real” events $ev\ e$ and the special event \checkmark . The former transition will be denoted by $P \rightsquigarrow_e Q$, the latter by $P \rightsquigarrow_{\checkmark} Q$.

Initially, Hoare and Jifeng in [20] proposed the following link between the operational and denotational semantics: $P \rightsquigarrow_{\tau} Q \equiv P \sqsubseteq_{FD} Q$ and $P \rightsquigarrow_e Q \equiv P \sqsubseteq_{FD} (e \rightarrow Q) \square P$. This approach is fine as long as we do not consider explicit termination of processes via the special event \checkmark . Moreover, the initial presentation was referring to fragment of the language which foundation wrt. the underlying denotational semantics (including process ordering) was still prone to subtle errors [34]. For these reasons, we opted for another bridge based on the *After* operator, denoted $P\ after\ e$, which represents a kind of inversion of $e \rightarrow P$. We will investigate the precise connection between both definitions in Section 6.

3.1 The Notion of initials

As prerequisite, we need the events P can start with, called *initials* P and denoted P^0 .

► **Definition 4** (*initials*). *The definition is straightforward: $P^0 \equiv \{e \mid [e] \in \mathcal{T}\ P\}$ or equivalently, thanks to ' α process properties, $P^0 = \{e \mid \exists s. e \cdot s \in \mathcal{T}\ P\}$.*

Intuitively, for each $ev\ e$ in P^0 , the traces of $P\ after\ e$ should be the tails of the traces of P beginning with $ev\ e$. The question arises what happens with processes P where no trace is beginning with $ev\ e$. The semantic domain of $P\ after\ e$ will require that the set of traces is non-empty (recall the *is-process* invariant from which we built the ' α process' type).

The *initials* are a notion commonly evoked in [6, 27, 28]; for now let us derive the general computational rules for it.

► **Theorem 5** (Basic rules for *initials*). *We derive:*

$$\begin{array}{lll} \perp^0 = UNIV & (P^0 = \emptyset) = (P = STOP) & SKIP^0 = \{\checkmark\} \\ (P \sqcap Q)^0 = P^0 \cup Q^0 & (P \sqcup Q)^0 = P^0 \cup Q^0 & (e \rightarrow P)^0 = \{ev\ e\} \\ (P \triangleright Q)^0 = P^0 \cup Q^0 & (P \triangle Q)^0 = P^0 \cup Q^0 & (P \ominus_{a \in A} Q\ a)^0 = P^0 \end{array}$$

► **Theorem 6** (More complex rules for *initials*). *The following list requires more caution:*

- $(Renaming\ P\ f)^0 = (if\ P = \perp\ then\ UNIV\ else\ EvExt\ f\ \cdot\ P^0)$
- $(P ; Q)^0 = (if\ P = \perp\ then\ UNIV\ else\ P^0 - \{\checkmark\} \cup (if\ \checkmark \in P^0\ then\ Q^0\ else\ \emptyset))$
- $(P \llbracket S \rrbracket Q)^0 = P^0 \cup Q^0 - (\{\checkmark\} \cup ev\ \cdot\ S) \cup P^0 \cap Q^0 \cap (\{\checkmark\} \cup ev\ \cdot\ S)$
if $P \neq \perp$ and $Q \neq \perp$ (otherwise $(\perp \llbracket S \rrbracket Q)^0 = UNIV$ and $(P \llbracket S \rrbracket \perp)^0 = UNIV$).

The equality for the *Hiding* operator, proved but omitted here, is downright difficult. Note that the function *initials* is of type ' α process \Rightarrow ' α event set. This implies that we may have $\checkmark \in P^0$, especially when $P = SKIP$ for which it serves as a refinement characterization.

► **Theorem 7** (Characterization of initial \checkmark). $\checkmark \in P^0$ if and only if $P \sqsubseteq_{FD} SKIP$.

3.2 The After Operator

There is no comprehensive treatment of the *After* operator in the CSP literature, at least not with a formal definition and a precise clarification of the behaviour wrt. the other operators; we had to do a number of trials and second-guessing here. A key element is the notion of *initials* (Definition 4); assuming $ev\ e \in P^0$ for $P::'\alpha$ process, we obviously choose its failures to be $\{(s, X) \mid (ev\ e \cdot s, X) \in \mathcal{F}\ P\}$ and its divergences $\{s \mid ev\ e \cdot s \in \mathcal{D}\ P\}$.

This solves part of the problem, but is not enough. We will need a *total* definition of this operator in HOL, i.e. we need to deal with the case $ev\ e \notin P^0$. There are basically the alternatives *STOP*, \perp , or just some underspecified constant *undefined*. Since the actual choice made leads to subtle differences in corner cases but does not impact the operational rules that we establish, we use Isabelle **locales** mentioned in Subsection 2.3 to model this:

► **Definition 8** (*After* operator).

locale *After* = **fixes** $\Psi :: \langle [\alpha\ process, \alpha] \Rightarrow \alpha\ process \rangle$ **begin**

lift-definition *After* :: $\langle [\alpha\ process, \alpha] \Rightarrow \alpha\ process \rangle$ (**infixl** $\langle after \rangle$ 77)

is $\langle \lambda P\ e.\ if\ ev\ e \in initials\ P$
 $then\ (\{(s, X). (ev\ e \# s, X) \in \mathcal{F}\ P\}, \{s.\ ev\ e \# s \in \mathcal{D}\ P\})$
 $else\ (\mathcal{F}\ (\Psi\ P\ e), \mathcal{D}\ (\Psi\ P\ e)) \rangle$

Here, **lift-definition** is a variant of Isabelle constant **definition** which gives automatic support for “lifting” an operation of the ' α process₀-level to ' α process. Note that Ψ is a parameter of the locale requiring no assumption which can be instantiated freely.

The need of the theory of the *After* operator and its straightforward generalisation to traces denoted by P / s was identified at many places in the CSP literature [15, 27, 28], especially after basing the process-ordering (\sqsubseteq) on its refusals. *After* and P / s play a pivotal role when linking CSP to automata-theoretic concepts; nevertheless it was commonly treated as something “meta”⁴.

⁴ Roscoe states that “this operator should not be thought of as an ordinary part of the CSP language”[27].

In our work, we turn *After* into an ordinary operator, compatible with all other concepts, preserving the invariant of ' α process, enjoying monotony and continuity, and a number of distributivities (that one could call “algebraic laws”) useful for establishing an operational semantics. The formal proofs of this part of our theory amount to 2000 lines. For example, we obtain equalities like the following:

- **Theorem 9** (*After and Sync*). ($P \llbracket S \rrbracket Q$) after e is equal to:
- if $P = \perp \vee Q = \perp$ then \perp
 - else if $ev\ e \in P^0 \cap Q^0$
 - then if $e \in S$ then P after $e \llbracket S \rrbracket Q$ after e
 - else $(P$ after $e \llbracket S \rrbracket Q) \sqcap (P \llbracket S \rrbracket Q$ after $e)$
 - else if $ev\ e \in P^0 \wedge e \notin S$ then P after $e \llbracket S \rrbracket Q$
 - else if $ev\ e \in Q^0 \wedge e \notin S$ then $P \llbracket S \rrbracket Q$ after e else $\Psi (P \llbracket S \rrbracket Q) e$

The only operator for which we have not managed to establish such a property is *Hiding*. However, we have at least the following monotonies:

- **Theorem 10** (*After and Hiding*).

$$\begin{aligned} \llbracket ev\ e \in P^0; e \in B \rrbracket &\Longrightarrow (P \setminus B) \sqsubseteq_{FD} (P \text{ after } e \setminus B) \\ \llbracket ev\ e \in P^0; e \notin B \rrbracket &\Longrightarrow (P \setminus B) \text{ after } e \sqsubseteq_{FD} (P \text{ after } e \setminus B) \end{aligned}$$

This will not be too restrictive for our construction thanks to the following theorem.

- **Theorem 11** (Characterization of FD-refinement).

The FD-refinement $P \sqsubseteq_{FD} Q$ holds if and only if $P = P \sqcap Q$.

3.3 The Rationale for an Operational Semantics

With respect to the τ transition, (\rightsquigarrow_τ), we follow the choice of Jifeng and Hoare in [20], i. e. we define it by $P \rightsquigarrow_\tau Q \equiv P \sqsubseteq_{FD} Q$.

With respect to the external transitions, we expect that:

- $P \rightsquigarrow_e Q$ (resp. $P \rightsquigarrow_\checkmark Q$) is impossible if $ev\ e \notin P^0$ (resp. $\checkmark \notin P^0$)
- event transitions should absorb τ transitions (on both sides) because (\sqsubseteq_{FD}) is transitive
- since $P \sqsubseteq_{FD} Q$ can be interpreted as “ Q is more deterministic than P ”, Q should be at least as deterministic as P after e when P makes a transition via event e .

The formalization still requires an extension of the *After* operator to deal with \checkmark . This is achieved by extending the **locale** with an additional parameter Ω .

- **Definition 12** (*After_{tick} operator*).

definition $After_{tick} :: \langle [\alpha\ process, \alpha\ event] \Rightarrow \alpha\ process \rangle$ (**infixl** $\langle after_\checkmark \rangle$ 77)
where $\langle P \text{ after}_\checkmark e \equiv \text{case } e \text{ of } ev\ x \Rightarrow P \text{ after } x \mid \checkmark \Rightarrow \Omega\ P \rangle$

3.4 Finally: Formal Definitions of the Transition Relations

To make our construction as general as possible, we formalize the requirements of Subsection 3.3 by parameterizing the τ transition in a **locale** with four hypotheses:

locale $OpSemTransitions = AfterExt\ \Psi\ \Omega$
for $\Psi :: \langle [\alpha\ process, \alpha] \Rightarrow \alpha\ process \rangle$ **and** $\Omega :: \langle \alpha\ process \Rightarrow \alpha\ process \rangle +$
fixes $\tau\text{-trans} :: \langle [\alpha\ process, \alpha\ process] \Rightarrow bool \rangle$ (**infixl** $\langle \rightsquigarrow_\tau \rangle$ 50)
assumes $\tau\text{-trans-NdetL}: \langle P \sqcap Q \rightsquigarrow_\tau P \rangle$
and $\tau\text{-trans-transitivity}: \langle P \rightsquigarrow_\tau Q \Longrightarrow Q \rightsquigarrow_\tau R \Longrightarrow P \rightsquigarrow_\tau R \rangle$

and τ -trans-anti-mono-initials: $\langle P \rightsquigarrow_{\tau} Q \implies Q^0 \subseteq P^0 \rangle$
and τ -trans-mono-AfterExt: $\langle Q^0 \implies P \rightsquigarrow_{\tau} Q \implies P \text{ after}_{\checkmark} e \rightsquigarrow_{\tau} Q \text{ after}_{\checkmark} e \rangle$
begin

abbreviation *ev-trans* :: $\langle [\alpha \text{ process}, \alpha, \alpha \text{ process}] \Rightarrow \text{bool} \rangle$ ($\langle _ \rightsquigarrow_{\tau} _ \rangle$ [50, 3, 51] 50)
where $\langle P \rightsquigarrow_e Q \equiv ev \ e \in P^0 \wedge P \text{ after}_{\checkmark} ev \ e \rightsquigarrow_{\tau} Q \rangle$

abbreviation *tick-trans* :: $\langle [\alpha \text{ process}, \alpha \text{ process}] \Rightarrow \text{bool} \rangle$ ($\langle _ \rightsquigarrow_{\checkmark} _ \rangle$ [50, 51] 50)
where $\langle P \rightsquigarrow_{\checkmark} Q \equiv \checkmark \in P^0 \wedge P \text{ after}_{\checkmark} \checkmark \rightsquigarrow_{\tau} Q \rangle$

To sum up, this **locale** needs to be instantiated with:

- a function Ψ that is a placeholder for the value of $P \text{ after } e$ when $ev \ e \notin P^0$
- a function Ω that is a placeholder for the value of $P \text{ after}_{\checkmark} \checkmark$
- a binary relation (\rightsquigarrow_{τ}) which is compatible with (\sqcap), is transitive, makes *initials* anti-monotonic and makes *After_{tick}* monotonic.

With these only four local axioms, we can derive most of the basic operational rules for *SKIP*, $e \rightarrow P$, etc., and some of the rules for $P \setminus S$ or $P ; Q, \dots$ In order to recover the remaining rules, we divide the work. For each operator with a missing rule, we introduce a **locale** inheriting from *OpSemTransitions* in which we add a local axiom (about (\rightsquigarrow_{τ})). With the rules already proven and the denotational properties of the operator, we can derive the missing rules. Here, we illustrate the case of one of the rules for the *Sync* operator:

$$\frac{e \notin S \quad P \rightsquigarrow_e P'}{P \llbracket S \rrbracket Q \rightsquigarrow_e P' \llbracket S \rrbracket Q}$$

Proof. (Derivation of one of the *Sync* Operational Rules).

Case $P = \perp$ or $Q = \perp$, this is obvious because of the properties of \perp .

Otherwise since $P \rightsquigarrow_e P'$ we have $ev \ e \in P^0$.

With Theorem 6 and $e \notin S$, we additionally have $ev \ e \in (P \llbracket S \rrbracket Q)^0$.

Thus, from Theorem 9, $(P \llbracket S \rrbracket Q) \text{ after } e$ is equal to $(P \text{ after } e \llbracket S \rrbracket Q) \sqcap (P \llbracket S \rrbracket Q \text{ after } e)$ if $ev \ e \in Q^0$, and $P \text{ after } e \llbracket S \rrbracket Q$ otherwise.

In both cases, with (\rightsquigarrow_{τ}) properties, we obtain $(P \llbracket S \rrbracket Q) \text{ after } e \rightsquigarrow_{\tau} P \text{ after } e \llbracket S \rrbracket Q$.

Using the additional assumption that, in general, $P \rightsquigarrow_{\tau} P' \implies P \llbracket S \rrbracket Q \rightsquigarrow_{\tau} P' \llbracket S \rrbracket Q$, we finally conclude that $P \llbracket S \rrbracket Q \rightsquigarrow_e P' \llbracket S \rrbracket Q$. ◀

Finally, by assembling the **locales** of each operator (which is inheriting of all these **locales**), their instantiations lead to formal proofs that the core of the ruleset shown in Section 4 are actually derivable for T,F, and FD semantics. In the end, they all rely on the four assumptions of *OpSemTransitions* and on eight additional assumptions of monotony for the first argument wrt. (\rightsquigarrow_{τ}) for the operators *Det*, *Seq*, *Hiding*, *Sync*, *Sliding*, *Interrupt*, *Throw* and *Renaming* e.g. $P \rightsquigarrow_{\tau} P' \implies P \triangleright Q \rightsquigarrow_{\tau} P' \triangleright Q$. Special cases of rules not in the core ruleset will be discussed in the subsequent sections.

4 The Derived Rules of the Operational Semantics at a Glance

$$\frac{P \rightsquigarrow_e P' \quad P' \rightsquigarrow_{\tau} P''}{P \rightsquigarrow_e P''} \quad \frac{P \rightsquigarrow_{\tau} P' \quad P' \rightsquigarrow_e P''}{P \rightsquigarrow_e P''}$$

$$\frac{P \rightsquigarrow_{\checkmark} P' \quad P' \rightsquigarrow_{\tau} P''}{P \rightsquigarrow_{\checkmark} P''} \quad \frac{P \rightsquigarrow_{\tau} P' \quad P' \rightsquigarrow_{\checkmark} P''}{P \rightsquigarrow_{\checkmark} P''}$$

ABSORPTION

7:10 An Operational Semantics in Isabelle/HOL-CSP

$$\frac{}{SKIP \rightsquigarrow_{\checkmark} \Omega SKIP} SKIP$$

$$\frac{cont\ f \quad P = (\mu x. f\ x)}{P \rightsquigarrow_{\tau} f\ P} \text{ FIXED POINT}$$

$$\frac{}{e \rightarrow P \rightsquigarrow_e P} \quad \frac{e \in A}{\Box a \in A \rightarrow P\ a \rightsquigarrow_e P\ e} \quad \frac{e \in A}{\Box a \in A \rightarrow P\ a \rightsquigarrow_e P\ e}$$

PREFIX

$$\frac{}{P \sqcap Q \rightsquigarrow_{\tau} P} \quad \frac{}{P \sqcap Q \rightsquigarrow_{\tau} Q} \quad \frac{e \in A}{\Box a \in A. P\ a \rightsquigarrow_{\tau} P\ e}$$

INTERNAL CHOICE

$$\frac{P \rightsquigarrow_{\tau} P'}{P \sqcap Q \rightsquigarrow_{\tau} P' \sqcap Q} \quad \frac{P \rightsquigarrow_e P'}{P \sqcap Q \rightsquigarrow_e P'} \quad \frac{P \rightsquigarrow_{\checkmark} P'}{P \sqcap Q \rightsquigarrow_{\checkmark} \Omega SKIP}$$

$$\frac{Q \rightsquigarrow_{\tau} Q'}{P \sqcap Q \rightsquigarrow_{\tau} P \sqcap Q'} \quad \frac{Q \rightsquigarrow_e Q'}{P \sqcap Q \rightsquigarrow_e Q'} \quad \frac{Q \rightsquigarrow_{\checkmark} Q'}{P \sqcap Q \rightsquigarrow_{\checkmark} \Omega SKIP}$$

EXTERNAL CHOICE

$$\frac{}{P \triangleright Q \rightsquigarrow_{\tau} Q} \quad \frac{P \rightsquigarrow_{\tau} P'}{P \triangleright Q \rightsquigarrow_{\tau} P' \triangleright Q} \quad \frac{P \rightsquigarrow_e P'}{P \triangleright Q \rightsquigarrow_e P'} \quad \frac{P \rightsquigarrow_{\checkmark} P'}{P \triangleright Q \rightsquigarrow_{\checkmark} \Omega SKIP}$$

SLIDING CHOICE

$$\frac{P \rightsquigarrow_{\tau} P'}{P ; Q \rightsquigarrow_{\tau} P' ; Q} \quad \frac{P \rightsquigarrow_e P'}{P ; Q \rightsquigarrow_e P' ; Q} \quad \frac{P \rightsquigarrow_{\checkmark} P' \quad Q \rightsquigarrow_{\tau} Q'}{P ; Q \rightsquigarrow_{\tau} Q'}$$

SEQUENTIAL COMPOSITION

$$\frac{P \rightsquigarrow_{\tau} P'}{P \setminus B \rightsquigarrow_{\tau} P' \setminus B} \quad \frac{P \rightsquigarrow_{\checkmark} P'}{P \setminus B \rightsquigarrow_{\checkmark} \Omega SKIP}$$

$$\frac{e \notin B \quad P \rightsquigarrow_e P'}{P \setminus B \rightsquigarrow_e P' \setminus B} \quad \frac{e \in B \quad P \rightsquigarrow_e P'}{P \setminus B \rightsquigarrow_{\tau} P' \setminus B}$$

HIDING

$$\frac{P \rightsquigarrow_{\tau} P'}{P \llbracket S \rrbracket Q \rightsquigarrow_{\tau} P' \llbracket S \rrbracket Q} \quad \frac{e \notin S \quad P \rightsquigarrow_e P'}{P \llbracket S \rrbracket Q \rightsquigarrow_e P' \llbracket S \rrbracket Q} \quad \frac{P \rightsquigarrow_{\checkmark} P'}{P \llbracket S \rrbracket Q \rightsquigarrow_{\tau} SKIP \llbracket S \rrbracket Q}$$

$$\frac{Q \rightsquigarrow_{\tau} Q'}{P \llbracket S \rrbracket Q \rightsquigarrow_{\tau} P \llbracket S \rrbracket Q'} \quad \frac{e \notin S \quad Q \rightsquigarrow_e Q'}{P \llbracket S \rrbracket Q \rightsquigarrow_e P \llbracket S \rrbracket Q'} \quad \frac{Q \rightsquigarrow_{\checkmark} Q'}{P \llbracket S \rrbracket Q \rightsquigarrow_{\tau} P \llbracket S \rrbracket SKIP}$$

$$\frac{e \in S \quad P \rightsquigarrow_e P' \quad Q \rightsquigarrow_e Q'}{P \llbracket S \rrbracket Q \rightsquigarrow_e P' \llbracket S \rrbracket Q'} \quad \frac{}{SKIP \llbracket S \rrbracket SKIP \rightsquigarrow_{\checkmark} \Omega SKIP}$$

SYNCHRONIZATION

$$\begin{array}{c}
\frac{P \rightsquigarrow_{\tau} P'}{P \Delta Q \rightsquigarrow_{\tau} P' \Delta Q} \quad \frac{P \rightsquigarrow_e P'}{P \Delta Q \rightsquigarrow_e P' \Delta Q} \quad \frac{P \rightsquigarrow_{\checkmark} P'}{P \Delta Q \rightsquigarrow_{\checkmark} \Omega \text{ SKIP}} \\
\frac{Q \rightsquigarrow_{\tau} Q'}{P \Delta Q \rightsquigarrow_{\tau} P \Delta Q'} \quad \frac{Q \rightsquigarrow_e Q'}{P \Delta Q \rightsquigarrow_e P \Delta Q'} \quad \frac{Q \rightsquigarrow_{\checkmark} Q'}{P \Delta Q \rightsquigarrow_{\checkmark} \Omega \text{ SKIP}} \\
\text{INTERRUPT} \\
\hline
\frac{P \rightsquigarrow_{\tau} P'}{P \Theta a \in A. Q a \rightsquigarrow_{\tau} P' \Theta a \in A. Q a} \quad \frac{P \rightsquigarrow_{\checkmark} P'}{P \Theta a \in A. Q a \rightsquigarrow_{\checkmark} \Omega \text{ SKIP}} \\
\frac{e \notin A \quad P \rightsquigarrow_e P'}{P \Theta a \in A. Q a \rightsquigarrow_e P' \Theta a \in A. Q a} \quad \frac{e \in A \quad P \rightsquigarrow_e P'}{P \Theta a \in A. Q a \rightsquigarrow_e Q e} \\
\text{THROW} \\
\hline
\frac{P \alpha \rightsquigarrow_{\tau} P'}{\text{Renaming } P f \beta \rightsquigarrow_{\tau} \text{Renaming } P' f} \\
\frac{f a = b \quad P \alpha \rightsquigarrow_a P'}{\text{Renaming } P f \beta \rightsquigarrow_b \text{Renaming } P' f} \quad \frac{P \alpha \rightsquigarrow_{\checkmark} P'}{\text{Renaming } P f \beta \rightsquigarrow_{\checkmark} \Omega_{\beta} \text{ SKIP}} \\
\text{RENAMING}
\end{array}$$

5 Big Steps Semantics

The notation $P / [e]$ sometimes appearing in the classical literature will now be given a formal definition in terms of the *After_{tick}* (Definition 12) operator. From there, the generalization to an operator “connecting” two processes P and Q via a trace s – analogously to the notion of δ function in automata theory – is straightforward. In the same manner, we can combine small steps transitions to a trace transition. These generalized notions will allow for both establishing new formats of refinement proofs as well as (bi)simulation theorems.

5.1 Extensions to Traces

The definition of the generalized *After* operator proceeds inductively:

► **Definition 13** (*After_{trace}* operator).

fun *After_{trace}* :: $\langle [\alpha \text{ process}, \alpha \text{ trace}] \Rightarrow \alpha \text{ process} \rangle$ (**infixl** $\langle \text{after}_{\tau} \rangle$ 77)
where $\langle P \text{ after}_{\tau} [] = P \rangle$
 $| \quad \langle P \text{ after}_{\tau} (e \# s) = (P \text{ after}_{\checkmark} e) \text{ after}_{\tau} s \rangle$

The definition of the generalized trace-transition is done analogously:

► **Definition 14** (Transition with a trace).

inductive *trace-trans* :: $\langle [\alpha \text{ process}, \alpha \text{ trace}, \alpha \text{ process}] \Rightarrow \text{bool} \rangle$ ($\langle - / \rightsquigarrow^* - / \rightarrow [50, 3, 51] 50 \rangle$)
where $\text{trace-}\tau\text{-trans} : \langle P \rightsquigarrow_{\tau} P' \Longrightarrow P \rightsquigarrow^* [] P' \rangle$
 $| \quad \text{trace-tick-trans} : \langle P \rightsquigarrow_{\checkmark} P' \Longrightarrow P \rightsquigarrow^* [\checkmark] P' \rangle$
 $| \quad \text{trace-Cons-ev-trans} : \langle P \rightsquigarrow_e P' \Longrightarrow P' \rightsquigarrow^* s P'' \Longrightarrow P \rightsquigarrow^* (ev e) \# s P'' \rangle$

The *After_{trace}* operator and the trace transition $P \rightsquigarrow^* s Q$ are deeply related, which is expressed in the following theorem:

► **Theorem 15** (Bridge between trace transition and $After_{trace}$ operator).

$P \rightsquigarrow^* s Q$ if and only if $s \in \mathcal{T} P \wedge P \text{ after}_{\mathcal{T}} s \rightsquigarrow_{\mathcal{T}} Q$.

Informally spoken, $P \text{ after}_{\mathcal{T}} s$ is the the least deterministic process that we can expect from process P after the trace s . This interpretation will become clearer when we will instantiate the formal $(\rightsquigarrow_{\mathcal{T}})$ -relation of the locale with concrete refinements in Section 6.

Since the projections for the $After_{trace}$ operator are relatively easy to handle, this theorem is an important new weapon in our arsenal. We will illustrate this by the following “reality checks” which have concrete applications when certifying traces or divergences.

► **Theorem 16** (Reality Checks).

- We have $s \in \mathcal{T} P$ if and only if $\exists Q. P \rightsquigarrow^* s Q$.
- Under the assumption $\forall P. P \rightsquigarrow_{\mathcal{T}} \perp \longrightarrow P = \perp$ of unicity of the least element of $(\rightsquigarrow_{\mathcal{T}})$, a trace $tickFree\ s$ verifies $s \in \mathcal{D} P$ if and only if $P \rightsquigarrow^* s \perp$.
- Under the assumption $\forall P Q. P \rightsquigarrow_{\mathcal{T}} Q \longrightarrow P \sqsubseteq_F Q$ that a τ transition implies F -refinement, and if $tickFree\ s$, we have $(s, X) \in \mathcal{F} P$ if and only if $\exists Q. P \rightsquigarrow^* s Q \wedge X \in \mathcal{R} Q$.

(where $\mathcal{R} Q$ is the set of refusals of Q , defined as $\{X \mid ([], X) \in \mathcal{F} Q\}$).

5.2 Strong Induction and (Bi)Simulations

The following theorem (and its generalizations not shown here) represents a new form of induction over the set of reachable processes:

► **Theorem 17** (Strong Induction for Refinements).] Let f be a continuous function:

$$\begin{aligned} & \llbracket \exists s \in \mathcal{T} P. tickFree\ s \wedge Q = P \text{ after}_{\mathcal{T}} s; \\ & \wedge s\ x. \llbracket s \in \mathcal{T} P; \forall y. (\exists s \in \mathcal{T} P. tickFree\ s \wedge y = P \text{ after}_{\mathcal{T}} s) \longrightarrow x \sqsubseteq_{FD} y \rrbracket \\ & \implies f\ x \sqsubseteq_{FD} P \text{ after}_{\mathcal{T}} s \rrbracket \\ & \implies (\mu x. f\ x) \sqsubseteq_{FD} Q \end{aligned}$$

Note that as in Hoare and Jifengs approach, there will always be infinite sequences of τ transitions. This is a consequence of the fact that the FD-refinement is reflexive. More generally speaking, since any process equality $P = Q$ is subsumed by the reflexivity $P \rightsquigarrow_{\mathcal{T}} Q$, this is unavoidable: unfolding fixed points, for example, also falls into the category of infinite sequences of τ transitions. Compared to classical operational semantics in CCS, which is defined purely in terms of syntactic manipulations on process-terms, this means that we can never have “strong simulations” in our denotational framework, which is based on higher-order abstract syntax and a congruence generated from the equality on the semantic domain. Rather, we will target *weak* transitions resp. simulations, which are, as we argue, more suited for the semantic treatment we are heading for.

6 The Construction put into a Global Perspective

6.1 Transitions as Local Refinements

We use a **sublocale** to partially instantiate $OpSemTransitions$ with (\sqsubseteq_{FD}) , i. e. by leaving Ψ and Ω as parameters. In this context we prove that we have only one remaining hypothesis : $\llbracket \checkmark \in Q^0; P \text{ }_{FD} \rightsquigarrow_{\mathcal{T}} Q \rrbracket \implies \Omega P \text{ }_{FD} \rightsquigarrow_{\mathcal{T}} \Omega Q$, which is obvious if Ω takes the value $STOP$

(a choice commonly used in the literature, whether explicitly stated or implied). However, in principle, any constant function is acceptable for Ω since we do not care about the traces after a termination⁵.

Independent of this choice, we recover all the rules of operational semantics, and even better for the “reality checks” (Theorem 16) since we get rid of the hypotheses.

However, it remains unfortunate that in the right hand side of the equivalence for failures appears the denotational notion of refusals.

One work-around for this problem are *deterministic processes*. A process P is said to be *deterministic* if $\forall s e. s @ [e] \in \mathcal{T} P \longrightarrow (s, \{e\}) \notin \mathcal{F} P$. We will not detail much this notion here⁶, but to cut a long story short we prove that being in the refusals set of a *deterministic* process P is the same as non intersecting its *initials* P^0 . Moreover, the notion of *deterministic* is preserved by *tickFree* trace transitions. We finally prove:

► **Theorem 18** (Deterministic Version of failures reality Check).

Assuming *deterministic* P and *tickFree* s , we have:

$$(s, X) \in \mathcal{F} P \text{ if and only if } \exists Q. P \text{ FD}^* s Q \wedge X \cap Q^0 = \emptyset.$$

It came as a pleasant surprise when we observed that the same argument applies for the trace divergence refinement (\sqsubseteq_{DT}). Initially defined and studied in [31] for pure curiosity, it behaves remarkably well: the only remaining hypothesis is a monotony for $\Omega : \llbracket \checkmark \in Q^0; P \text{ DT}^* \tau Q \rrbracket \implies \Omega P \text{ DT}^* \tau \Omega Q$ and we recover all the rules of the operational semantics.

Of course by restricting ourselves to traces and the divergences, we can not reason about failures anymore.

It turns out that it is also possible to instantiate the τ transition in the **locale** *OpSemTransitions* with the failure refinement or the trace refinement. However, for them, the result is somewhat unimpressive. The following table summarizes the rule sets corresponding to operators which can be established.

	basic	(\square)	($;$)	$P \setminus S$	$P \llbracket S \rrbracket Q$	(\triangleright)	(\triangle)	Throw	Renaming
(\sqsubseteq_{FD})	✓	✓	✓	✓	✓	✓	✓	✓	✓
(\sqsubseteq_{DT})	✓	✓	✓	✓	✓	✓	✓	✓	✓
(\sqsubseteq_F)	✓	✗	✗	✓	✗	✗	✗	✗	✗
(\sqsubseteq_T)	✓	✓	✗	✓	✗	✓	✓	✗	✗

where “basic” refers to the rules for absorption, *SKIP*, $e \rightarrow P$, $\square a \in A \rightarrow P a$, $\sqcap a \in A \rightarrow P a$, $P \sqcap Q$ and $\mu X. f X$ that we get as soon as we instantiate *OpSemTransitions*.

Being able to instantiate the locale represents the main result: we can formally derive an operational semantics from the denotational one developed in HOL-CSP. This also constitutes a formal proof that the expected rules are consistent (see Section 4).

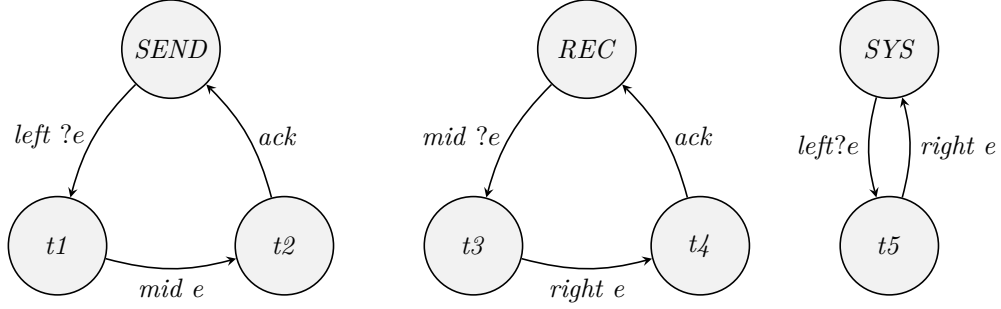
And last but not least, we obtained actually two interesting variants with FD-refinement and DT-refinement, plus all the sub-variants resulting from the free choice of Ψ and Ω .

6.2 Running Example: the Copy Buffer Again

From the derived laws of Section 4 we can formally obtain the following LTSs for the Copy Buffer example presented in Subsection 2.5. Note that the τ -transition are collapsed thanks to the absorption rules, and with the properties of *initials* we ensure that no external transition is missed. Further note that $t1$ is a key for the term $mid e \rightarrow ack \rightarrow SEND$, $t2$ for $ack \rightarrow SEND$, $t3$ for $right e \rightarrow ack \rightarrow REC$, $t4$ for $ack \rightarrow REC$, and finally $t5$ for $right e \rightarrow SYS$.

⁵ Therefore, we are free to choose for $\Omega \lambda P. P \sqcap STOP$ or $\lambda P. Renaming P f$ or $\lambda P. e \rightarrow P, \dots$

⁶ We refer to [27].



■ **Figure 1** LTSs for the Copy Buffer Example.

6.3 Comparison with the Work of Jifeng and Hoare

As mentioned in Section 3, Hoare and Jifeng in [20] defined $P \rightsquigarrow_{\tau} Q \equiv P \sqsubseteq_{FD} Q$ and $P \rightsquigarrow_e Q \equiv P \sqsubseteq_{FD} (e \rightarrow Q) \sqcap P$. Looking at our version (instantiated with the FD-refinement), we differ on the external transition with $ev \ e \in P^0 \wedge P \text{ after } e \sqsubseteq_{FD} Q$ instead. What happened? Did we miss something?

In general, from the generic τ transition in our **locale**, we can define $P \text{ HOARE} \rightsquigarrow_e Q \equiv P \rightsquigarrow_{\tau} (e \rightarrow Q) \sqcap P$. We immediately prove that their version is stronger than ours i.e. $P \text{ HOARE} \rightsquigarrow_e Q \implies P \rightsquigarrow_e Q$. However, adding the two hypotheses of monotony on $(\rightsquigarrow_{\tau})$, we can prove the reciprocal. The situation is summarized in the following theorem:

► **Theorem 19** (Equivalence of Transitions).

Assuming a τ monotony for prefix: $\forall P P' e. P \rightsquigarrow_{\tau} P' \longrightarrow e \rightarrow P \rightsquigarrow_{\tau} e \rightarrow P'$
 and for Det: $\forall P P' Q. P = \perp \vee P' \neq \perp \longrightarrow P \rightsquigarrow_{\tau} P' \longrightarrow P \sqcap Q \rightsquigarrow_{\tau} P' \sqcap Q$,
 we have $P \text{ HOARE} \rightsquigarrow_e Q$ if and only if $P \rightsquigarrow_e Q$.

These two hypotheses are verified by all four refinement relations. In other words, the definition of Jifeng and Hoare is equivalent to ours as long as we do not consider \checkmark !

Indeed, as mentioned in Section 3, their definition can not handle \checkmark because the *prefix* operator only accepts a “real” event. In this sense one can say that our construction is a generalization. Furthermore, the *After_{tick}* operator gives a direct access to the least deterministic process that can be expected while doing an external transition, which is not easily accessible from the version of Hoare and Jifeng. Finally we note that the *After* operator is itself of interest, even if we restrict ourselves to a purely denotational reasoning⁷.

6.4 Discussion

Our construction and the resulting proof rules (Section 4) permit the following observations:

1. As a general rule, when looking at a transition involving the special event \checkmark , we obtain something like $P \otimes Q \rightsquigarrow_{\checkmark} \Omega \text{ SKIP}$. This is a consequence of Theorem 7 and Theorem 11.
2. The “absorption” rules at the beginning allow additional rules to be derived directly e.g.

$$\frac{P \rightsquigarrow_{\checkmark} P' \quad Q \rightsquigarrow_e Q'}{P ; Q \rightsquigarrow_e Q'} \quad \frac{P \rightsquigarrow_{\checkmark} P' \quad Q \rightsquigarrow_{\checkmark} Q'}{P ; Q \rightsquigarrow_{\checkmark} Q'}$$

3. About the termination of *Sync* operator, Roscoe postulates in [27] that:

$$\frac{P \rightsquigarrow_{\checkmark} P'}{P \llbracket S \rrbracket Q \rightsquigarrow_{\tau} \Omega' \llbracket S \rrbracket Q} \quad \frac{P \rightsquigarrow_{\checkmark} P'}{P \llbracket S \rrbracket Q \rightsquigarrow_{\tau} \Omega' \llbracket S \rrbracket Q} \quad \frac{}{\Omega' \llbracket S \rrbracket \Omega' \rightsquigarrow_{\checkmark} \Omega'}$$

⁷ The interested reader is referred to examples of fixed point induction to reason about deadlock freeness[4].

where Ω' is intended to denote any process that has already terminated. In the common interpretation that Ω' can be identified with $STOP$, these rules are incompatible with the denotational properties since we have $STOP \llbracket S \rrbracket STOP = STOP$ that can not make a \checkmark transition. Under the assumptions of the **locale**, we rather prove the rules of Section 4.

4. We deliberately focus in Section 4 on the operational rules that we found in the literature. In particular for the *Throw* operator, where the right argument remains inactive until an exception is triggered, we should not write a right τ transition rule like:

$$\frac{\forall a \in A. Q a \rightsquigarrow_{\tau} Q' a}{P \Theta a \in A. Q a \sqsubseteq_{FD} P \Theta a \in A. Q' a}$$

while this is true when instantiating $(\rightsquigarrow_{\tau})$ with (\sqsubseteq_{FD}) , (\sqsubseteq_{DT}) , (\sqsubseteq_F) or (\sqsubseteq_T) .

7 Related Work

In the introduction, we claimed that HOL-CSP is arguably the most *comprehensive* formalization of CSP; here, we'd like to substantiate this claim.

The theory of CSP has attracted a lot of interest since the eighties and nineties, both as a theoretical device as well as a modelling language to analyze complex concurrent systems. A wealth of theoretical articles appeared to investigate certain fragments and extensions of the core framework; it is therefore not surprising that attempts to their formalisations have been undertaken with the advent of interactive proof assistants.

Most noteworthy to these attempts is an early CSP trace semantics model in HOL System proposed by [11]. Its successor [10] presented a first failure-divergence semantics for a restricted set of operators and used the notion of a universal (polymorphic) alphabet⁸. Note that [34] tackled already with subtle difficulties concerning *is-process* and \checkmark .

The tool CSP-Prover [18] – based on a deep embedding of CSP in an Isabelle/HOL theory on the stable failures model – allows for the refinement verification [18] by using some automated support for induction. However, only if a process is divergence-free, its failures are the same as its stable failures. In our view, this is a too strong assumption for both a theory as well as a practical tool.

In the past few years, CSP benefited from a renewed interest with proof assistants. CSP Agda was introduced in 2026 [16] with an implementation quite different from HOL-CSP since it is based on coinductive data types. Only trace and stable failures semantics have been covered so far, and the library of proven laws is fairly modest [17]. In 2020, an operational semantics of CSP in Coq was introduced [13] by a direct definitional approach. The theory covers only trace refinement and a subset of CSP's operators, but offers rather well-developed proof automation for this language fragment close to conventional automata theory.

With respect to all these formalizations in HOL, we would like to remind the importance of the general fact that invariants (like *is-process*) or bridge theorems (like Hoare's $P \rightsquigarrow_e Q \equiv P \sqsubseteq_{FD} (e \rightarrow Q) \sqcap P$) do not simply generalise from one fragment to the next, and that features which are well studied in one fragment are not necessarily well-understood in the whole picture. It is our main contribution to provide an integrated formal theory that tackles with the complexities of the necessary generalisations. This involved a revision of the role of the *After* operator in the entire theory.

In the late nineties, research focused on automated verification tools for CSP, most notably on FDR (see [1] for the latest instance). It relies on an operational CSP semantics, that allows for a conversion of processes into labelled transition systems, where the states were normalized

⁸ Our first attempts for HOL-CSP [34] are based on an extended version of this theory ported to Isabelle/HOL

by the “axioms” derived from the denotational semantics. For finite event sets, FDR can reduce refinement proofs to bisimulation problems. With efficient compression techniques, state-elimination and factorization by semantic equivalence [26], FDR was successful in analysing some industrial applications. However, such a model checker can never handle infinite cases. Another similar model checking tool [30] implemented some more optimization techniques, such as partial order reduction, symmetric reduction, and parallel model checking, but is also restricted to the finite case. In a way, these tool require for their foundation integrated denotational/algebraic/operational techniques as provided by our theory.

Attempts to find characterizations of processes to generalise finite results to infinite ones by *data-independence* [21, 2, 25], a variant of parametric model-checking, have seen only a limited success. Roscoe developed a data independent technology to verify security protocols modelled with CSP/FDR, which allows the node to call infinite fresh values for nonces, thus infinite sequence of operations [25]. An extension of this work was proposed in [2] using the script language CSP_M . However, in their works, even though each agent in the security protocol can perform infinite number of operations, the number of agent entities remains finite. HOL-CSP satisfies the need to parameterization and high-order processes naturally [32] as a consequence of their *pcpo*-type structure. A formalization and theory development of CSP_M has been undertaken [3] but is out of the scope of this paper.

8 Conclusion

We presented a formalisation of a comprehensive semantic theory for CSP, a ‘classical’ language for the specification and analysis of concurrent systems studied in a rich body of literature. The theory comprises the denotational part (including recursion and permitting higher-order processes), the algebraic part paving the way for parametric refinement proofs involving fixed point induction, and the operational semantics part. The size of the latter, which constitutes an original contribution, is about 16 kLOC of Isabelle/HOL proofs.

The resulting framework offers new ways to reason consistently over denotationally defined CSP processes paving the way to symbolic execution of LTS-based representations of processes as well as the possibility to *certify* output from model-checkers like [1, 30], which excel in the calculational parts related to interleaving processes. As a by-product, the theory allows for new proof principles like strong induction (cf. Theorem 17).

An interesting line of future work is the development of a library of “process-bricks” containing, e. g., semaphores, monitors or just global variables like:

$$VAR \text{ Read Update} \equiv \mu x. (\lambda\sigma. (\text{Read!}\sigma \rightarrow x \sigma) \square (\text{Update?}\sigma' \rightarrow x \sigma'))$$

or non-deterministic key-generators like:

$$KEY \text{ chan} \equiv (\mu x. (\lambda\sigma. \text{chan!}a \in \sigma \rightarrow x (\sigma - \{a\}))) \mathbf{N}$$

which will have symbolic traces like:

$$\llbracket a \in \mathbf{N}; b \in \mathbf{N} - \{a\}; c \in \mathbf{N} - \{a, b\} \rrbracket \Longrightarrow [C a, C b, C c] \in \mathcal{T} (KEY C)$$

which can be derived both algebraically as well as operationally.

References

- 1 FDR4 - The CSP Refinement Checker. <https://www.cs.ox.ac.uk/projects/fdr/>, 2019.
- 2 Jing An, Lei Zhang, and Chun You. The design and implementation of data independence in the csp model of security protocol. *Advanced Materials Research*, 915-916:1386–1392, April 2014. doi:10.4028/www.scientific.net/AMR.915-916.1386.

- 3 Benoît Ballenghien, Safouan Taha, and Burkhart Wolff. HOL-CSPM - Architectural operators for HOL-CSP. *Archive of Formal Proofs*, 2023, 2023. URL: <https://www.isa-afp.org/entries/HOL-CSPM.html>.
- 4 Benoît Ballenghien and Burkhart Wolff. Operational Semantics formally proven in HOL-CSP. *Archive of Formal Proofs*, December 2023. URL: https://isa-afp.org/entries/HOL-CSP_OpSem.html.
- 5 G. Barrett. Model checking in practice: the t9000 virtual channel processor. *IEEE Transactions on Software Engineering*, 21(2):69–78, February 1995. doi:10.1109/32.345823.
- 6 S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *J. ACM*, 31(3):560–599, 1984.
- 7 S. D. Brookes and A. W. Roscoe. An improved failures model for communicating sequential processes. In Stephen D. Brookes, Andrew William Roscoe, and Glynn Winskel, editors, *Seminar on Concurrency*, pages 281–305, Berlin, Heidelberg, 1985. Springer.
- 8 Achim D. Brucker, Idir Aït-Sadoune, Nicolas Méric, and Burkhart Wolff. Using deep ontologies in formal software engineering. In Uwe Glässer, José Creissac Campos, Dominique Méry, and Philippe A. Palanque, editors, *Rigorous State-Based Methods - 9th International Conference, ABZ 2023, Nancy, France, May 30 - June 2, 2023, Proceedings*, volume 14010 of *Lecture Notes in Computer Science*, pages 15–32. Springer, 2023. doi:10.1007/978-3-031-33163-3_2.
- 9 Achim D. Brucker and Burkhart Wolff. Isabelle/dof, July 2022. doi:10.5281/zenodo.6810799.
- 10 Albert J. Camilleri. A higher order logic mechanization of the csp failure-divergence semantics. In Graham Birtwistle, editor, *IV Higher Order Workshop, Banff 1990*, pages 123–150, London, 1991. Springer.
- 11 Albert John Camilleri. Mechanizing CSP trace theory in higher order logic. *IEEE Trans. Software Eng.*, 16(9):993–1004, 1990.
- 12 Paolo Crisafulli, Safouan Taha, and Burkhart Wolff. Modeling and analysing cyber-physical systems in HOL-CSP. *Robotics Auton. Syst.*, 170:104549, 2023. doi:10.1016/J.ROBOT.2023.104549.
- 13 Carlos Alberto da Silva Carvalho de Freitas. A theory for communicating, sequential processes in coq, 2020. URL: <https://api.semanticscholar.org/CorpusID:259373665>.
- 14 A.A.A. Donovan and B.W. Kernighan. *The Go Programming Language*. Addison-Wesley Professional Computing Series. Pearson Education, 2015.
- 15 C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1985.
- 16 Bashar Igried and Anton Setzer. Programming with monadic csp-style processes in dependent type theory. In *Proceedings of the 1st International Workshop on Type-Driven Development, TyDe 2016*, pages 28–38, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2976022.2976032.
- 17 Bashar Igried and Anton Setzer. Trace and stable failures semantics for csp-agda. *arXiv preprint arXiv:1709.04714*, 2017.
- 18 Yoshinao Isobe and Markus Roggenbach. A complete axiomatic semantics for the CSP stable-failures model. In *CONCUR 2006 - Concurrency Theory, 17th International Conference, Bonn, Germany, August 27-30, 2006*, pages 158–172, 2006.
- 19 Yoshinao Isobe and Markus Roggenbach. Csp-prover: a proof tool for the verification of scalable concurrent systems. *Information and Media Technologies*, 5(1):32–39, 2010. doi:10.11185/imt.5.32.
- 20 He Jifeng and CAR Hoare. From algebra to operational semantics. *Information Processing Letters*, 45(2):75–80, 1993.
- 21 Ranko S. Lazic. *A semantic study of data-independence with applications to the mechanical verification of concurren*. PhD thesis, University of Oxford, 1999.
- 22 Olaf Müller, Tobias Nipkow, David von Oheimb, and Oskar Slotosch. HOLCF = HOL + LCF. *J-fp*, 9(2):191–223, 1999. doi:10.1017/S095679689900341X.

- 23 Pasquale Noce. Conservation of CSP noninterference security under sequential composition. *Archive of Formal Proofs*, 2016. URL: https://www.isa-afp.org/entries/Noninterference_Sequential_Composition.shtml.
- 24 A. W. Roscoe. An alternative order for the failures model. *J. Log. Comput.*, 2:557–577, 1992.
- 25 A. W. Roscoe and Philippa J. Broadfoot. Proving security protocols with model checkers by data independence techniques. *Journal of Computer Security*, 7(1):147–190, 1999.
- 26 A. W. Roscoe, P. H. B. Gardiner, M. H. Goldsmith, J. R. Hulance, D. M. Jackson, and J. B. Scattergood. Hierarchical compression for model-checking csp or how to check 1020 dining philosophers for deadlock. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 133–152, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.
- 27 A.W. Roscoe. *Theory and Practice of Concurrency*. Prentice Hall, 1997.
- 28 A.W. Roscoe. *Understanding Concurrent Systems*. Springer-Verlag, Berlin, Heidelberg, 1st edition, 2010.
- 29 Dana Scott. Continuous lattices. In F. W. Lawvere, editor, *Toposes, Algebraic Geometry and Logic*, pages 97–136, Berlin, Heidelberg, 1972. Springer.
- 30 Jun Sun, Yang Liu, Jin Song Dong, and Jun Pang. Pat: Towards flexible verification under fairness. In Ahmed Bouajjani and Oded Maler, editors, *Computer Aided Verification*, pages 709–714, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.
- 31 Safouan Taha, Burkhart Wolff, and Lina Ye. The HOL-CSP refinement toolkit. *Arch. Formal Proofs*, 2020, 2020. URL: https://www.isa-afp.org/entries/CSP_RefTK.html.
- 32 Safouan Taha, Burkhart Wolff, and Lina Ye. Philosophers may dine - definitively! In Brijesh Dongol and Elena Troubitsyna, editors, *Integrated Formal Methods - 16th International Conference, IFM 2020, Lugano, Switzerland, November 16-20, 2020, Proceedings*, volume 12546 of *Lecture Notes in Computer Science*, pages 419–439. Springer, 2020. doi:10.1007/978-3-030-63461-2_23.
- 33 Safouan Taha, Lina Ye, and Burkhart Wolff. HOL-CSP Version 2.0. *Archive of Formal Proofs*, April 2019. URL: <http://isa-afp.org/entries/HOL-CSP.html>.
- 34 Haykal Tej and Burkhart Wolff. A corrected failure divergence model for CSP in Isabelle/HOL. In John S. Fitzgerald, Cliff B. Jones, and Peter Lucas, editors, *Formal Methods Europe (FME)*, volume 1313 of *LNCS*, pages 318–337. Springer, 1997. doi:10.1007/3-540-63533-5_17.
- 35 Jim Woodcock and Ana Cavalcanti. The semantics of circus. In Didier Bert, Jonathan P. Bowen, Martin C. Henson, and Ken Robinson, editors, *ZB 2002: Formal Specification and Development in Z and B, 2nd International Conference of B and Z Users, Grenoble, France, January 23-25, 2002, Proceedings*, volume 2272 of *Lecture Notes in Computer Science*, pages 184–203. Springer, 2002. doi:10.1007/3-540-45648-1_10.
- 36 Jim Woodcock, Ana Cavalcanti, Simon Foster, Marcel Oliveira, Augusto Sampaio, and Frank Zeyda. Utp, circus, and isabelle. In Jonathan P. Bowen, Qin Li, and Qiwen Xu, editors, *Theories of Programming and Formal Methods - Essays Dedicated to Jifeng He on the Occasion of His 80th Birthday*, volume 14080 of *Lecture Notes in Computer Science*, pages 19–51. Springer, 2023. doi:10.1007/978-3-031-40436-8_2.

The Directed Van Kampen Theorem in Lean

Henning Basold  

Leiden Institute of Advanced Computer Science, Leiden University, The Netherlands

Peter Bruin  

Mathematisch Instituut, Leiden University, The Netherlands

Dominique Lawson  

Student, Leiden University, The Netherlands

Abstract

Directed topology augments the concept of a topological space with a notion of directed paths. This leads to a category of directed spaces, in which the morphisms are continuous maps respecting directed paths. Directed topology thereby enables an accurate representation of computation paths in concurrent systems that usually cannot be reversed.

Even though ideas from algebraic topology have analogues in directed topology, the directedness drastically changes how spaces can be characterised. For instance, while an important homotopy invariant of a topological space is its fundamental groupoid, for directed spaces this has to be replaced by the fundamental category because directed paths are not necessarily reversible.

In this paper, we present a Lean 4 formalisation of directed spaces and of a Van Kampen theorem for them, which allows the fundamental category of a directed space to be computed in terms of the fundamental categories of subspaces. Part of this formalisation is also a significant theory of directed spaces, directed homotopy theory and path coverings, which can serve as basis for future formalisations of directed topology. The formalisation in Lean can also be used in computer-assisted reasoning about the behaviour of concurrent systems that have been represented as directed spaces.

2012 ACM Subject Classification Theory of computation → Type theory; Mathematics of computing → Algebraic topology

Keywords and phrases Lean, Directed Topology, Van Kampen Theorem, Directed Homotopy Theory, Formalised Mathematics

Digital Object Identifier 10.4230/LIPIcs.ITP.2024.8

Related Version *Extended pre-print using Lean 3*: <https://arxiv.org/abs/2312.06506> [1]

Supplementary Material

Software (Lean Code): <https://github.com/Dominique-Lawson/Directed-Topology-Lean-4> [16]
archived at [swh:1:dir:479a73373a2bf508149f7d1b889b42304fe78a9e](https://sw.h1.dir:479a73373a2bf508149f7d1b889b42304fe78a9e)

Funding *Peter Bruin*: Partially supported by the Dutch Research Council (NWO/OCW), as part of the Quantum Software Consortium programme (project number 024.003.037).

1 Introduction

Any topological space is equipped with a set of *paths* (continuous maps from the unit interval into the space), which is closed under composition and reversion. However, one often needs to distinguish a subset of paths following a particular *direction*, for example to model non-reversible processes. One motivation stems from models of true concurrency [9], where executions are modelled as non-reversible paths in a space. For instance, two programs A and B can be executed *sequentially* in two ways: either we first run A and then B, or vice versa, see a) of Figure 1. This choice between two sequential linearisations corresponds to semantics of labelled transition systems, but it neglects potential parallel execution. To see this, suppose that A and B have no dependency or interaction and can be run in parallel. This situation can be modelled by admitting any path in the square from the bottom left



© Henning Basold, Peter Bruin, and Dominique Lawson;
licensed under Creative Commons License CC-BY 4.0

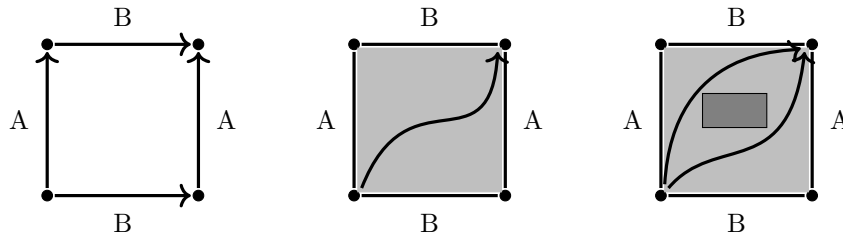
15th International Conference on Interactive Theorem Proving (ITP 2024).

Editors: Yves Bertot, Temur Kutsia, and Michael Norrish; Article No. 8; pp. 8:1–8:18



Leibniz International Proceedings in Informatics

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



■ **Figure 1** Possible execution paths of two programs A and B under three conditions: a) sequential (left), b) simultaneous (middle) and c) simultaneous with obstacles (right).

to the top right as a valid execution, with the intuition that going along the path tracks how far each of the processes has been run, see b) of Figure 1. The caveat is that processes can, in general, not be reversed and therefore the path may only ever go up and to the right, following the directions of the arrows. Suppose that there is a dependency between the processes, for instance they need to write to the same memory location. To prevent race conditions, we could rule out execution paths in which the processes access that memory location at the same time. This can be modelled by the space in Figure 1 c), where the darker rectangle is an obstacle that paths have to bypass. The two displayed paths in that space represent different memory access patterns: the lower path means that process B first gets access to the memory location, while the upper means that A first gets access. These two paths are essentially different because the observable behaviour of the system differs and because we cannot change the access pattern during execution. In contrast, the different paths in Figure 1 b) model executions that differ only in the relative execution speeds of A and B but are otherwise equivalent. By giving one process more execution time, we can always deform one path into another in this space. Finally, the space in Figure 1 a) has exactly two paths from the bottom left to top right, neither of which can be deformed to the other due to the absence of parallelism. This tells us that the spaces in Figure 1 all model different systems. The question is then how our intuition about relating execution paths can be made precise and how we can reason about these relations.

Directed topology and directed homotopy theory [8, 13] make the above intuition precise and enable the analysis of concurrent systems with the tools of algebraic topology. There are various ways to enforce direction in topological spaces, such as higher-dimensional automata [22, 20], spaces with a global order [10], spaces with local orders [7], streams [15], and various others [6, 11]. We will focus here on the notion of d-space [12], which represents a directed space as a topological space with a distinguished set of directed paths. It then turns out that reasoning about concurrent systems becomes reasoning about the homotopy type of d-spaces, that is, the relation between directed paths in a d-space.

An important strategy in building and analysing large systems is to prove local properties of subsystems and deduce properties of the composed system from these local properties. In algebraic topology, an important result allowing us to combine knowledge of the homotopy type of subspaces into knowledge about the whole space is the Van Kampen theorem [3]. This result expresses the fundamental group of a topological space as a pushout of fundamental groups of suitably chosen subspaces. It has been extended to d-spaces by Grandis [12]. To make the latter result applicable in larger systems, we set out in this paper to formalise the Van Kampen theorem for d-spaces in the proof assistant Lean [5], thereby enabling compositional reasoning about homotopy types of d-spaces and of concurrent systems modelled as d-spaces.

1.1 Contributions

Our main contribution is the formalisation of definitions and theorems relating to directed topology, in particular the Van Kampen Theorem. For this formalisation we used Lean 4.6.0-rc1 and we built upon the work already present in `mathlib` [18]. All of the formalisation can be found in the accompanying Git repository [16]. It consists of 5.6k lines of code distributed over 30 files. Throughout the article, excerpts from the formalisation are given to show the implementations of definitions and lemmas.

As directed topology has not been formalised before, our formalisation is a natural starting point for the development of a formalised directed topology. Our work has not yet been integrated into `mathlib`, but we plan on doing so in the near future.

1.2 Related work

There are currently no other formalisations of (parts of) directed topology. The undirected Van Kampen theorem has been formalised in Agda by Favonia and Shulman [14], and in Lean 2 by Van Doorn et al. [21]. In both cases, the formalisation uses synthetic homotopy theory in the form of univalent homotopy type theory, while our formalisation is analytic, that is, we define homotopy as concept derived from (directed) topological spaces. At the moment, `mathlib` does not contain a proof of the undirected Van Kampen Theorem.

1.3 Overview

In Section 2, we define the notion of directed spaces and directed maps and give a few examples. In Section 3, the definitions and some properties of directed homotopies and directed path homotopies are given. We use those to define relations on the set of directed paths between two points. In Section 4, the equivalence classes of paths under these relations are used to define the fundamental category. The Van Kampen Theorem is stated in Section 5 and we describe the connection between its proof and its formalisation in a precise manner. Finally, in Section 6 we reflect on the ideas presented in this article.

2 Directed Spaces

In this section, we will look at the basic structure of a directed space. With directed maps as morphisms, the category of directed spaces **dTop** is obtained.

2.1 Directed Spaces

A directed space is a topological space with a distinguished set of paths, whose elements are called directed paths. This set must contain all constant paths and must be closed under concatenation and monotone subparametrisation. We denote the concatenation of two paths by \odot .

► **Definition 1** (Directed space). *A directed space is a topological space X together with a subset P_X of the set of paths in X , satisfying the following three properties:*

1. *For any point $x \in X$, we have $0_x \in P_X$, where 0_x is the constant path in x .*
2. *For any two paths $\gamma_1, \gamma_2 \in P_X$ with $\gamma_1(1) = \gamma_2(0)$, we have $\gamma_1 \odot \gamma_2 \in P_X$.*
3. *For any path $\gamma \in P_X$ and any continuous, monotone map $\varphi : [0, 1] \rightarrow [0, 1]$, we have $\gamma \circ \varphi \in P_X$.*

The elements of P_X are called directed paths or dipaths.

8:4 The Directed Van Kampen Theorem in Lean

We will first consider some examples of directed spaces.

► **Example 2** (Directed unit interval). We can give the unit interval a rightward direction. This is done by taking $P_{[0,1]} = \{\varphi : [0, 1] \rightarrow [0, 1] \mid \varphi \text{ continuous and monotone}\}$. We will denote this directed space by I . More generally, every (pre)ordered space can be given a set of directed paths this way.

► **Example 3** (Product of directed spaces). If (X, P_X) and (Y, P_Y) are two directed spaces, then the space $X \times Y$ with the product topology can be made into a directed space by letting $P_{X \times Y} = \{t \mapsto (\gamma_1(t), \gamma_2(t)) \mid \gamma_1 \in P_X \text{ and } \gamma_2 \in P_Y\}$. As we will see in Section 2.2, with this set of directed paths both projection maps will be examples of directed maps and $(X \times Y, P_{X \times Y})$ becomes a product in a categorical sense.

► **Example 4** (Induced directed space). Let X be a topological space and (Y, P_Y) a directed space. Let a continuous map $f : X \rightarrow Y$ be given. If $\gamma : [0, 1] \rightarrow X$ is a path in X , then $f \circ \gamma : [0, 1] \rightarrow Y$ is a path in Y . We can make X into a directed space by taking $P_X = \{\gamma \in C([0, 1], X) \mid f \circ \gamma \in P_Y\}$. In the special case that X is a subspace of Y and f is the inclusion map, we find that every subspace of a directed space can be given a natural directedness.

We formalised the notion of a directed space by extending the `TopologicalSpace` class. In our formalisation, we do not explicitly use a set containing paths. Rather, being a directed path is a property of a path itself, analogously to how being open is a property of a set in the `TopologicalSpace` class. Paths in topological spaces have been implemented in `mathlib` in the file `Topology/Connected/PathConnected.lean`. A path has type `Path x y`, where its starting point is `x` and its endpoint is `y`. The definition of a directed space can be found in `directed_space.lean`.

```
class DirectedSpace (α : Type u) extends TopologicalSpace α where
  isDipath : ∀ {x y : α}, Path x y → Prop
  isDipath_constant : ∀ (x : α), IsDipath (Path.refl x)
  isDipath_concat : ∀ {x y z : α} {γ₁ : Path x y} {γ₂ : Path y z},
    IsDipath γ₁ → IsDipath γ₂ → IsDipath (Path.trans γ₁ γ₂)
  isDipath_reparam : ∀ {x y : α} {γ : Path x y} {t₀ t₁ : I}
    {f : Path t₀ t₁}, Monotone f → IsDipath γ →
    IsDipath (f.map (γ.continuous_toFun))
```

The term `IsDipath` determines whether a path is directed. The three other terms are exactly the three properties of a directed space. `Path.refl x` is the constant path in a point `x` and `Path.trans` is used for the concatenation of paths. The `mathlib` library only has support for reparametrisations of paths (meaning that the endpoints must remain the same), but we want to also allow strict subparametrisations. We do this by interpreting the subparametrisation f as a monotone path in $[0, 1]$. Then the path $\gamma \circ f$ can be obtained using `Path.map`, where we interpret γ as a continuous map.

In `constructions.lean`, various instances of directed spaces can be found: topological spaces with a preorder (Example 2), products of directed spaces (Example 3) and induced directedness (Example 4).

For brevity, we introduce a notation for the set of all directed paths between x and y .

► **Definition 5.** *If X is a directed space and $x, y \in X$ points, we use the shorthand notation $P_X(x, y)$ for the set $\{\gamma \in P_X \mid \gamma(0) = x \text{ and } \gamma(1) = y\}$.*

This definition can also be seen as a type for our formalisation. That is exactly how to interpret the structure `Dipath`, found in `dipath.lean`:

```
variable {X : Type u} [DirectedSpace X]
structure Dipath (x y : X) extends Path x y :=
  (dipath_toPath : IsDipath toPath)
```

It extends the `path` structure and depends on two points `x` and `y` in a directed space `X`. The term `dipath_toPath` has type `IsDipath toPath`. That means that the underlying path it extends must be a directed path. Due to the axioms of a directed space, we can define `Dipath.refl` and `Dipath.trans` analogously to their path-counterparts. However, `Path.symm`, the reversal of a path, cannot be converted to a directed variant as it is not guaranteed that the reversal of a directed path is directed.

We introduce a notation for a special kind of subpath of a directed path.

► **Definition 6.** *Let X be a directed space and $\gamma \in P_X$ a directed path. Given integers $n > 0$ and $1 \leq i \leq n$, we will define $\gamma_{i,n} \in P_X$ to be the path from $\gamma(\frac{i-1}{n})$ to $\gamma(\frac{i}{n})$ given by $\gamma_{i,n}(t) = \gamma(\frac{i+t-1}{n})$.*

We can now say what it means for a directed path to be covered by a cover of a directed space. This definition will play a big role in proving and formalising the Van Kampen Theorem for directed spaces.

► **Definition 7.** *Let X be a directed space, $U \subseteq X$ a subset and $\gamma \in P_X$ a directed path. We say that γ is contained in U if $\text{Im } \gamma \subseteq U$.*

► **Definition 8.** *Let X be a directed space and \mathcal{U} a cover of X . Let $\gamma \in P_X$ be a directed path and $n > 0$ an integer. We say that γ is n -covered (by \mathcal{U}) if $\gamma_{i,n}$ is contained in some $U_i \in \mathcal{U}$ for each $1 \leq i \leq n$.*

In `path_cover.lean` we formalise this definition of n -covered in the special case that \mathcal{U} consists of two sets X_0 and X_1 using induction:

```
variable {x y : X} (hX : X0 ∪ X1 = univ)

def covered (γ : Dipath x y) : Prop :=
  (range γ ⊆ X0) ∨ (range γ ⊆ X1)

def covered_partwise (γ : Dipath x y) (n : ℕ) : Prop := match n with
| Nat.zero => covered hX γ
| Nat.succ n =>
  covered hX (FirstPart γ (Fraction.ofPos (Nat.succ_pos n.succ))) ∧
  covered_partwise hX
  (SecondPart γ (Fraction.ofPos (Nat.succ_pos n.succ))) n
```

Here `covered` corresponds with γ being 1-covered: its image is either contained in X_0 or in X_1 . We use this definition to inductively define `covered_partwise`. As it is easier to start at zero in Lean, `covered_partwise hX γ n` corresponds with γ being $(n+1)$ -covered. In the case that $n = 0$, we have that `covered_partwise` simply agrees with `covered`. Otherwise, we use an induction step to define that `covered_partwise hX γ (Nat.succ n)` holds if the first part $\gamma_{1,n+2}$ is `covered` and the remainder of γ is `covered_partwise hX γ n`. Note the use of $n+2$ instead of $n+1$ due to the offset between the definitions. The remainder of `path_cover.lean` contains lemmas about conditions for being n -covered.

2.2 Directed Maps

As directed spaces are an extension of topological spaces, directed maps will be extensions of continuous maps. They will need to respect the extra directed structure. If a path in the domain space is given, a path in the codomain space can be obtained by composing the continuous map with the path. If the former is directed, so should be the latter.

► **Definition 9** (Directed map). *Let X and Y be two directed spaces. A directed map $f : X \rightarrow Y$ is a continuous map on the underlying topological spaces that furthermore satisfies: for any $\gamma \in P_X$, we have $f \circ \gamma \in P_Y$.*

By the construction of the product of directed spaces in Example 3, the continuous projection maps on both coordinates are directed: a directed path in the product space is a pair of directed paths and a projection returns the original directed path. Similarly, if a continuous map $f : X \rightarrow Y$ is used to induce a direction on X as in Example 4, then f becomes a directed map from X to Y , where X has the induced directedness.

In order to formalise the definition of a directed map in Lean, we define the property `Directed`, which expresses exactly that a continuous map between two directed spaces maps directed paths to directed paths. A directed map is then an extension of the `ContinuousMap` structure with a proof for being `Directed`.

```
variable {α β : Type*} [DirectedSpace α] [DirectedSpace β]
def Directed (f : C(α, β)) : Prop := ∀ {x y : α} (γ : Path x y),
  IsDipath γ → IsDipath (γ.map f.continuous_toFun)

structure DirectedMap extends ContinuousMap α β where
  protected directed_toFun : DirectedMap.Directed toContinuousMap
```

Within Lean, we use the notation $D(\alpha, \beta)$ for the type of directed maps between two spaces α and β . Directed paths are also instances of directed maps, because they map directed paths in I to monotone subparametrisation of themselves. `dipath.lean` contains definitions on how to convert the `Dipath` type to the `DirectedMap` type and the other way around. These are called `toDirectedMap` and `of_directedMap` respectively.

Directed spaces and directed maps form a category, which we will denote by `dTop`.

3 Directed Homotopies

In this section, we will look at directed homotopies and directed path homotopies. These two concepts realise the idea of deformation, while respecting the directedness of a directed space.

3.1 Homotopies

A directed homotopy is the deformation of one directed map into another.

► **Definition 10** (Directed homotopy). *Let X and Y be two directed spaces. A homotopy between two directed maps $f, g : X \rightarrow Y$ is a directed map $H : I \times X \rightarrow Y$ such that for all $x \in X$ we have $H(0, x) = f(x)$ and $H(1, x) = g(x)$, where the product $I \times X$ is taken between directed spaces, see Example 3.*

We say that H is a directed homotopy from f to g . This order matters, as unlike in the undirected case a directed homotopy cannot generally be reversed. In our formalisation, we adhere to the method used in defining homotopies between continuous maps in `mathlib`, which

can be found in `Topology/Homotopy/Basic.lean`. In an analogous manner, the structure extends the `DirectedMap (I x X) Y` structure and has two extra properties.

```
structure Dihomotopy (f0 f1 : D(X, Y)) extends D((I × X), Y) :=
  (map_zero_left : ∀ x, toFun (0, x) = f0.toFun x)
  (map_one_left : ∀ x, toFun (1, x) = f1.toFun x)
```

As a directed map is always a continuous map on the underlying topological spaces, we can convert a `Dihomotopy` to a `Homotopy`. Conversely, if we are given a `Homotopy` and we know that it is directed, we can obtain a `Dihomotopy`.

If $f : X \rightarrow Y$ is a directed map, there is an identity homotopy H from f to f , given by $H(t, x) = f(x)$. Also, if G is a directed homotopy from f to g and H a directed homotopy from g to h , we obtain a directed homotopy $G \otimes H$ from f to h given by

$$(G \otimes H)(t, x) = \begin{cases} G(2t, x), & t \leq \frac{1}{2}, \\ H(2t - 1, x), & \frac{1}{2} < t. \end{cases}$$

These constructions are called `refl` and `trans` in `directed_homotopy.lean`. In both cases we coerce a `Homotopy` to a `Dihomotopy`, by supplying proofs that the obtained homotopies are directed. Here we use the existing proofs in `mathlib` that the constructed maps are indeed homotopies, i.e. are continuous and satisfy the two mapping properties.

3.2 Path Homotopies

► **Definition 11** (Directed path homotopy). *Let X be a directed space and $x, y \in X$ two points. A directed path homotopy between two directed paths $\gamma_1, \gamma_2 \in P_X(x, y)$ is a directed homotopy $H : I \times I \rightarrow X$ from γ_1 to γ_2 such that additionally for all $t \in [0, 1]$ we have $H(t, 0) = x$ and $H(t, 1) = y$.*

In other words, a path homotopy is a homotopy between two paths that keeps both endpoints fixed. Again we say that H is a directed path homotopy from γ_1 to γ_2 . Between two paths γ_1 and γ_2 in I with the same endpoints exists a path homotopy under the condition that $\gamma_1(t) \leq \gamma_2(t)$ for all $t \in I$ as the following example shows.

► **Example 12.** Let $t_0, t_1 \in I$ be two points and $\gamma_1, \gamma_2 \in P_I(t_0, t_1)$. If $\gamma_1(t) \leq \gamma_2(t)$ for all $t \in I$, then there is a directed path homotopy H from γ_1 to γ_2 given by $H(t, s) = (1-t) \cdot \gamma_1(s) + t \cdot \gamma_2(s)$. It is continuous by continuity of paths, multiplication and addition. It can be shown that $H(a_0, b_0) \leq H(a_1, b_1)$ if $a_0 \leq a_1$ and $b_0 \leq b_1$. From this, it follows that H is directed, because a directed path in $I \times I$ is exactly a pair of monotone maps $I \rightarrow I$ by definition.

Note that H interpolates two paths γ_1 and γ_2 . The formalised proof of it being a directed map can be found in the file `interpolate.lean`.

Let $x, y, z \in X$ be three points, $\beta_1, \gamma_1 \in P_X(x, y)$ and $\beta_2, \gamma_2 \in P_X(y, z)$. If there are two directed path homotopies G from β_1 to γ_1 and H from β_2 to γ_2 , we can construct a directed path homotopy $G \odot H$ from $\beta_1 \odot \beta_2$ to $\gamma_1 \odot \gamma_2$ given by

$$(G \odot H)(t, s) = \begin{cases} G(t, 2s), & s \leq \frac{1}{2}, \\ H(t, 2s - 1), & \frac{1}{2} < s. \end{cases}$$

Let $x, y \in X$ be two points and $\gamma_1, \gamma_2 \in P_X(x, y)$. If there exists a path homotopy from γ_1 to γ_2 , we will write $\gamma_1 \rightsquigarrow \gamma_2$. This defines a relation on the set $P_X(x, y)$, but that relation is not guaranteed to be an equivalence relation, as it is generally not symmetric. This is due to

8:8 The Directed Van Kampen Theorem in Lean

the fact that the reversal of a directed path may not be directed. In order get an equivalence relation on the set of directed paths between two points, we will take the symmetric transitive closure of this relation.

► **Definition 13.** Let X be a directed space and $x, y \in X$ two points. We say that two dipaths $\gamma_1, \gamma_2 \in P_X(x, y)$ are equivalent, or $\gamma_1 \simeq \gamma_2$, if there is an integer $n \geq 0$ together with dipaths $\beta_i \in P_X(x, y)$, for each $1 \leq i \leq n$, such that

$$\gamma_1 \rightsquigarrow \beta_1 \leftarrow \dots \rightsquigarrow \beta_n \leftarrow \gamma_2.$$

This alternating sequence of arrows is also called a zigzag. As $\gamma_2 \leftarrow \gamma_2$ holds for any path γ_2 by reflexivity, we can always assume that there is an odd number of paths in a zigzag between two paths γ_1 and γ_2 . By taking $n = 0$, it follows that $\gamma_1 \simeq \gamma_2$ holds if $\gamma_1 \rightsquigarrow \gamma_2$. More precisely, \simeq is the smallest equivalence relation on $P_X(x, y)$ such that that property holds [17, p. 129]. As \simeq is an equivalence relation, we can talk about equivalence classes of paths, denoted by $[\gamma]$. An important property of these equivalence classes is that they are invariant under directed maps and path reparametrisation.

► **Lemma 14.** Let X, Y be directed spaces and $x, y \in X$. Let $\gamma_1, \gamma_2 \in P_X(x, y)$ and $f : X \rightarrow Y$ directed. If $\gamma_1 \simeq \gamma_2$, then $f \circ \gamma_1 \simeq f \circ \gamma_2$.

Proof. Let $n > 0$ odd and $\beta_i \in P_X(x, y)$ for $1 \leq i \leq n$ such that

$$\gamma_1 \rightsquigarrow \beta_1 \leftarrow \beta_2 \rightsquigarrow \dots \rightsquigarrow \beta_n \leftarrow \gamma_2.$$

If $H : I \times I \rightarrow X$ is a directed path homotopy from γ_1 to β_1 , then $f \circ H$ is a directed path homotopy from $f \circ \gamma_1$ to $f \circ \beta_1$. We find that $f \circ \gamma_1 \rightsquigarrow f \circ \beta_1$. Repeating this for all other arrows in the zigzag gives us

$$f \circ \gamma_1 \rightsquigarrow f \circ \beta_1 \leftarrow f \circ \beta_2 \rightsquigarrow \dots \rightsquigarrow f \circ \beta_n \leftarrow f \circ \gamma_2,$$

We conclude that $f \circ \gamma_1 \simeq f \circ \gamma_2$. ◀

► **Lemma 15.** Let X be a directed space and $x, y \in X$. Let $\gamma \in P_X(x, y)$ and $\varphi, \varphi' : I \rightarrow I$ continuous and monotone with $\varphi(0) = \varphi'(0) = 0$ and $\varphi(1) = \varphi'(1) = 1$. Then $\gamma \circ \varphi \simeq \gamma \circ \varphi'$.

Proof. As γ is a directed map from I to X , it is enough by Lemma 14 to show that $\varphi \simeq \varphi'$. Let $\beta_1 = \varphi \odot 0_1$ and $\beta_2 = 0_0 \odot \varphi'$. Then, by applying Example 12 three times, we obtain the zigzag $\varphi \rightsquigarrow \beta_1 \leftarrow \beta_2 \rightsquigarrow \varphi'$. This shows that $\varphi \simeq \varphi'$, completing the proof. ◀

In the next section, we will construct the fundamental category of a directed space. For that we need the following four additional equalities of equivalence classes.

► **Lemma 16.** Let X be a directed space and $x, y, z, w \in X$. Let $\beta_1, \gamma_1 \in P_X(x, y)$, $\beta_2, \gamma_2 \in P_X(y, z)$ and $\gamma_3 \in P_X(z, w)$ such that $\beta_1 \simeq \gamma_1$ and $\beta_2 \simeq \gamma_2$. Then the following holds:

1. $\beta_1 \odot \beta_2 \simeq \gamma_1 \odot \gamma_2$
2. $0_x \odot \gamma_1 \simeq \gamma_1$
3. $\gamma_1 \odot 0_y \simeq \gamma_1$
4. $(\gamma_1 \odot \gamma_2) \odot \gamma_3 \simeq \gamma_1 \odot (\gamma_2 \odot \gamma_3)$

Proof. Statements 2, 3 and 4 are direct applications of Lemma 15 as they are all reparametrisations. We will now show statement 1. Let $n, m > 0$ odd and $p_i, q_j \in P_X(x, y)$ for $1 \leq i \leq n$ and $1 \leq j \leq m$ such that

$$\beta_1 \rightsquigarrow p_1 \leftarrow p_2 \rightsquigarrow \dots \rightsquigarrow p_n \leftarrow \gamma_1 \quad \text{and} \quad \beta_2 \rightsquigarrow q_1 \leftarrow q_2 \rightsquigarrow \dots \rightsquigarrow q_m \leftarrow \gamma_2.$$

Let G be a directed path homotopy from β_1 to p_1 and H be the identity homotopy from β_2 to β_2 . Then $G \odot H$ is a directed path homotopy from $\beta_1 \odot \beta_2$ to $p_1 \odot \beta_2$. Repeating this, we obtain a zigzag

$$\beta_1 \odot \beta_2 \rightsquigarrow p_1 \odot \beta_2 \leftarrow p_2 \odot \beta_2 \rightsquigarrow \dots \rightsquigarrow p_n \odot \beta_2 \leftarrow \gamma_1 \odot \beta_2,$$

so $\beta_1 \odot \beta_2 \simeq \gamma_1 \odot \beta_2$. Analogously we obtain a zigzag

$$\gamma_1 \odot \beta_2 \rightsquigarrow \gamma_1 \odot q_1 \leftarrow \gamma_1 \odot q_2 \rightsquigarrow \dots \rightsquigarrow \gamma_1 \odot q_m \leftarrow \gamma_1 \odot \gamma_2.$$

This results in $\gamma_1 \odot \beta_2 \simeq \gamma_1 \odot \gamma_2$ and combining both equivalences gives us $\beta_1 \odot \beta_2 \simeq \gamma_1 \odot \gamma_2$. ◀

The definition of a directed path homotopy and the three lemmas above have all been formalised in `directed_path_homotopy.lean`. For the path homotopies, we followed the more general approach from `mathlib`, where we first defined directed homotopies that satisfy some property P . Thereafter we defined `DihomotopyRel` as directed homotopies that are fixed on a select subset of points. This is all defined in `directed_homotopy.lean`. A path homotopy is a homotopy that is fixed on both endpoints, that is, on $\{0, 1\} \subseteq I$, so we can define a directed path homotopy as

```
abbrev Dihomotopy (p₀ p₁ : Dipath x y) :=
  DirectedMap.DihomotopyRel p₀.toDirectedMap p₁.toDirectedMap {0, 1}
```

The construction \odot is called `hcomp` and \otimes is called `trans`. If $f, g \in D(I, I)$ are two directed maps with $f(t) \leq g(t)$ for all $t \in I$, the definition `Dihomotopy.reparam` constructs a homotopy from $\gamma \circ f$ to $\gamma \circ g$. This is done by composing γ and the homotopy obtained from Example 12. If H is a homotopy from γ_1 to γ_2 with $\gamma_1, \gamma_2 \in P_X(x, y)$, and $f : X \rightarrow Y$ is a directed map, then the homotopy from $f \circ \gamma_1$ to $f \circ \gamma_2$ given by $f \circ H$ is exactly what `Dihomotopy.map` entails.

Now we can formalise the relations \rightsquigarrow and \simeq . These are called `PreDihomotopic` and `Dihomotopic` respectively.

```
def PreDihomotopic : Prop := Nonempty (Dihomotopy p₀ p₁)
def Dihomotopic : Prop := EqvGen PreDihomotopic p₀ p₁
```

The term `Nonempty` means exactly that there exists some directed homotopy, which corresponds with our definition of \rightsquigarrow . `EqvGen` gives the smallest equivalence relation generated by a relation. The lemmas `map`, `reparam` and `hcomp` in the namespace `Dihomotopic` now correspond with Lemma 14, Lemma 15 and the first point of Lemma 16 respectively.

This gives us enough tools to construct the so called fundamental category.

4 The Fundamental Category

Using the properties found in Section 3.2, we can define a category that captures the information of all paths up to directed deformation in a directed space. This is the directed version of the fundamental groupoid.

► **Definition 17** (Fundamental Category). *Let X be a directed space. The fundamental category of X , denoted by $\overrightarrow{\Pi}(X)$, is the category that consists of:*

- *Objects: points $x \in X$.*
- *Morphisms: $\overrightarrow{\Pi}(X)(x, y) = P_X(x, y) / \simeq$.*
- *Composition: $[\gamma_2] \circ [\gamma_1] = [\gamma_1 \odot \gamma_2]$.*
- *Identity: $id_x = [0_x]$.*

8:10 The Directed Van Kampen Theorem in Lean

► **Remark 18.** The fact that this category is well defined follows from Lemma 16. Due to property 1, composition is well defined. Due to properties 2 and 3, the constant path behaves as an identity and property 4 gives us associativity.

Note that $\vec{\Pi}$ maps objects in **dTop** to objects in **Cat**. It turns out that it can also be defined on morphisms making it into a functor.

► **Definition 19.** Let $f : X \rightarrow Y$ be a directed map. We define $\vec{\Pi}(f) : \vec{\Pi}(X) \rightarrow \vec{\Pi}(Y)$ as the functor:

- On objects: $\vec{\Pi}(f)(x) = f(x)$.
- On morphisms: $\vec{\Pi}(f)([\gamma]) = [f \circ \gamma]$.

It is well behaved on morphisms, because of Lemma 14. It is straightforward to verify that $\vec{\Pi}(f)$ respects composition and identities.

In our formalisation, we follow the construction of the fundamental groupoid in `mathlib` found in `AlgebraicTopology/FundamentalGroupoid/Basic.lean` closely. Our implementation is found in `fundamental_category.lean`.

```
structure FundamentalCategory (X : Type u) where
  as : X

instance : CategoryTheory.Category (FundamentalCategory X) where
  Hom x y := Dipath.Dihomotopic.Quotient x.as y.as
  id x := [[Dipath.refl x.as]]
  comp {_ _ _} := Dipath.Dihomotopic.Quotient.comp
  id_comp {x _} f := Quotient.inductionOn f fun a =>
    show [[(Dipath.refl x.as).trans a]] = [[a]] from
      Quotient.sound (EqvGen.rel _ _ (Dipath.Dihomotopy.refl_trans a))
  comp_id {_ y} f := /- Proof omitted -/
  assoc {_ _ _} f g h := /- Proof omitted -/
```

We show that `FundamentalCategory X` is an instance of a category by defining the morphisms (`hom`), identities (`id`) and composition (`comp`). The morphisms between two objects `x` and `y` are given by `Dipath.Dihomotopic.Quotient x y`. This is the quotient of `Dipath x y` under the `Dihomotopic` relation and is defined in `directed_path_homotopy.lean`. The identity on `x` is then the equivalence class (denoted by `[[]]`) of the constant path in `x`. The composition of the equivalence classes of two compatible paths is defined as the equivalence class of the concatenation of the two paths in `Dipath.Dihomotopic.Quotient.comp`.

The proof that this defines a category is given by `id_comp`, `comp_id` and `assoc`. For example, `id_comp` requires us to show that the directed paths `(Dipath.refl x).trans a` and `a` are `dihomotopic`, corresponding to statement 2 of Lemma 16. The file also contains the definition of the $\vec{\Pi}$ -functor from `dTop` to `Cat`. Analogously to the undirected `mathlib` implementation, we use the notation `d π` for this functor.

5 The Van Kampen Theorem

In this section, we will state and prove the Van Kampen Theorem. We follow the proof of Grandis [12] and work out some of the details that were omitted there. In Section 5.2 we show how we have formalised this proof by comparing the proof to the Lean code.

5.1 The Van Kampen Theorem

Before we state and prove the theorem, we will define the notion of being covered for directed homotopies.

► **Definition 20.** *Let X be a directed space and \mathcal{U} a cover of X . Let $H : I \times I \rightarrow X$ be a directed homotopy and $n, m > 0$ two integers. We say that H is (n, m) -covered (by \mathcal{U}) if for all $1 \leq i \leq n$ and $1 \leq j \leq m$ the image of $[\frac{i-1}{n}, \frac{i}{n}] \times [\frac{j-1}{m}, \frac{j}{m}] \subseteq I \times I$ under H is contained in some $U \in \mathcal{U}$.*

By the Lebesgue Number Lemma [19, p. 179], for any homotopy H and open cover \mathcal{U} of X , there are $n, m > 0$ such that H is (n, m) -covered by \mathcal{U} .

► **Theorem 21 (Van Kampen Theorem).** *Let X be a directed space and X_1 and X_2 two open subspaces such that $X = X_1 \cup X_2$ and let $X_0 = X_1 \cap X_2$. Let $i_k : X_0 \rightarrow X_k$ and $j_k : X_k \rightarrow X$ be the inclusion maps, $k \in \{1, 2\}$. Then we obtain a pushout square in **Cat**:*

$$\begin{array}{ccc}
 \vec{\Pi}(X_0) & \xrightarrow{\vec{\Pi}(i_1)} & \vec{\Pi}(X_1) \\
 \vec{\Pi}(i_2) \downarrow & & \downarrow \vec{\Pi}(j_1) \\
 \vec{\Pi}(X_2) & \xrightarrow{\vec{\Pi}(j_2)} & \vec{\Pi}(X)
 \end{array}$$

Proof. As $j_1 \circ i_1 = j_2 \circ i_2$ and $\vec{\Pi}$ is a functor, the square is commutative. It remains to show it satisfies the universal property of a pushout square. Let \mathcal{C} be any category and $F_1 : \vec{\Pi}(X_1) \rightarrow \mathcal{C}$ and $F_2 : \vec{\Pi}(X_2) \rightarrow \mathcal{C}$ be two functors such that $F_1 \circ \vec{\Pi}(i_1) = F_2 \circ \vec{\Pi}(i_2)$. We will explicitly construct a functor $F : \vec{\Pi}(X) \rightarrow \mathcal{C}$ such that $F \circ \vec{\Pi}(j_1) = F_1$ and $F \circ \vec{\Pi}(j_2) = F_2$. The construction will show that this functor is necessarily unique with this property.

Step 1) The objects of $\vec{\Pi}(X)$ are exactly the points of X . If an object $x \in \vec{\Pi}(X)$ is also contained in $\vec{\Pi}(X_1)$, it holds that $F(x) = F(j_1(x)) = (F \circ \vec{\Pi}(j_1))(x)$. The desired condition $F \circ \vec{\Pi}(j_1) = F_1$ then requires us to define $F(x) = F_1(x)$. A similar argument gives us that if $x \in \vec{\Pi}(X_2)$ then $F(x) = F_2(x)$. As X_1 and X_2 cover X , for all $x \in \vec{\Pi}(X)$ we have

$$F(x) = \begin{cases} F_1(x), & x \in X_1, \\ F_2(x), & x \in X_2. \end{cases}$$

By the assumption that $F_1 \circ \vec{\Pi}(i_1) = F_2 \circ \vec{\Pi}(i_2)$ this is well defined, so we know how F must behave on objects.

Step 2) Let $[\gamma] : x \rightarrow y$ be a morphism in $\vec{\Pi}(X)$. Then there is an $n > 0$ such that γ is n -covered by the open cover $\{X_1, X_2\}$, with $\gamma_{i,n}$ contained in X_{k_i} , $k_i \in \{1, 2\}$. One important thing to note is that $\gamma_{i,n}$ can be both seen as a path in X and as a path in X_{k_i} by restricting its codomain. This matters when we talk about $[\gamma_{i,n}]$, as it could be a morphism in $\vec{\Pi}(X)$ and in $\vec{\Pi}(X_{k_i})$. Within this proof will always consider it as a morphism in $\vec{\Pi}(X_{k_i})$ and write $[j_{k_i} \circ \gamma_{i,n}]$ for the morphism in $\vec{\Pi}(X)$. Note that we have $[\gamma] = [j_{k_n} \circ \gamma_{n,n}] \circ \dots \circ [j_{k_1} \circ \gamma_{1,n}]$ in $\vec{\Pi}(X)$, as γ is equal to $\gamma_{1,n} \odot (\gamma_{2,n} \odot \dots (\gamma_{n-1,n} \odot \gamma_{n,n}))$ up to reparametrisation. Because

8:12 The Directed Van Kampen Theorem in Lean

we want F to be a functor and thus to respect composition, we find that necessarily

$$\begin{aligned}
 F[\gamma] &= F([j_{k_n} \circ \gamma_{n,n}] \circ \dots \circ [j_{k_1} \circ \gamma_{1,n}]) \\
 &= F[j_{k_n} \circ \gamma_{n,n}] \circ \dots \circ F[j_{k_1} \circ \gamma_{1,n}] \\
 &= F\left(\vec{\Pi}(j_{k_n})[\gamma_{n,n}]\right) \circ \dots \circ F\left(\vec{\Pi}(j_{k_1})[\gamma_{1,n}]\right) \\
 &= (F \circ \vec{\Pi}(j_{k_n}))[\gamma_{n,n}] \circ \dots \circ (F \circ \vec{\Pi}(j_{k_1}))[\gamma_{1,n}] \\
 &= F_{k_n}[\gamma_{n,n}] \circ \dots \circ F_{k_1}[\gamma_{1,n}].
 \end{aligned}$$

As multiple choices were made, we need to make sure that F is well defined this way. We do this by defining a map $F' : P_X \rightarrow \text{Mor}(\mathcal{C})$, where $\text{Mor}(\mathcal{C})$ is the collection of all morphisms in \mathcal{C} . The map is given by

$$F'(\gamma) = F_{k_n}[\gamma_{n,n}] \circ \dots \circ F_{k_1}[\gamma_{1,n}],$$

where γ is n -covered with $\gamma_{i,n}$ contained in X_{k_i} . In the next steps, we will first show that this map is well defined. Then we show that F' respects equivalence classes. From this it follows that F is well defined, as it is simply F' descended to equivalence classes.

Step 3) We first need to make sure that F' does not depend on any choices of k_i . In the case that $\gamma_{i,n}$ is contained in both X_1 and X_2 , the value of k_i can be either 1 or 2. The condition that $F_1 \circ \vec{\Pi}(i_1) = F_2 \circ \vec{\Pi}(i_2)$ assures us that both options give us the same morphism.

Step 4) The second choice we made is that of n . It is possible that γ is also m -covered for another integer $m > 0$, with $\gamma_{j,m}$ being contained in X_{p_j} . We want to show that

$$F_{k_n}[\gamma_{n,n}] \circ \dots \circ F_{k_1}[\gamma_{1,n}] = F_{p_m}[\gamma_{m,m}] \circ \dots \circ F_{p_1}[\gamma_{1,m}].$$

If we refine the partition of γ in n pieces into a partition of mn pieces, that partition will surely also be partwise covered. Let $l_i \in \{1, 2\}$ for all $1 \leq i \leq mn$ such that $\gamma_{i,mn}$ is contained in X_{l_i} . We now claim that for all $1 \leq i \leq n$ it holds that $F_{k_i}[\gamma_{i,n}] = F_{l_{mi}}[\gamma_{mi,mn}] \circ \dots \circ F_{l_{m(i-1)+1}}[\gamma_{m(i-1)+1,mn}]$. As $\gamma_{m(i-1)+j,mn}$ with $1 \leq j \leq m$ is a subparametrisation of $\gamma_{i,n}$, we may assume that $l_{m(i-1)+j} = k_i$. This is because F_1 and F_2 agree on $X_1 \cap X_2$. As F_{k_i} is a functor, the claim now follows because functors respect composition and because $\gamma_{i,n}$ is exactly the concatenation of all the smaller paths up to reparametrisation. By a similar claim for $F_{p_j}[\gamma_{j,m}]$ we find:

$$\begin{aligned}
 F_{k_n}[\gamma_{n,n}] \circ \dots \circ F_{k_1}[\gamma_{1,n}] &= F_{l_{mn}}[\gamma_{mn,mn}] \circ \dots \circ F_{l_1}[\gamma_{1,mn}] \\
 &= F_{p_m}[\gamma_{m,m}] \circ \dots \circ F_{p_1}[\gamma_{1,m}].
 \end{aligned}$$

We conclude that the definition is independent of the value of n . This makes F' well defined.

Step 5) Before we verify that F' is independent of the choice of representative γ , we will first show that F' satisfies two properties:

$$\forall x \in \vec{\Pi}(X) : F'(0_x) = \text{id}_{F(x)}. \quad (1)$$

$$\forall \gamma \in P_X(x, y), \delta \in P_X(y, z) : F'(\gamma \odot \delta) = F'(\delta) \circ F'(\gamma). \quad (2)$$

Let $x \in \vec{\Pi}(X)$ be given. If $x \in X_1$, then 0_x is already contained in X_1 and so by definition of F' we find $F'(0_x) = F_1[0_x] = \text{id}_{F_1(x)} = \text{id}_{F(x)}$. Otherwise it holds that $x \in X_2$, so $F'(0_x) = F_2[0_x] = \text{id}_{F_2(x)} = \text{id}_{F(x)}$. This proves Equation (1).

Let $\gamma \in P_X(x, y)$ and $\delta \in P_X(y, z)$ be two paths in X . We can then find an n such that both γ and δ are n -covered, with $\gamma_{i,n}$ contained in X_{k_i} and $\delta_{i,n}$ contained in X_{p_i} . Then $\gamma \odot \delta$ is $2n$ -covered as it holds that

$$(\gamma \odot \delta)_{i,2n} = \begin{cases} \gamma_{i,n}, & i \leq n, \\ \delta_{i-n,n}, & i > n. \end{cases}$$

We find:

$$\begin{aligned} F'(\delta \odot \gamma) &= \\ F_{p_n}[(\delta \odot \gamma)_{2n,2n}] \circ \dots \circ F_{p_1}[(\delta \odot \gamma)_{n+1,2n}] \circ F_{k_n}[(\delta \odot \gamma)_{n,2n}] \circ \dots \circ F_{k_1}[(\delta \odot \gamma)_{1,2n}] &= \\ (F_{p_n}[\delta_{n,n}] \circ \dots \circ F_{p_1}[\delta_{1,n}]) \circ (F_{k_n}[\gamma_{n,n}] \circ \dots \circ F_{k_1}[\gamma_{1,n}]) &= F'(\delta) \circ F'(\gamma). \end{aligned}$$

This shows that Equation (2) holds.

Step 6) We will now show that F' respects equivalence classes. Then it descends to the quotient and it follows that F is well defined. If $[\gamma] = [\delta]$ with δ another path from x to y , we want that

$$F'(\gamma) = F'(\delta). \tag{3}$$

Because of the way the equivalence classes are defined, it is enough to show this for γ and δ such that $\gamma \rightsquigarrow \delta$. Let in that case a directed path homotopy H from γ to δ be given. We take $n, m > 0$ such that H is (n, m) -covered by $\{X_1, X_2\}$. Firstly assume that $n > 1$. Restricting H to the rectangle $[0, \frac{1}{n}] \times [0, 1]$ gives us a directed path homotopy H_1 from γ to the directed path η given by $\eta(t) = H(\frac{1}{n}, t)$. By restricting H to the rectangle $[\frac{1}{n}, 1] \times [0, 1]$ we get a directed path homotopy H_2 from η to δ . It is clear that H_1 is $(1, m)$ -covered and that H_2 is $(n-1, m)$ -covered. By applying induction on n , we can conclude that it is enough to show that Equation (3) holds for $(1, m)$ -covered directed path homotopies, as we would obtain that $F'(\gamma) = F'(\eta) = F'(\delta)$.

Step 7) We will prove the case where H is $(1, m)$ -covered by showing a more general statement:

Let H be any directed homotopy – not necessarily a path homotopy – from one path $\gamma \in P_X(x, y)$ to another path $\delta \in P_X(x', y')$ that is $(1, m)$ -covered, $m > 0$. Let η_0 be the path given by $\eta_0(t) = H(t, 0)$ and η_1 be given by $\eta_1(t) = H(t, 1)$. Then $F'(\eta_0 \odot \delta) = F'(\gamma \odot \eta_1)$. We do this by induction on m .

In the case that $m = 1$, we have a homotopy contained in X_1 or X_2 . Without loss of generality, we can assume it is contained in X_1 . Let Γ_1 be the directed homotopy given by $\Gamma_1(t, s) = \eta_0(\min(t, s))$ from 0_x to η_0 . Let Γ_2 be the directed homotopy given by $\Gamma_2(t, s) = \eta_1(\max(t, s))$ from η_1 to $0_{y'}$. We then can construct a directed path homotopy from $(0_x \odot \gamma) \odot \eta_1$ to $(\eta_0 \odot \delta) \odot 0_{y'}$ given by $(\Gamma_1 \odot H) \odot \Gamma_2$. It is a directed path homotopy because $\Gamma_1(t, 0) = \eta_0(\min(t, 0)) = \eta_0(0) = x$ and $\Gamma_2(t, 1) = \eta_1(\max(t, 1)) = \eta_1(1) = y'$ for all $t \in I$. As η_0, η_1 and H are all contained in X_1 , this directed path homotopy will be contained in X_1 as well. We find that $[\gamma \odot \eta_1] = [\eta_0 \odot \delta]$ in $\vec{\Pi}(X_1)$. This gives us that $F'(\gamma \odot \eta_1) = F_1[\gamma \odot \eta_1] = F_1[\eta_0 \odot \delta] = F'(\eta_0 \odot \delta)$.

Let now $m > 1$ and assume the statement holds for $(1, m-1)$ -covered homotopies. We can restrict H to $[0, 1] \times [0, \frac{m-1}{m}]$ to obtain a $(1, m-1)$ -covered homotopy H_1 , say from γ_1 to δ_1 . Similarly, we can restrict H to $[0, 1] \times [\frac{m-1}{m}, 1]$ to obtain a $(1, 1)$ -covered homotopy H_2 , say from γ_2 to δ_2 . We write η' for the path given by $\eta'(t) = H(t, \frac{m-1}{m}) = H_1(t, 1) = H_2(t, 0)$.

8:14 The Directed Van Kampen Theorem in Lean

Note that $F'(\gamma) = F'(\gamma_2) \circ F'(\gamma_1)$ by definition, because γ_1 is $(m-1)$ -covered, γ_2 is 1-covered and γ is m -covered. Similarly it holds that $F'(\delta) = F'(\delta_2) \circ F'(\delta_1)$. We find:

$$\begin{aligned}
 F'(\gamma \odot \eta_1) &= F'(\eta_1) \circ F'(\gamma) && \text{(Equation (2))} \\
 &= F'(\eta_1) \circ (F'(\gamma_2) \circ F'(\gamma_1)) \\
 &= (F'(\eta_1) \circ F'(\gamma_2)) \circ F'(\gamma_1) \\
 &= (F'(\delta_2) \circ F'(\eta')) \circ F'(\gamma_1) && \text{(Case } m = 1\text{)} \\
 &= F'(\delta_2) \circ (F'(\eta') \circ F'(\gamma_1)) \\
 &= F'(\delta_2) \circ (F'(\delta_1) \circ F'(\eta_0)) && \text{(Induction Hypothesis)} \\
 &= (F'(\delta_2) \circ F'(\delta_1)) \circ F'(\eta_0) \\
 &= F'(\delta) \circ F'(\eta_0) \\
 &= F'(\eta_0 \odot \delta) && \text{(Equation (2)).}
 \end{aligned}$$

This proves the statement. From the statement we find that Equation (3) holds:

$$\begin{aligned}
 F'(\delta) &= F'(\delta) \circ \text{id}_x = F'(\delta) \circ F'(0_x) = F'(0_x \odot \delta) = \\
 F'(\gamma \odot 0_y) &= F'(0_y) \circ F'(\gamma) = \text{id}_x \circ F'(\gamma) = F'(\gamma).
 \end{aligned}$$

Here, the fourth equality follows from the statement. We conclude that F is well defined.

Step 8) As we have $F[\gamma] = F'(\gamma)$, it is immediate that F is a functor by Equation (1) and Equation (2). The equalities $F \circ \vec{\Pi}(j_1) = F_1$ and $F \circ \vec{\Pi}(j_2) = F_2$ hold by construction: if γ is contained in X_1 , then $\gamma_{1,1}$ is as well, so $(F \circ \vec{\Pi}(j_1))[\gamma] = F[j_1 \circ \gamma] = F'(\gamma) = F_1[\gamma_{1,1}] = F_1[\gamma]$. We conclude that the commutative square is indeed a pushout. ◀

5.2 Formalisation

In the formalisation of Theorem 21, we follow the constructive nature of its proof. It can be found in `directed_van_kampen.lean`. We have the following global variables, corresponding with the assumptions of the Van Kampen Theorem:

```

variable {X : dTopCat.{u}} {X1 X2 : Set X}
variable (hX : X1 ∪ X2 = Set.univ)
variable (X1_open : IsOpen X1) (X2_open : IsOpen X2)

```

Like in the proof, we introduce a category C and two functors $F_1 : \vec{\Pi}(X_1) \rightarrow C$ and $F_2 : \vec{\Pi}(X_2) \rightarrow C$. Using these we are going to explicitly construct a functor from $\vec{\Pi}(X)$ to C and show that it is unique. We will use that to prove that we indeed have a pushout square.

```

variable {C : CategoryTheory.Cat.{u, u}}
variable (F1 : (dπx (dTopCat.of X1) → C))
variable (F2 : (dπx (dTopCat.of X2) → C))
variable (h_comm : (dπm i1) ≫ F1 = (dπm i2) ≫ F2)
/- Here we use two abbreviations:
i1 = dTopCat.DirectedSubsetHom (Set.inter_subset_left X1 X2)
i2 = dTopCat.DirectedSubsetHom (Set.inter_subset_right X1 X2)
-/

```

The variable `h_comm` is the assumption that the two maps F_1 and F_2 out of C form a commutative square when composed with the inclusions $\vec{\Pi}(X_1) \rightarrow \vec{\Pi}(X)$ and $\vec{\Pi}(X_2) \rightarrow \vec{\Pi}(X)$. These inclusions are obtained by `DirectedSubsetHom`, defined in `dTop.lean`. This defines the inclusion morphism $X_0 \rightarrow X_1$ in **dTop** in the case that $X_0 \subseteq X_1 \subseteq X$. We start with defining the functor F on objects (**Step 1**).

```

def FunctorOnObj (x : dπx X) : C := Or.by_cases
  ((Set.mem_union x.as X1 X2).mp (Filter.mem_top.mpr hX x.as))
  (fun hx => F1.obj ⟨x.as, hx⟩)
  (fun hx => F2.obj ⟨x.as, hx⟩)

```

We use `Filter.mem_top.mpr hX x.as` to show that $x \in X_1 \cup X_2$. From this, we use `Set.mem_union` to obtain $x \in X_1$ or $x \in X_2$ and we can split by those cases to apply either F_1 or F_2 . We abbreviate `FunctorOnObj hX F1 F2` to `F_obj` in our formalisation to maintain clarity. After this definition, there are two lemmas that prove for $k \in \{1, 2\}$ that $F(x) = F_k(x)$ if $x \in X_k$.

In the proof of Theorem 21, F' is first defined and it is then shown to be a valid definition. Within our Lean formalisation, we have to do these two parts in the reverse order. Once we have shown that the construction is well-defined, we can define F' in our formalisation. That is why **Step 2** will be completed later.

We use the definitions of `covered` and `covered_partwise`, shown in Section 2, to define the mapping of morphisms inductively (**Step 3**):

```

def FunctorOnHomOfCovered {γ : Dipath x y} (hγ : covered hX γ) :
  F_obj ⟨x⟩ → F_obj ⟨y⟩ :=
  Or.by_cases hγ
  (fun hγ => FunctorOnHomOfCoveredAux1 hX h_comm hγ)
  (fun hγ => FunctorOnHomOfCoveredAux2 hX h_comm hγ)

def FunctorOnHomOfCoveredPartwiseAux {n : ℕ} :
  ∀ (x y : X) (γ : Dipath x y) (hγ : covered_partwise hX γ n),
  F_obj ⟨x⟩ → F_obj ⟨y⟩ :=
  Nat.recOn n
  (fun _ _ _ hγ => F0 hγ)
  (fun _ ih _ _ _ hγ => (F0 hγ.1) >> (ih _ _ _ hγ.2))

```

In `FunctorOnHomOfCovered` we define what to do with a path γ that is 1-covered, that is, we map it to $F_1[\gamma]$ or $F_2[\gamma]$ depending on whether γ is contained in X_1 or X_2 . It depends on `FunctorOnHomOfCoveredAux1`, which specifies what $F_1[\gamma]$ should be, as $[\gamma]$ is a morphism in $\vec{\Pi}(X)$ and not in $\vec{\Pi}(X_1)$. We use F_0 to abbreviate `FunctorOnHomOfCovered hX h_comm`. We can then use this base case to define `FunctorOnHomOfCoveredPartwiseAux` for an n -covered path inductively by applying F_0 to the first covered part of γ . In the construction of `FunctorOnHomOfCoveredPartwiseAux`, the variables x , y and γ are given explicitly in order to use induction. We use this definition in order to define `FunctorOnHomOfCoveredPartwise` which uses these implicitly and we abbreviate it to F_n to maintain readability.

Since n is an input of the definition, we need to show that it is independent of the choice of n . The lemma `functorOnHomOfCoveredPartwise_unique` captures this (**Step 4**).

```

lemma functorOnHomOfCoveredPartwise_unique {n m : ℕ} {γ : Dipath x y}
  (hγn : covered_partwise hX γ n) (hγm : covered_partwise hX γ m) :
  Fn hγn = Fm hγm :=
  /- Proof omitted -/

```

This lemma makes use of the following lemma that shows that the image remains the same if we refine the partition of γ , so when we use an nk -covering instead of an n -covering.

```

lemma functorOnHomOfCoveredPartwise_refine {n : ℕ} (k : ℕ) :
  Π {x y : X} {γ : Dipath x y} (hγn : covered_partwise hX γ n),
  Fn hγn = Fn (covered_partwise_refine hX n k hγn) :=
  /- Proof omitted -/

```

8:16 The Directed Van Kampen Theorem in Lean

Now we know that the image is independent of n , and because an $n > 0$ exists such that γ is n -covered (shown by `has_subpaths`), we can choose one such n and we obtain the following formalisation of F' , completing **Step 2**. We abbreviate this map to `Fh_aux`.

```
def FunctorOnHomAux (γ : Dipath x y) : F_obj ⟨x⟩ → F_obj ⟨y⟩ :=
  F_n (Classical.choose_spec (has_subpaths hX X1_open X2_open γ))
```

Now we show that Equation (1) and Equation (2) from the proof hold (**Step 5**).

```
lemma functorOnHomAux_refl {x : X} :
  Fh_aux (Dipath.refl x) = 1 (F_obj ⟨x⟩) :=
  /- Proof omitted -/
```

```
lemma functorOnHomAux_trans {x y z : X} (γ1 : Dipath x y)
  (γ2 : Dipath y z) :
  Fh_aux (γ1.trans γ2) = Fh_aux γ1 >>> Fh_aux γ2 :=
  /- Proof omitted -/
```

As shown in **Step 6**, we want to show that F' is invariant under the Dihomotopic relation. To do this we need to show the claim from the proof: if we have a directed homotopy H from f to g that is $(1, m)$ -covered, then $F'[H(_, 1)] \circ F'[f] = F'[g] \circ F'[H(_, 0)]$ (**Step 7**).

```
lemma functorOnHomAux_of_homotopic_dimaps {m : ℕ} :
  Π {f g : D(I, X)} {H : DirectedMap.Dihomotopy f g}
  (c : DirectedMap.Dihomotopy.coveredPartwise hX H 0 m),
  Fh_aux (Dipath.of_directedMap f) >>> Fh_aux (H.eval_at_right 1) =
  Fh_aux (H.eval_at_right 0) >>> Fh_aux (Dipath.of_directedMap g) :=
  /- Proof omitted -/
```

By using induction once again, we end up with the lemma showing us that the choice of representative does not matter.

```
variable (γ γ' : Dipath x y)
lemma functorOnHomAux_of_dihomotopic (h : γ.Dihomotopic γ') :
  Fh_aux γ = Fh_aux γ' :=
  /- Proof omitted -/
```

We can now finally define the behaviour on morphisms to obtain a functor by using the universal mapping property of quotients.

```
def FunctorOnHom {x y : dπ_x X} (γ : x → y) : F_obj x → F_obj y :=
  Quotient.liftOn γ Fh_aux
  (functorOnHomAux_of_dihomotopic hX X1_open X2_open h_comm)

def Functor : (dπ_x X) → C where
  obj := F_obj
  map γ := F_hom γ
  map_id x := functorOnHom_id hX X1_open X2_open h_comm x
  map_comp γ1 γ2 := functorOnHom_comp hX X1_open X2_open h_comm γ1 γ2
```

Here `F_hom` is an abbreviation for `FunctorOnHom` and the final `Functor` is abbreviated to `F`. Finally, we get to **Step 8**. The remaining lemmas show that $F \circ \vec{\Pi}(j_k) = F_k$ for $k = 1$ and $k = 2$, and that F is the unique functor with this property.

```
lemma functor_comp_left : (dπ_m j1) >>> F = F1 := /- Proof omitted -/
lemma functor_comp_right : (dπ_m j2) >>> F = F2 := /- Proof omitted -/
lemma functor_uniq (F' : (dπ_x X) → C) (h1 : (dπ_m j1) >>> F' = F1)
  (h2 : (dπ_m j2) >>> F' = F2) : F' = F := /- Proof omitted -/
```

The Van Kampen Theorem is stated as

```
theorem directed_van_kampen (X : IsOpen X1) (X : IsOpen X2)
  (hX : X1 ∪ X2 = Set.univ) :
  IsPushout (dπm i1) (dπm i2) (dπm j1) (dπm j2) :=
  /- Proof omitted -/
```

This theorem now follows easily from the lemmas above.

6 Conclusion and Further Research

In this article, we presented a formalisation of the Van Kampen Theorem in directed topology in the proof assistant Lean 4. This theorem allows one to calculate the fundamental category of a directed space using the fundamental categories of subspaces under a mild condition on the subspaces. At the moment, `mathlib` does not have a version of the Van Kampen Theorem for groupoids, originally proven by Brown in 1968 [2, 3]. The undirected version is a corollary of the directed version because the fundamental groupoid of a topological space can be seen as the fundamental category of a directed space, where all paths are directed. We have not formalised this implication, but it should not be hard to prove the Van Kampen Theorem for groupoids in this manner.

There are generalisations of the undirected version that allow an arbitrary open cover [4, Theorem 2.3.5]. An extension of our formalisation to allow this would be possible using the same general approach, but we have not investigated this in depth.



As a next step, it would be natural to formalise the relation between d-spaces and their homotopy theory with other models of concurrency, such as higher-dimensional automata and their languages, and to develop the homotopy theory of d-spaces further in Lean.

References

- 1 Henning Basold, Peter Bruin, and Dominique Lawson. The Directed Van Kampen theorem in Lean, 2023. Pre-print. [arXiv:2312.06506](https://arxiv.org/abs/2312.06506).
- 2 Ronald Brown. *Elements of Modern Topology*. McGraw-Hill, 1968.
- 3 Ronald Brown. *Topology and Groupoids*. BookSurge Publishing, 2006.
- 4 Ronald Brown, Philip J. Higgins, and Rafael Sivera. *Nonabelian Algebraic Topology: Filtered spaces, crossed complexes, cubical homotopy groupoids*. European Mathematical Society, 2011. doi:10.4171/083.
- 5 Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn, and Jakob von Raumer. The Lean Theorem Prover (System Description). In Amy P. Felty and Aart Middeldorp, editors, *Automated Deduction-CADE-25: 25th International Conference on Automated Deduction*, Lecture Notes in Computer Science, pages 378–388. Springer, Springer International Publishing, 2015. doi:10.1007/978-3-319-21401-6_26.
- 6 Jérémy Dubut. *Directed Homotopy and Homology Theories for Geometric Models of True Concurrency*. PhD thesis, Université Paris-Saclay, 2017. URL: <https://tel.archives-ouvertes.fr/tel-01590515>.
- 7 Lisbeth Fajstrup. Discovering Spaces. *Homology, Homotopy and Applications*, 5(2):1–17, 2003. doi:10.4310/HHA.2003.v5.n2.a1.
- 8 Lisbeth Fajstrup, Eric Goubault, Emmanuel Haucourt, Samuel Mimram, and Martin Raußen. *Directed Algebraic Topology and Concurrency*. Springer, 2016. doi:10.1007/978-3-319-15398-8.
- 9 Lisbeth Fajstrup, Eric Goubault, and Martin Raußen. Detecting Deadlocks in Concurrent Systems. In *CONCUR '98: Concurrency Theory, 9th International Conference, Nice, France, September 8-11, 1998, Proceedings*, pages 332–347, 1998. doi:10.1007/BFb0055632.

- 10 Lisbeth Fajstrup, Martin Raußen, and Eric Goubault. Algebraic topology and concurrency. *Theoretical Computer Science*, 357(1):241–278, 2006. Clifford Lectures and the Mathematical Foundations of Programming Semantics. doi:10.1016/j.tcs.2006.03.022.
- 11 Philippe Gaucher. Six Model Categories for Directed Homotopy. *Categories and General Algebraic Structures with Applications*, 15(1):145–181, 2021. doi:10.52547/cgasa.15.1.145.
- 12 Marco Grandis. Directed homotopy theory, I. The fundamental category. *Cahiers de topologie et géométrie différentielle catégoriques*, 44(4):281–316, 2003. URL: http://archive.numdam.org/item/CTGDC_2003__44_4_281_0/.
- 13 Marco Grandis. *Directed Algebraic Topology: Models of Non-Reversible Worlds*. New Mathematical Monographs. Cambridge University Press, 2009. doi:10.1017/CB09780511657474.
- 14 Kuen-Bang Hou (Favonia) and Michael Shulman. The Seifert-van Kampen theorem in homotopy type theory. In *25th EACSL Annual Conference on Computer Science Logic, CSL 2016 and the 30th Workshop on Computer Science Logic*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, 2016. doi:10.4230/LIPIcs.CSL.2016.22.
- 15 Sanjeevi Krishnan. A Convenient Category of Locally Preordered Spaces. *Applied Categorical Structures*, 17(5):445–466, 2009. doi:10.1007/s10485-008-9140-9.
- 16 Dominique Lawson. GitHub - Dominique-Lawson/Directed-Topology-Lean-4. Software, version 1.1., swHId: swH:1:dir:479a73373a2bf508149f7d1b889b42304fe78a9e (visited on 2024-07-08). URL: <https://github.com/Dominique-Lawson/Directed-Topology-Lean-4/tree/v1.1>.
- 17 Tom Leinster. *Basic Category Theory*. Cambridge University Press, 2014. doi:10.1017/cbo9781107360068.
- 18 The mathlib Community. The Lean Mathematical Library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020*, pages 367–381, New Orleans, LA, 2020. ACM. doi:10.1145/3372885.3373824.
- 19 James R. Munkres. *Topology, a first course*. Prentice-Hall, 1975.
- 20 Vaughan R. Pratt. Modeling Concurrency with Geometry. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages (POPL)*, pages 311–322, 1991. doi:10.1145/99583.99625.
- 21 Floris van Doorn, Jakob von Raumer, and Ulrik Buchholtz. Homotopy type theory in Lean. In *Interactive Theorem Proving: 8th International Conference, ITP 2017, Brasília, Brazil, September 26–29, 2017, Proceedings 8*, pages 479–495. Springer, 2017. doi:10.1007/978-3-319-66107-0_30.
- 22 Rob J. van Glabbeek. On the expressiveness of higher dimensional automata. *Theoretical Computer Science*, 356(3):265–290, 2006. doi:10.1016/j.tcs.2006.02.012.

Verifying Peephole Rewriting in SSA Compiler IRs

Siddharth Bhat  

Cambridge University, UK

Alex Keizer  

Cambridge University, UK

Chris Hughes 

University of Edinburgh, UK

Andrés Goens  

University of Amsterdam, The Netherlands

Tobias Grosser  

Cambridge University, UK

Abstract

There is an increasing need for domain-specific reasoning in modern compilers. This has fueled the use of tailored intermediate representations (IRs) based on static single assignment (SSA), like in the MLIR compiler framework. Interactive theorem provers (ITPs) provide strong guarantees for the end-to-end verification of compilers (e.g., CompCert). However, modern compilers and their IRs evolve at a rate that makes proof engineering alongside them prohibitively expensive. Nevertheless, well-scoped push-button automated verification tools such as the Alive peephole verifier for LLVM-IR gained recognition in domains where SMT solvers offer efficient (semi) decision procedures. In this paper, we aim to combine the convenience of automation with the versatility of ITPs for verifying peephole rewrites across domain-specific IRs. We formalize a core calculus for SSA-based IRs that is generic over the IR and covers so-called regions (nested scoping used by many domain-specific IRs in the MLIR ecosystem). Our mechanization in the Lean proof assistant provides a user-friendly frontend for translating MLIR syntax into our calculus. We provide scaffolding for defining and verifying peephole rewrites, offering tactics to eliminate the abstraction overhead of our SSA calculus. We prove correctness theorems about peephole rewriting, as well as two classical program transformations. To evaluate our framework, we consider three use cases from the MLIR ecosystem that cover different levels of abstractions: (1) bitvector rewrites from LLVM, (2) structured control flow, and (3) fully homomorphic encryption. We envision that our mechanization provides a foundation for formally verified rewrites on new domain-specific IRs.

2012 ACM Subject Classification Software and its engineering → Compilers; Software and its engineering → Semantics; Computing methodologies → Theorem proving algorithms; Theory of computation → Rewrite systems

Keywords and phrases compilers, semantics, mechanization, MLIR, SSA, regions, peephole rewrites

Digital Object Identifier 10.4230/LIPIcs.ITP.2024.9

Supplementary Material *Software*: <https://github.com/opencompl/lean-mlir/tree/ITP24>
archived at `swh:1:dir:037d3d2587a091456ac21509c79a65076ccd348e`

Funding This project has received funding from the European Union’s Horizon EUROPE research and innovation program under grant agreement no. 101070374 (CONVOLVE).

Acknowledgements We thank Sébastien Michelland and Sebastian Ullrich for their early help in this project and feedback, as well as Anton Lorenzen for his helpful feedback.



© Siddharth Bhat, Alex Keizer, Chris Hughes, Andrés Goens, and Tobias Grosser;
licensed under Creative Commons License CC-BY 4.0

15th International Conference on Interactive Theorem Proving (ITP 2024).

Editors: Yves Bertot, Temur Kutsia, and Michael Norrish; Article No. 9; pp. 9:1–9:20

Leibniz International Proceedings in Informatics



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

1 Introduction

Static single assignment (SSA) [34] intermediate representations (IRs) are at the core of modern compilers, thanks to the benefits their immediate encoding of use-def relationships brings to compiler analyses and transformations. *Peephole optimizations* [26], which replace assembly-level instruction sequences of bounded length with semantically equivalent optimized ones, benefit from SSA during target code generation [25] and are now also widely used for optimizing SSA-based IRs. Peephole optimizations are so common, that 10% of all IR transforming code in LLVM [17] belongs to its InstCombine peephole optimizer,¹ which is beyond the size of LLVM’s loop optimizer. Further evidence is offered by LLVM’s commit log, where the most referenced tool is the Alive peephole verifier [23]. Alive has brought automatic SMT-based verification into the day-to-day of the LLVM compiler community.

In the context of the end-to-end verified compiler CompCert [21], peephole rewriting has been formalized (and mechanized) in its classical form of straight-line assembly code [28], but this verification does not cover rewriting along the SSA def-use chain. As an example, consider the rewrite $(y = x + 1; z = y - 1) \mapsto (z = x)$. This pattern does *not* match the program $(y = x + 1; \mathbf{p} = \mathbf{y}; z = y - 1)$ in straight-line rewriting, due to the interleaved instruction $p = y$. On the other hand, by concentrating on the dataflow, we rewrite any subprogram of the form $(y = x + 1; \circ ; z = y - 1)$ to $(y = x + 1; \circ ; z = x)$, regardless of what fills the hole \circ . This rewriting on the “def-use” chain can be applied to assembly code before register allocation and all SSA-based IRs.

SSA-based IRs have been successful in domain-specific compilers, where they enable concise reasoning at the favored abstraction level. In particular, the MLIR compiler framework [18] has been widely adopted for machine learning [39], quantum computing [32], and even for compiling Lean [6]. MLIR lowers the cost of instantiating domain-specific IRs and encourages transformations on specialized high-level IRs. Instead of complex potentially side-effectful global reasoning at a lower abstraction level, these tailored IR abstractions often offer value semantics (i.e., referential transparency) to enable side-effect-free local reasoning. MLIR also introduces the concept of regions, which allow IR operations to be nested, enabling structured control flow. Structured control makes termination proofs of loops easier and the tailored domain-specific IRs have the potential to reduce the complexity of proofs.

In this paper, we verify peephole rewriting over SSA-based IRs. We formalize a core calculus for SSA-based IRs that is generic over the IR and covers regions instead of potentially diverging unstructured control. We mechanize our calculus in the Lean [9] proof assistant and make it accessible to MLIR developers by offering an embedding of MLIR syntax. Concretely, our contributions are:

- A formalization of SSA with regions parametrized over a user-defined IR X and its mechanization in our framework² `LeanMLIR(X)` that exploits denotational-style value semantics for optimizing along the SSA use-def chain of an MLIR-style IR (Sections 2, 3)
- Evidence that our formalization of SSA allows for effective meta-theoretic reasoning:
 - A verified peephole rewriter, for which we prove that lifting a peephole rewrite to a rewrite on the entire program preserves semantics (Subsection 4.1)
 - Two verified implementations of generic SSA-based optimizations: dead code elimination and common subexpression elimination (Subsection 4.2)
 - Proof automation for eliminating the abstraction overhead of our SSA calculus and exposing clean mathematical proof obligations for each rewrite (Subsection 4.3)

¹ Non-blank and non-comment lines of `.cpp` files in `llvm/lib/Transforms` on commit `f4f1cf6c3`.

² Our code is available at <https://github.com/opencomp1/lean-mlir/releases/tag/ITP24> (ae0dd933).

```

inductive Ty
| r
| nat

inductive Op
| arith_const (x : Nat) -- with compile-time data `x`
| monomial -- build equivalence class of monomial
| add -- add op.

```

(a) `QuotRing` has three `Op` constructors, `add`, `monomial`, and `arith_const x` (for `x` an element of \mathbb{N}) matching the three operations of the IR and two `Ty` constructors, `r` and `nat` matching the two IR types.

```

instance : OpSignature Op Ty where signature
| .arith_const _ => { sig := [], outTy := .nat } -- takes no args, returns `nat`
| .add          => { sig := [.r, .r], outTy := .r } -- takes two `r`, returns `r`
| .monomial     => { sig := [.nat, .nat], outTy := .r } -- takes two `nat`, returns `r`

```

(b) User-defined signatures of each `QuotRing` operation.

```

noncomputable def generator : (ZMod q) [X] := X^(2^n) + 1
abbrev R := (ZMod q) [X] / (span {generator q n})

```

```

instance : TyDenote Ty where
toType
| .r => R -- the denotation of `r` is an element of the ring `R`
| .nat => Nat

```

```

instance : OpDenote Op Ty where
denote
| .arith_const (x : Nat), _, _ => x -- the denotation of `arith_const x` is `x`
| .add, [(x : R), (y : R)]_h, _ => x + y
| .monomial, [(c : Nat), (i : Nat)]_h, _ =>
  Quotient.mk (span {generator q n}) (monomial i c)

```

(c) User-defined semantics of `QuotRing`. The `instance` syntax is used to define a typeclass instance, by specifying the corresponding members, which in this case are the denotation functions. The `noncomputable` annotation in Lean tells the compiler not to generate executable code for this function since `mathlib` uses a noncomputable definition for quotients of polynomial rings. Note that our framework ensures that values are well-typed according to `OpSignature` and `TyDenote`.

■ **Figure 1** User definitions for `QuotRing`, which declares the operations and types of the IR, the type signatures of the operations, and the denotations of the types and operations into Lean types.

- An extension of our pure optimizations in a context with side effects (Section 5)
- Syntax, semantics, and local rewrites for three MLIR-based IRs: (1) arithmetic over bitvectors, (2) structured control flow, and (3) fully homomorphic encryption (Section 6)

2 Motivation: Verifying Optimizations for High-Level IRs

Effective domain-specific optimizations are almost impossible when reasoning on traditional LLVM-style compiler IRs. These offer a “universal” low-level abstraction, originally designed to represent C-style imperative code. Such LLVM-style IRs are built around the concepts of load/store/arithmetic/branching, which is ideal when optimizing at the level of scalar arithmetic, instruction scheduling, or applying certain kinds of loop optimizations. However, this level of abstraction is unsuitable for reasoning about high-level mathematical abstractions.

Consider a compiler for Fully Homomorphic Encryption (FHE) [10], a cryptographic technique that uses algebraic structures to allow an untrusted third party to do computation on encrypted data. In such a compiler, we might have a rewrite like $(a + X^{2^n} + 1 \mapsto a)$,

which is a simple identity on the corresponding quotient ring.³ Expressed in LLVM, the computation of this simple operation consists of multiple basic blocks forming a loop, each containing memory loads, pointer arithmetic, scalar operations, and branches. As a result, the algebraic structure is completely lost and exploiting simple algebraic identities turns into a heroic effort of reasoning about side effects and stateful program behavior. State-of-the-art compilers for FHE consequently use domain-specific IRs (often expressed with MLIR [40, 30]) when generating optimized code for FHE, where algebraic optimizations can take place at an FHE-specific IR that has value-semantics (e.g., is referentially transparent) and is overall closer to the mathematical structure of the problem.

2.1 Defining LeanMLIR(QuotRing): Syntax and Semantics

As an example, we model an IR aimed at FHE that manipulates objects in the algebraic structure $R \equiv (\mathbb{Z}/q\mathbb{Z})[X]/(X^{2^n} + 1)$. To model it, we instantiate an IR `LeanMLIR(QuotRing)` in our framework. It has three simple operations: `arith_const` and `monomial`, to construct values in R , and `add` to add two values of R . To define the syntax and semantics of `LeanMLIR(QuotRing)`, we first declare the types and operations in the IR (Figure 1a). `QuotRing` has two types: `r`, which represents the ring R , and `nat` for naturals. Terms in `Op` represent the operations `arith_const`, `monomial`, and `add`, as well as associated compile-time data. We then define the operation signatures by giving an instance of the `OpSignature` typeclass, which is offered by our framework to instantiate custom IRs (Figure 1b). That is, for each operation we specify: (1) the arity and types of arguments `sig`, and (2) the type of the return value `outTy`. The operation `arith_const` takes no arguments and returns a `nat`, `monomial` and `add` take two `nat/r`-valued arguments respectively, and both return an `r`.

The type denotation is also simple to express with the `TyDenote` typeclass (Figure 1c). `Ty` thus represents the embedded type in the IR and has only two inhabitants `r` and `nat`, whose denotation are `R` and `Nat`, the Lean (host) type that represents the mathematical objects R and \mathbb{N} respectively. The denotation of operations is a Lean function from the denotation of the input types (as recorded in the signature of that operation), to the denotation of the output type.⁴ Concretely, an `arith_const n` operation takes no arguments, so its denotation is Lean’s `Nat`, while `add` takes two `r` arguments, so its denotation is a function from the product⁵ of its arguments to its output, i.e., $R \times R \rightarrow R$. The same is true for `monomial` for $\text{Nat} \times \text{Nat} \rightarrow R$. We define the denotation of `arith_const n` to evaluate to `n`, `add(x, y)` to evaluate to `x + y`, and `monomial(a, i)` to evaluate to `Quotient.mk (span generator p q (monomial a i))`, the equivalence class of the monomial aX^i . As the `QuotRing` IR does not require regions (Subsection 6.2), we only need to add MLIR syntax support (Subsection 3.2) and translate the MLIR AST to `Ty` and `Op`, e.g., mapping `!Nat`⁶ to `nat` and `!R` to `r`, and a full `QuotRing` example (Figure 2) can be written in Lean.

2.2 Defining and Executing Peephole Rewrites for QuotRing

We now verify the peephole rewrite $(a + X^{2^n} + 1 \mapsto a)$, where a is a variable and X^{2^n} is a constant in the ring. In $(\mathbb{Z}/q\mathbb{Z})[X]/(X^{2^n} + 1)$ this rewrite is simple to prove and, unsurprisingly, our custom `LeanMLIR(QuotRing)` IR enables us to rewrite at exactly this

³ We will discuss the underlying mathematical structure in more detail in Subsection 6.3

⁴ Our framework groups type and operation denotations into a `Dialect`, which we leave out for brevity.

⁵ The mechanization uses a heterogeneous vector type `HVector`, which is coerced into the product type.

⁶ In practice, one would use a fixed-bitwidth type `iN` but we use `!Nat` for a simpler exposition.

```

def a_plus_generator_eq_a : PeepholeRewrite Op [.r] .r := {
  lhs /- a + X^(2^n) + 1 -/ := [quotring_com q, n] {
    ^bb0(%a : !R):
      %one_int = arith.const 1 : !Nat
      %two_to_the_n = arith.const ${2**n} : !Nat
      %x2n = poly.monomial %one_int, %two_to_the_n : (!Nat, !Nat) -> !R
      %oner = poly.const 1 : !R
      %p = poly.add %x2n, %oner : !R
      %v1 = poly.add %a, %p : !R
      return %v1 : !R
  },
  rhs /- a -/ := [quotring_com q, n] {
    ^bb0(%a : !R):
      return %a : !R
  },
  correct := by
    funext Γv; simp_peephole [Nat.cast_one, Int.cast_one] at Γv 1
    /- ⊢ a + ((Quotient.mk (span {f q n})) ((monomial (2**n)) 1) + 1) = a -/
    ... /- simple proof with only definitions and theorems from Mathlib -/
}

```

■ **Figure 2** A peephole rewrite in `LeanMLIR(QuotRing)` asserts the semantic equivalence of two SSA programs given in MLIR syntax. Our proof automation through `simp_peephole` eliminates the framework overhead, such that closing a clean mathematical goal suffices to prove correctness.

level. Any given peephole rewrite (of which Figure 2 is an example) consists of a context Γ of free variables in the search pattern of the peephole rewrite. The search pattern is called `lhs`, and the replacement is `rhs`. The user has a proof obligation that the denotations of the left and right-hand sides are equal, which is given by the field `correct` of the peephole rewrite. In later examples, we reason upto semantic refinement to incorporate LLVM’s notion of poison values [20]. For now, we stick to equality to simplify exposition.

We declare our desired peephole rewrite by defining `a_plus_generator_eq_a`. Its type is `PeepholeRewrite Op [.r] r`, where the `Op` specifies the IR the rewrite belongs to and `[.r]` is the list of types of free variables in the program. For $(a + X^{2^n} + 1 \mapsto a)$, this is $(a : r)$. The final instruction we are matching yields a value of type `r`. The `lhs` is the program fragment we want to match on, with the free variable `%a` interpreted as being allowed to match any variable of type `r`. Observe that the type encapsulates exactly what is necessary for a well-typed match: the types of free variables `r` and the type of the instruction whose return value we are replacing (also `r` in this case). The rewritten program is the `rhs` field.

Both the left- and right-hand sides of the rewrite are written in MLIR syntax. Note that we also include a custom quasiquotation `${2**n}` , to inline the symbolic (universally quantified) value n , even though the IR would require 2^n to be a concrete constant. Using MLIR syntax matches the LLVM community’s use of automation tooling, such as Alive: copy a code snippet and get a response. Our goal is to make the use of an interactive theorem prover part of the day-to-day workflow of compiler engineers. To enable this workflow, we implement a full MLIR syntax parser, along with facilities to convert from the generic MLIR abstract syntax into our framework type, such that we can use MLIR syntax in Lean.

To prove the correctness of `a_plus_generator_eq_a`, we use the `simp_peephole 1` tactic from our framework, which removes any overhead of our SSA encoding. We are

left with: `⊢ a + ((Quotient.mk (span f q n)) ((monomial (2**n)) 1) + 1) = a`, a proof obligation in the underlying algebraic structure that, thanks to Lean’s `mathlib`, can be closed with a few (elided) lines of algebraic reasoning.

2.3 Executing Peephole Rewrites

Given a peephole rewrite `rw` and a source program `s`, we provide `rewritePeephole` to replace the pattern `rw.lhs` in the source program `s`. If the matching succeeds, we insert the target program `rw.rhs` (with appropriate substitutions) and replace all references to the original variable with a reference to the newly inserted `var`. Note that the matching is based on the def-use chain. Thus, a pattern need not be *syntactically* sequential in the program `s`. As long as the pattern `rw.lhs` can be found as a *subprogram* of `s`, `s` will be rewritten. This makes our peephole rewriter an SSA peephole rewriter, which distinguishes it from a straight-line peephole rewriter that only matches a linear sequence of instructions.

Thanks to our intrinsically well-typed encoding, we know that the result of the rewriter is always a well-typed program, under the same context and resulting in the same type as the original program. Furthermore, the framework extends the local proof of semantic equivalence to a global proof, showing that the rewriter is semantics preserving:

```
/- The denotation of the rewritten program is equal to the source program. -/
theorem denote_rewritePeephole (fuel : ℕ) (rw : PeepholeRewrite Op Γ t)
  (target : Com Op Γ₂ t₂) : (rewritePeephole fuel rw target).denote = target.denote
```

These typeclass definitions are all we need to define the `QuotRing` IR. Our framework takes care of building the intrinsically well-typed IR for `QuotRing` from this, and gives us a verified peephole rewriter, with other optimizations like CSE and DCE. We will now delve into the details of the framework and see how it achieves this.

3 LeanMLIR(X): A Framework for Intrinsically Well-Typed SSA

In this section, we describe the core design of the framework: the encoding of programs and their semantics in `LeanMLIR(X)` (Figure 3a). We review some dependently-typed tooling we use to define our IR. **Contexts:** Our encoding is intrinsically well-typed (i.e., each inhabitant of `Expr` or `Com` described below is, by construction, well typed). Thus, we need a *context* to track the types of variables that are allowed to occur (`Ctxt Ty`). A context is a list of types, where for example `[int, int, bool]` means that there are two variables of the (user-defined) type `int` and one variable of type `bool` we may refer to. **Variables:** The type `(Var Γ α)` encodes variables of type `α` in context `Γ`. We use De Bruijn indices [33] in the standard way, but, additionally, a variable with index `i` also carries a proof witness that the `i`-th entry of context `Γ` is the type `α`. **Heterogeneous Vectors:** To define an argument signature (`OpSignature.sig`), say, `[int, int, bool]`, we need an expression with this operation to store two variables of type `int` and one of type `bool`. We want to statically ensure that the types of these variables are correct, so we store them in a heterogeneous vector. A vector of type `HVector f [α₁, ..., αₙ]` is equivalent to a tuple `(f α₁ × ... × f αₙ)`.

3.1 Semantics of LeanMLIR(X)

The core types for programs are `Expr` and `Com`, shown in Figure 3a. The type `Expr Γ α` describes individual SSA operations; we think of it as a function from values in the context `Γ` – also called a *valuation* for that context – to a value in the denotation of type `α`. The type `Com Γ α` has a similar interpretation but represents sequences of operations. Each command binds a new value in the current context (the `var` constructor) until the sequence


```

inductive Expr [OpSignature Op Ty] : Ctxt Ty → Ty → Type where
| mk (op : Op) -- op (arg1, arg2, ..., argn) : outTy op
  (args : HVector (Var Γ) (OpSignature.sig op)) : Expr Γ (OpSignature.outTy op)

inductive Com [OpSignature Op Ty] : Ctxt Ty → Ty → Type where
| ret (v : Var Γ α) : Com Γ α -- return v
| var (e : Expr Γ α) (body : Com (Γ.snoc α) β) : Com Γ β -- let v : α := e in body

```

(a) Core syntax of $\text{LeanMLIR}(X)$, polymorphic over Op . The arguments in square brackets are assumed typeclass instances. Type is the base universe of Lean types.

```
variable [TyDenote Ty] [OpDenote Op Ty] [DecidableEq Ty]
```

```
def Expr.denote : {ty : Ty} → (e : Expr Op Γ ty) → (Γv : Valuation Γ) → toType ty
| _, ⟨op, args⟩, Γv => OpDenote.denote op (args.map (fun _ v => Γv v))
```

```
def Com.denote : Com Op Γ ty → (Γv : Valuation Γ) → (toType ty)
| .ret e, Γv => Γv e
| .var e body, Γv => body.denote (Γv.snoc (e.denote Γv))
```

(b) Denotation of Expr and Com in $\text{LeanMLIR}(X)$, which extends the user's OpDenote to entire programs. Intrinsic well-typing of Com makes its denotation a well-typed function from the context valuation to the return type. The angled brackets are used to pattern match on a structure constructor anonymously.

■ **Figure 3** Definitions in $\text{LeanMLIR}(X)$ for Expr and Com , and their associated denotations.

returns the value of one such variable v (the ret constructor). Thus, this encoding of SSA exploits the similarity to the ANF [2] and CPS [15] encodings. In particular, our Expr represents an SSA assignment, and Coms represents a block of operations, often called a basic block. A basic block typically would either return or branch to another block. In our case, blocks only return and consequently do not model branching. Instead, we use regions to model structured control flow (Subsection 6.2). Given our core syntax, our framework now automatically expands the semantics given by the user in OpDenote to semantics for Expr and Com (Figure 3b). An Expr evaluates its arguments by looking up their value in the valuation and then invokes the user-defined OpDenote.denote to evaluate the semantics of the op .

3.2 Writing $\text{LeanMLIR}(X)$ Programs Using MLIR Syntax

An important goal for our framework is to provide easy access to formalization for the MLIR community. Toward this goal, we have a deep embedding of MLIR's AST and a corresponding parser. This is developed using Lean's syntax extensions [38]. We extend Lean with a generic framework to build Expr and Com terms from a raw MLIR AST. This framework allows the user to pattern-match on the MLIR AST to build intrinsically well-typed terms, as well as to throw errors on syntactically correct, but malformed MLIR input. These are used by our framework to automatically convert MLIR syntax into our SSA encoding, along with the ability to provide precise error messages in cases of translation failure. This enables us to write all our examples in MLIR syntax, as demonstrated throughout the paper.

More concretely, we have an embedded domain-specific language (EDSL), which declares the MLIR grammar as a Lean syntax extension. As part of this work, we have found several inconsistencies with the MLIR language reference and contributed patches upstream to

9:8 Verifying Peephole Rewriting in SSA Compiler IRs

```
structure OpSignature (Tv : Type) where /- (1) Extending signature. -/  
  regSig : List (Ctx Ty × Ty)  
  ...  
  
class OpDenote [TyDenote Ty] [OpSignature Op Ty] where /- (2) Extending denotation. -/  
  denote : (op : Op) → (args : HVector toType (OpSignature.sig op)) →  
    (regArgs : HVector (fun (ctx, t) => Valuation ctx → toType t) (OpSignature.regSig op)) →  
    (toType (OpSignature.outTy op))  
  
inductive Expr : (Γ : Ctxt Ty) → (ty : Ty) → Type where  
  | mk (op : Op)  
  ...  
  (regArgs : HVector (fun (ctx, ty) => Com ctx ty) (OpSignature.regSig op)) :  
  Expr Γ ty  
  
mutual /- (3) extending expression denotation to recursively invoke regions. -/  
  def Expr.denote : {ty : Ty} → (e : Expr Op Γ ty) → (Γv : Γ.Valuation) → (toType ty)  
  | _, ⟨op, args, regArgs⟩, Γv =>  
  OpDenote.denote op (args.map (fun ty v => Γv v)) regArgs.denote  
  ...  
end
```

■ **Figure 4** Extending $\text{LeanMLIR}(X)$ with regions. New fields are in green. In `OpDenote`, one can now access the sub-computation represented by the region when defining the semantics of `Op`.

update them.⁷ Overall, this gives users the ability to write idiomatic MLIR code into our framework and receive an MLIR AST. Moreover, as we will showcase in the examples, our EDSL is idiomatically embedded into Lean, which allows us to quasiquote Lean terms. This will come in handy to write programs that are generic over constants, such as parameterizing a program by 2^n for any choice of n . We build our intrinsically well-typed data structures from this MLIR AST by writing custom elaborators.

3.3 Modelling Control Flow in $\text{LeanMLIR}(X)$ With Regions

So far, our definition of `Com` only allows straight-line programs. To be able to model control flow, we add regions to our IR. Regions are an extension to SSA introduced by MLIR. They add the syntactic ability to nest IR definitions, thereby allowing syntactic encoding of concepts such as structured control flow. This is in contrast with the approach of having a sea of basic blocks in a control-flow graph (CFG) that are connected by branch instructions. More specifically, structured control flow with regions allows modeling reducible control flow [1]. General CFGs allow us to represent more complex, irreducible control flow, which makes them harder to reason about. Consequently, compiler frameworks such as MLIR encourage structured control flow (even though they allow for a sea of basic blocks). In our framework, we focus on the novel aspects of MLIR: structured control via nested regions.

Intuitively, regions allow an `Op` to receive `Com`s as arguments, and choose to execute these `Com` arguments zero, one, or multiple times. This allows us to model if conditions (by executing the regions zero or once), loops (by executing the region n times), and complex operations such as tensor contractions and convolutions by executing the region on the elements of the tensor [39]. We implement this by extending `Expr` with a new field representing region arguments (Figure 4). We also extend `OpSignature` with an extra argument for the input types and output types of the region. In parallel, we add the denotation of regions as an

⁷ reviews.llvm.org/{D122979, D122978, D122977, D119950, D117668}

argument, extending `OpDenote`. Similarly, we extend the denotation of `Expr` to compute the denotation of the region `Coms` in the `Expr`, before handing off to `OpDenote`.

This extension to our core calculus gives us the ability to model structured nesting of programs whose denotation is a bounded computation.⁸ This is used pervasively in MLIR, to represent `if` conditions, `for` loops, and higher-level looping patterns such as multidimensional strided array accesses over multidimensional arrays (tensors). We show how to model control flow in Subsection 6.2.

4 Reasoning About `LeanMLIR(X)`

The correctness of peephole rewriting is a key aspect of the metatheory of `LeanMLIR(X)`. We begin by sketching the mechanized proof of correctness of peephole rewriting. We then discuss how the infrastructure built for this proof is reused to prove two other SSA optimizations: common subexpression elimination (CSE) and dead code elimination (DCE). Finally, we discuss our proof automation, which manipulates the IR encoding at elaboration time to eliminate all references to the framework and provide a clean goal to the proof engineer.

4.1 Verified SSA Rewriting With `rewritePeephole`

We now provide a sketch of the mechanized correctness proof of `rewritePeephole`. The key idea is that to apply a rewrite at location i , we open up the `Com` at location i in terms of a zipper [12]. This zipping and rewriting at a location i is implemented by `rewritePeepholeAt`. The zipper comprises of `Lets` to the left-hand side of i , and `Com` to the right: `let x2 = x1; (let x3 = x2; (let x4 = x3; (return x3))) : Com [x1] α =`

```
((let x2 = x1); let x3 = x2); : Lets [x1] [x1, x2, x3]
```

```
(let x4 = x3; (return x3)) : Com [x1, x2, x3] α
```

The use of a zipper enables us to easily traverse the sequence of `let`-bindings during transformation and exposes the current `let` binding being analyzed. This exposing is performed by `Lets`, which unzips a `Com` such that the outermost binding of a `Lets` is the innermost binding of a `Com`. This forms the zipper, which splices the `Com` into a `Com` and a `Lets`. Also, while `Com` tracks only the return type α in the type index, `Lets` tracks the entire resulting context Δ . That is, in `(lets : Lets Γ Δ)`, the first context, Γ , lists all free variables (just as in `Com Γ τ`), but the second context, Δ , consists of all variables in Γ plus a new variable for each `let`-binding in the sequence `lets`. We can thus think of Δ as the context at the current position of the zipper. Another difference is the order in which these sequences grow. Recall that in `Com`, the outermost constructor represents the topmost `let`-binding. In `Lets`, the outermost constructor instead corresponds to the bottom-most `let`-binding. This difference is what makes the zipper work.

We have two functions to go from a program to a zipper and back: (1) `(splitProgramAt pos prog)`, to create a zipper from a program `prog` by moving the specified number of bindings to a new `Lets` sequence, and (2) `(addComInMiddleOfLetCom top mid bot)`, to turn a zipper `top`, `bot` into the program, while inserting a program `mid : Com` in between. We also prove that the result of splitting a program with `splitProgramAt` is semantically

⁸ Since our semantics denote into Lean expressions, the user-given semantics must be provably terminating to be executable. We wish to explore richer denotations, such as (computable) coinductive and (noncomputable) domain theoretic semantics in future work.

equivalent to the original program. Similarly, we prove that stitching a zipper back together with `addComInMiddleOfLetCom` results in a semantically equivalent program.

Given a peephole rewrite (`matchCom`, `rewriteCom`), to rewrite at location i , we first split the target program into `top` and `bot`. We then attempt to match the def-use chain of the return variable in `matchCom` with the final variable in `top` (which is the target i , since we split the program there). This matching of variables recursively matches the entire expression tree.⁹ Upon successful matching, this returns a substitution σ for the free variables in `matchCom` in terms of (free or bound) variables of `top`. Using this successful matching, we stitch the program together as `top; σ (rewriteCom); τ (bot)`. Here, τ is another substitution that replaces the variable at location i with the return variable of `rewriteCom`. Since we derived a successful matching, we know that the semantics of variable i is equal to that of the return variable of `matchCom`. By assumption on the peephole rewrite, the variable i is equivalent to the return variable of `rewriteCom`. This makes it safe to replace all occurrences of the variable i in `bot` with the return variable of `rewriteCom`. This proves `denote_rewritePeephole`, which states that if a rewrite succeeds, then the semantics of the program remain unchanged. In this way, we use a zipper as a key inductive reasoning principle to mechanize the proof of correctness of SSA-based peephole rewriting. We extend this rewriting to regions by recursively rewriting over the regions in a program.

4.2 DCE & CSE: Folding Over Intrinsically Well Typed SSA

The classic optimizations enabled by SSA are peephole rewriting, dead code elimination (DCE), and common subexpression elimination (CSE). We implement these optimizations in our framework as a test of its suitability for metatheoretic reasoning. Our approach is different from previous approaches [47, 5] with our use of intrinsic well-typing, which mandates proofs of the structural rules on contexts to rewrite programs. We begin by building machinery to witness that a context Δ is equal to the context Γ , minus the variable x . This is spelled as `Deleted Γ x Δ` in `LeanMLIR(X)`. We then prove context-strengthening theorems to delete variables that do not occur in `Expr` and `Com` while preserving denotation.

Using this tooling, DCE is implemented in ≈ 400 LoC, which shows that our framework is well-suited to metatheoretic reasoning. The implementation is written in a proof-carrying style, interleaving function definitions with their proof of correctness. The recursive step of the dead code elimination takes a program $p : \text{Com } \Gamma \ \mathfrak{t}$ and a variable v to be deleted, and returns a new $p' : \text{Com } \Delta \ \mathfrak{t}$. The two contexts Γ and Δ are linked by a context morphism (`Hom Γ Δ`), to interpret p' (with the deleted variable) which lives in a strengthened context Δ in the old context Γ . We walk p recursively to eliminate dead values at each `let` binding. This produces a new p' with dead bindings removed, a proof of semantic preservation, and a context morphism from the context of p to the strengthened context of p' with all dead variables removed.

Similarly, the CSE implementation folds over `Com` recursively, maintaining data structures necessary to map variables and expressions to their canonical form. At each (`let $x = f(v_1, \dots, v_n)$ in b`) step, we canonicalize the variables v_i to find variables c_i . We then look up the canonicalized expression $f(c_1, \dots, c_n)$ in our data structure to find the canonical variable c_x if it exists and replace x with c_x . If such a canonical c_x does not exist, we add a new entry mapping $f(c_1, \dots, c_n)$ to x , thereby canonicalizing any further uses of this expression.

⁹ We match regions in expressions for structural equality. We *do not* recurse into regions during matching, and treat regions as black-boxes.

4.3 Proof Automation for Goal State Simplification in $\text{LeanMLIR}(X)$

The proof automation tactic `simp_peephole` Γ (used to eliminate framework definitions from the goal state) takes a context Γ , reduces its type completely, and abstracts out program variables to provide a theorem statement that is universally quantified over the variables of the program, with all framework definitions eliminated. It uses a set of equation theorems to normalize the type of Γ . This is necessary to extract the types of variables during metaprogramming. Once the type of Γ is known, we simplify away all framework definitions (such as `Expr.denote`). We then replace all occurrences of a variable accesses $\Gamma[i]$ with a new (Lean, i.e., host) variable. We do this by abstracting terms of the form $\Gamma[i]$ where i is the i -th variable. This gives us a proof state that is universally quantified over variables from the context. Finally, we clear the context away to eliminate all references to the context Γ . The set of definitions we simplify away is extensible, enabling us to add domain-specific simplification rewrites for the IR.

5 Pure Rewriting in a Side-Effectful World

While $\text{LeanMLIR}(X)$ streamlines the verification of higher-level IRs that use only value semantics, typical IRs may interleave islands of pure operations (with value semantics) with operations that carry side effects. An IR that is user-facing can usually be rephrased with high-level, side-effect-free semantics. Yet, each operation in such an IR is compiled through a sequence of IRs that are lower level and potentially side-effectful. For example, in the case of FHE, the pure FHE IR is compiled to a lower-level IR that encodes the coset representative of each ideal as an array, with control flow represented via structured control flow (`scf`). Eventually, this is compiled into LLVM which is rife with mutation and global state. In such a compilation flow, peephole rewrites are used at each intermediate IR to optimize pure fragments while leaving side-effectful fragments untouched. An effective compiler pipeline introduces the right abstractions to maximize rewrites on side effect-free fragments.

$\text{LeanMLIR}(X)$ is designed to facilitate verification of peephole rewrites as they arise in such a compiler pipeline. The previous sections already presented how our framework supports the verification of peephole rewrites in a pure setting. Yet, our design also allows for the optimization of a pure fragment in a side-effectful context. We have a mechanized proof of the correctness of the extended framework with support for side effects and a rewrite theorem that performs pure rewrites in the presence of side effects. The key idea is to annotate each `Op` with an `EffectKind`, where `EffectKind.pure` changes the denotation of the `Expr` into the `Id` monad, while `EffectKind.impure` denotes into an arbitrary, user-chosen, IR-specific monad. We also introduce a new notion of monadic evaluation of `Lets`, which returns a valuation plus a proof that, for every variable v that represents a pure expression e in the sequence of let-bindings, the valuation applied to v agrees with the (pure) denotation of e . This proof-carrying definition allows us to use this invariant when reasoning inside a subexpression of a monadic bind.

With the above at hand, the overall rewriter construction and proof strategy remains unchanged, with the additional constraint of performing rewriting only on those operations marked as `EffectKind.pure`, and the surrounding monadic ceremony required to show that a pure rewrite indeed does not change the state of pure variables in various lemmas.¹⁰

¹⁰ A limitation of our current mechanization is that we assume that all regions are potentially side-effecting. This is a simplification that shall be addressed in a newer version of the proof.

6 Case Studies

We mechanize three IRs based on ones found in the MLIR ecosystem as case studies for `LeanMLIR(X)` and show how they benefit from the different aspects of our framework. Note that the core of our framework (definitions of `Expr`, `Com`, `PeepholeRewrite`, lemmas about these objects, and the peephole rewriting theorem) is $\approx 2.2k$ LoC. The case studies based on our framework together are $\approx 5.6k$ LoC, which stresses the framework to ensure that it scales to realistic formal verification examples.

6.1 Reasoning About Bitvectors of Arbitrary Width

We first demonstrate our ability to reason about a well-established domain of peephole rewrites: LLVM’s arithmetic operations over fixed-bitwidth integers. Using the Z3 SMT solver [8], the Alive project [24, 23] can efficiently and automatically reason about these. Notably, at the time of this writing, almost 700 LLVM patches have justified their correctness by referencing Alive. In this way, accessible proof tools can find a place in production compiler development workflows. However, Alive is limited by the capabilities of the underlying SMT solvers. SMT solvers are complex, heuristic-driven, and sometimes even have soundness bugs [43]. They are also specialized to support very concrete theories. Among others, this means Alive can only reason about a given fixed bitwidth. Even recent work that specifically aims to generalize rewrites to arbitrary bitwidths, can only exhaustively test a concrete set of bitwidths [27]. Using our framework, we can reproduce Alive-style correctness proofs, and extend them to reason about arbitrary (universally quantified) bitwidths. This ability to handle arbitrary bitwidth is important in verification contexts that have wide bitvectors, as they can occur in real-life VLSI problems [13, 41]. MLIR itself has multiple IRs that require bitvector reasoning: `comb` for combinational logic in circuits, `arith` and `index` for integer and pointer manipulation, and `llvm` which embeds LLVM IR in MLIR. Our streamlined verification experience offers developers an Alive-style workflow for the `llvm` dialect, while allowing reasoning across bitwidths. As our framework is extensible, we believe we can also support other dialects that require bitvector reasoning, such as `comb`, `arith`, and `index`.

6.1.1 Modeling a fragment of LLVM IR: Syntax and Semantics

To test our ability to reason about bitvectors in practice, we model the semantics of the arithmetic fragment of LLVM as the IR `LeanMLIR(LLVM)`. We support the (scalar) operators: `not`, `and`, `or`, `xor`, `shl`, `lshr`, `ashr`, `urem`, `srem`, `add`, `mul`, `sub`, `sdiv`, `udiv`, `select` and `icmp`. We support all `icmp` comparison flags, but not the strictness flags `nsw` and `nw`.

At the foundation of our denotational semantics is Lean’s `BitVec` type, which models bitvectors of arbitrary width and offers `smtlib` [4] compatible semantics. However, when we started this work, most bitvector operations were not defined in the Lean ecosystem and the bitvector type itself was not fully fleshed out. Hence, we worked with the `mathlib` and Lean community to build and upstream a theory of bitvectors.¹¹ After developing the core theory in `mathlib`, Lean’s mathematical library, development subsequently moved into Lean core, where we continue to evolve Lean’s bitvector support.

¹¹ github.com/leanprover-community/mathlib4/pull/{5383,5390,5400,5421,5558,5687,5838,5896,7410,7451,8231,8241,8301,8306,8328,8345,8353},
github.com/leanprover/lean4/pull/{3487,3471,3461,3457,3445,3492,3480,3450,3436},
github.com/leanprover/std4/pull/{357,359,599,626,633-636,637,639,641,645-648,655,658-660,653}

The semantics of LLVM’s arithmetic operations follow the semantics of `smtlib` (and consequently Lean’s) bitvectors closely. In case of integer wrapping or large shifts, for example, LLVM can produce so-called poison values [24], which capture undefined behavior as a special value adjoined to the bitvector domain. LLVM’s poison is designed not to be a side effect and, consequently, can be reasoned about in a pure setting. In contrast, `ub` is a side effect that triggers immediate undefined behaviour, and can be refined into any behavior. In LLVM, the following refinements are legal: `ub` \sqsubseteq `poison` \sqsubseteq `val`. Among the instructions we model, division and remainder can produce immediate undefined behavior `ub`. In our framework, we approximate these by collapsing the side-effectful undefined behavior and side-effect-free poison both into `Option.none`. We thus denote bitvectors into the type `Option (BitVec w)`. This is safe as long as the right hand side is allowed to produce `ub` only when the left hand side produces `ub`. In our context, only the division and remainder operations produce `ub`. In all the Alive rewrites we translate that contain division and remainder operations, we manually verify that the right hand side of a rewrite triggers `ub` if and only if the left hand side does (by checking that any division/remainder on the right has a corresponding operation with syntactically equal arguments on the left). To be fully correct one can either treat division and remainder as side-effectful operations in our framework or develop further theory with respect to treating `ub` as a side effect. We leave separating `ub` as a side effect distinct from poison, and reasoning about peephole rewrites which refine such side effects as interesting future work.

For side-effect-free programs, our semantics match the LLVM semantics. We perform exhaustive enumeration tests between our semantics and that of LLVM. We take advantage of the fact that an IR with computable semantics automatically defines an interpreter in our framework. We build an executable program that runs every instruction, with all possible input combinations upto bitwidth 8. We get LLVM’s ground truth by using LLVM’s optimizer, `opt` to transform the same instruction with constant inputs. This optimizes the program into a constant output, handling undefined behavior. By exhaustive enumeration, our tested executable semantics correspond to the LLVM semantics wherever the result is `Option.some`, and also soundly model undefined behavior whenever the result is `Option.none`. This gives us confidence our semantics correspond to LLVM’s.

6.1.2 Proving Bitvector Rewrites in our Framework

Effective automation for bitvector reasoning is necessary to resolve the proof obligations that `LeanMLIR(X)` derives automatically from peephole rewrites expressed as MLIR program snippets. While Lean does not yet have extensive automation for bitvectors, thanks to our work we can already use a decision procedure for commutative rings [11] and an extensionality lemma that establishes the equality of bitvectors given equality on an arbitrary bit index.

We test the available automation on a dataset of peephole optimizations from Alive’s test suite, consisting of theorems about addition, multiplication, division, bit-shifting and conditionals. Out of the 435 tests in Alive’s test suite, we translate 93 tests which are the ones that are supported by the LLVM fragment we model and without preconditions. We prove 54 of these rewrites from the Alive test suite automatically. Some rewrites cannot be handled automatically. Of those where automation struggles, we manually prove an additional 6, selecting the ones where an SMT solver takes long to prove them even for a specific bitwidth (e.g., 64). Our proofs are over arbitrary (universally quantified) bitwidth, save for some theorems that are only true at particular bitwidths.¹² As an example, let us consider the following rewrites:

¹²e.g., `a + b = a xor b` is true only at bitwidth 1.

```

example (w : Nat) :
  [llvm (w) | {
    ^bb0(%X : _, %Y : _):
      %v1 = llvm.sub %X, %X
      %r = llvm.xor %v1, %Y
      llvm.return %r
  }] ⊆ [llvm (w) | {
    ^bb0(%X : _, %Y : _):
      llvm.return %Y
  }] := by
  simp_alive_peephole
  alive_auto

example (w : Nat) :
  [llvm (w) | {
    ^bb0(%X : _, %Y : _):
      %v1 = llvm.and %X, %Y
      %v2 = llvm.or %X, %Y
      %v3 = llvm.add %v1, %v2
      llvm.return %v3
  }] ⊆ [llvm (w) | {
    ^bb0(%X : _, %Y : _):
      %v3 = llvm.add %X, %Y
      llvm.return %v3
  }] := by
  simp_alive_peephole
  <proof omitted>

```

Note that due to the support of MLIR syntax in our framework, these rewrites are specified in MLIR syntax. We use a custom extension with the placeholder syntax `_`, to stand for an arbitrary bitwidth w . Simplification of the framework code with `simp_peephole`, yields the proof obligation $(w : \text{Nat}) (X Y : \text{BitVec } w) \vdash \text{LLVM.xor } (\text{LLVM.sub } X X) Y \sqsubseteq Y$ for the first example. This proof obligation only concerns the semantics in the semantic domain of bitvectors, it does not feature MLIR and SSA anymore. This goal is automatically proven by our proof automation for bitvectors, `alive_auto`. The proof for the second example (omitted) is slightly more involved and currently requires manual intervention. It yields the proof state: $\vdash (B \&\&\& A) + (B ||| A) = B + A$, where the proof follows by reasoning about the addition as a state machine. In the longer term, we aim to also connect our work to a verified SAT checker that is under development.¹³

6.2 Structured Control Flow

The examples of IRs we have seen so far are all straight-line code. In this use case, we show how we can add control flow to existing IRs, thanks to the parametricity of our framework. We also demonstrate how encoding control flow structures as regions enable succinct proofs for transformations, by exploiting the high-level structure of these operations. To this end, we model structured control flow as a fragment of the `scf` IR in MLIR, by giving semantics to two common kinds of control flow: `if` conditions and bounded `for` loops. Note that we choose to model *bounded* for loops, since these are the loops that are used in MLIR to model high-level operations such as tensor contractions. A pleasant upshot is that these are guaranteed to terminate, and can thus have a denotation as a Lean function without requiring modelling of nontermination (which is side-effectful). Our sketch of the extended framework with side effects will be used to pursue this line of research in the future. The conditionals and bounded for loops allow us to concisely express loop canonicalizations and transformations from MLIR in `LeanMLIR(scf)`.

We built this parametrically over an existing IR X to allow these constructs to be added to an existing IR X . The key idea is that the `Op` corresponding to `scf` is parametrized by the `Op` corresponding to another IR X . Since the only datatypes `scf` requires are booleans and natural numbers, we ask that the type domain of X contains these types. We then provide denotations in `LeanMLIR(scf(X))` for booleans and integers from the type domain of X . Thus, what we encode is `LeanMLIR(scf(X))`, which is an IR for structured control flow parametrized by another, user-defined IR X .

The `scf.for` operation (Figure 5) has three arguments: the number of times the loop is to be executed, a starting and step value for the iteration, and a seed value for the loop to

¹³<https://github.com/leanprover/leansat>


```

/-- only control flow operations, parametric over another IR Op' -/
inductive Op (Op' : Type) [OpDenote Op' Ty'] : Type
| coe (o : Op') -- coerce Op' to Op
| for (ty : Ty') -- a for loop whose loop carried data is Ty'

instance [I : HasTy Op' Int] : OpSignature (Op Op') Ty' where
  signature
  | .coe o => signature o
  | .for t => <[/-start-/I.ty, /-step-/I.ty, /-niters-/N.ty, /-v-/t],
    /- region arguments: -/ [(/-i-/I.ty, /-v-/t), /-v'-/t)],
    /-return-/t)
instance [I : HasTy Op' Int] [OpDenote Op' Ty'] : OpDenote (Op Op') Ty' where
  denote
  | .coe o', args', regArgs' => OpDenote.denote o' args' regArgs' -- reuse denotation of o'
  | .for ty, [istart, istep, niter, vstart]h, [f]h =>
    let istart : ℤ := I.denote_eq ▶ istart -- coerce to `int`.
    ... -- coerce other arguments
    let loop_fn := ... -- build up the function that's iterated.
    (loop_fn (istart, vstart)).2

```

■ **Figure 5** Simplified implementation of $\text{LeanMLIR}(\text{scf}(X))$ Observe that the IR is parametrized over another IR Op' , and that we add control flow to the other IR in a modular fashion.

iterate on. Note that in the definition, the IR Op is defined parametrically over another IR Op' , and the types of Op are the same as the types of other IR Ty' . We perform a similar construction for `if` conditions.

The denotation of the `for` loop, as well as theorems about loop transformations, follow from `mathlib`'s theory for iterating functions, `Nat.iterate`. The loop body in `scf.for` has a region that receives the current value of the loop counter and the current iterated value and returns the next iterated value. We prove the inductive invariant for loops using the standard theory of iterated function compositions ($f^0 = \text{id}$, $f^k \circ f^l = f^{k+l}$, $\text{id}^k = \text{id}$). We also prove common rewrites over loops: running a `for` loop for zero iterations is the same as not running a loop at all (dead loop deletion), two adjacent loops with the same body can be fused into one when the ending index of the first loop is the first index of the second loop (loop fusion), and a loop whose loop body does not depend on the iteration count can be reversed (loop reversal). Similarly, we prove that `if true e e' = e`, and `if false e e' = e'`.

These do *not* count as peephole rewrites in our framework, as they are universally quantified over the loop body (which is a region). This is unsupported – peephole rewrites in $\text{LeanMLIR}(X)$ may only have free variables, not free region arguments.

Consider the loop optimization that converts iterated addition into a single multiplication. Its proof obligation is $(\vdash \lambda x. x + \delta)^n(c) = n \cdot \delta + c$. This transformation is challenging to perform in a low-level IR, since there is no syntactic concept of a loop. However, this transformation *is* a valid peephole rewrite in our framework since it uses a *statically* known loop body. We showcase how regions permit MLIR (and, consequently, us) to easily encode and reason with commonplace loop transformations. The parametricity of our framework allows us to prove theorems that are valid on all IR extensions $\text{scf}(X)$.

6.3 Fully Homomorphic Encryption

A key motivation for $\text{LeanMLIR}(X)$ is to enable specifying formal semantics for high-level, mathematical IRs. These IRs require access to complex mathematical objects that are available in proof assistants, and verifying rewrites on such IRs is out of practical reach for

today’s SMT solvers. As a case study, we formalize the complete “Poly” IR.¹⁴ This IR is a work in progress and is in flux, as it is part of the discussion of an upcoming open standard for homomorphic encryption, developed in collaboration by Intel and Google.¹⁵ Contrary to what its naming implies, this IR does *not* model operations on polynomials.¹⁶ Instead, codewords are encoded as elements in a finitely-presented commutative ring, specifically, the ring $R \equiv (\mathbb{Z}/q\mathbb{Z})[x]/(x^{2^n} + 1)$, where $q, n \in \mathbb{N}$ are positive integers (q composite). The name “Poly” comes from the equivalence class representatives are polynomials, but not all IR operations are invariants of the equivalence class.

The “Poly” IR is, in fact, a superset of the `QuotRing` IR we defined in Section 2. It consists of the operations `add`, `sub`, `mul`, `mul_constant`, `leading_term`, `monomial`, `monomial_mul`, `from_tensor`, `to_tensor`, `arith.constant` and `constant`.¹⁷

Most of these operations are self-explanatory and derive from the (commutative) ring structure of R or are used to build elements in R , like the equivalence classes of constants or monomials. Three operations, `to_tensor` and `from_tensor` and `leading_term` do not follow directly from the algebraic properties of the polynomial ring. Instead, they depend on a (non-canonical) choice of representatives for each ideal coset in the polynomial ring. More precisely, let $\pi : (\mathbb{Z}/q\mathbb{Z})[x] \rightarrow (\mathbb{Z}/q\mathbb{Z})[x]/(x^{2^n} + 1)$ be the canonical surjection into the quotient, taking a polynomial to its equivalence class modulo division by $x^{2^n} + 1$. Further let $\sigma : (\mathbb{Z}/q\mathbb{Z})[x]/(x^{2^n} + 1) \hookrightarrow \mathbb{Z}/q\mathbb{Z}[x]$ be the injection taking an equivalence class to its (unique) representative with degree $\leq 2^n$. This is a right-inverse of π , i.e. $\pi \circ \sigma = id$. Note that multiple right-inverses could have been chosen for σ : As long as $\sigma(x)$ is a representative of the equivalence class of x for all $x \in (\mathbb{Z}/q\mathbb{Z})[x]/(x^{2^n} + 1)$, σ will be a right-inverse of π . The operation `to_tensor(p)` returns the vector $(\sigma(p)[i])_{i=0,\dots,2^n}$, where $a[i]$ represents the i -th coefficient, i.e. $\sigma(p) = \sum_{i=0}^{2^n} (\sigma(p)[i])x^i$, and `to_tensor` the converse. Similarly, `leading_term(p)` returns the equivalence class of the leading term of the representative $\sigma(p)$ (which also depends on the choice of σ).

This allows us to define the semantics and prototype both the IR and rewrites in it. Rewrites like `mul(p,q) → mul(q,p)` follow immediately from the fact that R is a commutative ring. Other rewrites like `from_tensor(to_tensor(p)) → p`, or even `add(p,monomial(1,2^n)) → sub(p,1)`, on the other hand, are more specific to this IR and have a higher manual-proof overhead. We prove all of these.

We discussed the IR and potential semantics with the authors of the HEIR IR in the context of the upcoming open standard for homomorphic encryption. We believe that a framework like the one presented in this paper will allow standards like these to be defined with formal semantics from the ground up.

7 Related Work

The semantics of LLVM, the spiritual ancestor of MLIR, have been well-studied. Both Vellvm [46] and K-LLVM [22] formalized a large portion of LLVM, including reasoning about SSA transformations explicitly [47]. Alive [24] and Alive 2 [23] provide push-button verification for a subset of LLVM by leveraging SMT solvers. Alive-tv does the same for a set of concrete IRs for tensor operations in MLIR [3]. AliveInLean [19] proves the correctness

¹⁴ as of commit 2db7701de

¹⁵ <https://homomorphicencryption.org/>

¹⁶ In the same way that rationals \mathbb{Q} are not pairs of integers $\mathbb{Z} \times \mathbb{Z}$.

¹⁷ It also has distinct types for integers and naturals, which we unified in Section 2 for simplicity.

of the translation from the Alive DSL into SMT expressions, as well as the correctness of their encoding of program refinement as an SMT expression. In contrast, our work focuses on building a framework for describing full programs (rather than rewrite snippets), and formally defines and proves the correctness of peephole rewriting within a larger program context. The semantics and correctness of compiling compositionally have been explored by multiple authors, like Pilsener [29] or many variants of CompCert [21]: like compositional CompCert [37], CompCertX [42], SepCompCert [14], CompCertM [36], and CompCertO [16]. A great summary of the approaches to this problem (including the ones mentioned above), with their differences and similarities, is given by Patterson et al [31]. All of these use fixed languages but are reasonable ways of giving semantics to relevant IRs in `LeanMLIR(X)`.

The authors of [45] introduce a more modular approach to LLVM’s semantics, based on interaction trees [44]. Like theirs, our semantics is also denotational and can be executed. We currently only model a restricted set of side effects, whereas interaction trees shine when modeling more complex side effects such as memory or non-terminating behavior. An approach like this would thus be a great candidate for the semantics of a lower-level IR such as LLVM within MLIR. Similarly, the work of DimSum [35] deals with the boundaries between languages in the context of linking. This addresses also important part aspect we don’t model yet: what occurs at the boundaries of IRs, when mixing them.

There is a longer line of work studying SSA and its relationship to functional programming. Our work is inspired by and builds on the ideas from [15, 2, 5]. Complex compiler optimizations have also been studied formally and verified, like [7] which implements verified polyhedral optimization. We focus on the simpler and more ubiquitous peephole rewrites.

Our work differs from prior work on formalizing peephole rewrites by providing a framework for reasoning about SSA peephole rewrites. The closest similar work, Peek [28] defines peephole rewriting over an assembly instruction set. Their rewriter expects instructions to be adjacent to one another. Furthermore, their rewriter restricts source and target patterns to be of the same length, filling in the different lengths with `nop` instructions. Their patterns permit side effects, which we disallow since we are interested in higher-level, pure rewrites. Our patterns provide more flexibility since the source and target patterns are arbitrary programs, and are matched on sub-DAGs instead of a linear sequence.

8 Conclusion

Peephole rewrites represent a large and important class of compiler optimizations. We have seen how domain-specific IRs in SSA with regions greatly extend the scope of these peephole rewrites. They raise the level of abstraction both syntactically with def-use chains and nesting, and semantically, with domain-specific abstractions. We have shown how to reason effectively about such SSA-based compilers, and, specifically, local reasoning in the form of peephole rewrites. We advocate building on top of a proof assistant with a small TCB, an expressive language and a large library of mathematics. This increases the confidence in our verification and extends its applicability to many domains where more specialized methods don’t exist. We also advocate proof automation and an intrinsically well-typed mechanized core that can be designed to focus on the semantics of the domain. We incarnate these principles in `LeanMLIR(X)`, a framework built on Lean and `mathlib` to reason about domain-specific IRs in SSA with regions. We show how `LeanMLIR(X)` is simple to use, amenable to automation, and effective for verifying IRs over complex domains.

References

- 1 AV Aho, R Sethi, and JD Ullman. *Compilers: Principles, Techniques, and Tools*. Citeseer, 1985.
- 2 Andrew W Appel. SSA is functional programming. *Acm Sigplan Notices*, 33(4):17–20, 1998.
- 3 Seongwon Bang, Seunghyeon Nam, Inwhan Chun, Ho Young Jhoo, and Juneyoung Lee. SMT-based translation validation for machine learning compiler. In Sharon Shoham and Yakir Vizel, editors, *Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part II*, volume 13372 of *Lecture Notes in Computer Science*, pages 386–407. Springer, 2022. doi:10.1007/978-3-031-13188-2_19.
- 4 Clark Barrett, Aaron Stump, Cesare Tinelli, et al. The smt-lib standard: Version 2.0. In *Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, UK)*, volume 13, page 14, 2010.
- 5 Gilles Barthe, Delphine Demange, and David Pichardie. Formal verification of an SSA-based middle-end for CompCert. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 36(1):1–35, 2014.
- 6 Siddharth Bhat and Tobias Grosser. Lambda the ultimate ssa: optimizing functional programs in ssa. In *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 1–11. IEEE, 2022.
- 7 Nathanaël Courant and Xavier Leroy. Verified code generation for the polyhedral model. *Proceedings of the ACM on Programming Languages*, 5(POPL):1–24, 2021.
- 8 Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems: 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings 14*, pages 337–340. Springer, 2008.
- 9 Leonardo de Moura and Sebastian Ullrich. The Lean 4 theorem prover and programming language. In *International Conference on Automated Deduction*, pages 625–635. Springer, 2021.
- 10 Craig Gentry. *A fully homomorphic encryption scheme*. Stanford university, 2009.
- 11 Benjamin Grégoire and Assia Mahboubi. Proving equalities in a commutative ring done right in Coq. In Joe Hurd and Thomas F. Melham, editors, *Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005, Oxford, UK, August 22-25, 2005, Proceedings*, volume 3603 of *Lecture Notes in Computer Science*, pages 98–113. Springer, 2005. doi:10.1007/11541868_7.
- 12 Gérard Huet. The zipper. *Journal of functional programming*, 7(5):549–554, 1997.
- 13 Petter Källström and Oscar Gustafsson. Fast and area efficient adder for wide data in recent Xilinx FPGAs. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4. IEEE, 2016.
- 14 Jeehoon Kang, Yoonseung Kim, Chung-Kil Hur, Derek Dreyer, and Viktor Vafeiadis. Lightweight verification of separate compilation. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 178–190, 2016.
- 15 Richard A Kelsey. A correspondence between continuation passing style and static single assignment form. *ACM SIGPLAN Notices*, 30(3):13–22, 1995.
- 16 Jérémie Koenig and Zhong Shao. CompCertO: compiling certified open C components. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 1095–1109, 2021.
- 17 Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.


- 18 Chris Lattner, Jacques Pienaar, Mehdi Amini, Uday Bondhugula, River Riddle, Albert Cohen, Tatiana Shpeisman, Andy Davis, Nicolas Vasilache, and Oleksandr Zinenko. MLIR: A compiler infrastructure for the end of Moore’s law. *arXiv preprint arXiv:2002.11054*, 2020.
- 19 Juneyoung Lee, Chung-Kil Hur, and Nuno P Lopes. AliveInLean: a verified LLVM peephole optimization verifier. In *International Conference on Computer Aided Verification*, pages 445–455. Springer, 2019.
- 20 Juneyoung Lee, Yoonseung Kim, Youngju Song, Chung-Kil Hur, Sanjoy Das, David Majnemer, John Regehr, and Nuno P Lopes. Taming undefined behavior in LLVM. *ACM SIGPLAN Notices*, 52(6):633–647, 2017.
- 21 Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. CompCert—a formally verified optimizing compiler. In *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*, 2016.
- 22 Liyi Li and Elsa L Gunter. K-LLVM: a relatively complete semantics of LLVM IR. In *34th European Conference on Object-Oriented Programming (ECOOP 2020)*, 2020.
- 23 Nuno P Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. Alive2: bounded translation validation for LLVM. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 65–79, 2021.
- 24 Nuno P Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. Provably correct peephole optimizations with alive. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 22–32, 2015.
- 25 Carl McConnell and Ralph E. Johnson. Using static single assignment form in a code optimizer. *ACM Lett. Program. Lang. Syst.*, 1(2):152–160, June 1992. doi:10.1145/151333.151368.
- 26 William M McKeeman. Peephole optimization. *Communications of the ACM*, 8(7):443–444, 1965.
- 27 Manasij Mukherjee and John Regehr. Hydra: Generalizing peephole optimizations with program synthesis. *Proceedings of the ACM on Programming Languages*, (OOPSLA), 2024.
- 28 Eric Mullen, Daryl Zuniga, Zachary Tatlock, and Dan Grossman. Verified peephole optimizations for CompCert. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 448–461, 2016.
- 29 Georg Neis, Chung-Kil Hur, Jan-Oliver Kaiser, Craig McLaughlin, Derek Dreyer, and Viktor Vafeiadis. Pilsner: A compositionally verified compiler for a higher-order imperative language. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, pages 166–178, 2015.
- 30 Sunjae Park, Woosung Song, Seunghyeon Nam, Hyeongyu Kim, Junbum Shin, and Juneyoung Lee. HEaaN.MLIR: An optimizing compiler for fast ring-based homomorphic encryption. *Proceedings of the 44th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2023.
- 31 Daniel Patterson and Amal Ahmed. The next 700 compiler correctness theorems (functional pearl). *Proceedings of the ACM on Programming Languages*, 3(ICFP):1–29, 2019.
- 32 Anurudh Peduri, Siddharth Bhat, and Tobias Grosser. QSSA: an SSA-based IR for quantum computing. In *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction*, pages 2–14, 2022.
- 33 Benjamin C Pierce and C Benjamin. *Types and programming languages*. MIT press, 2002.
- 34 Fabrice Rastello and Florent Bouchez Tichadou. *SSA-based Compiler Design*. Springer Nature, 2022.
- 35 Michael Sammler, Simon Spies, Youngju Song, Emanuele D’Osualdo, Robbert Krebbers, Deepak Garg, and Derek Dreyer. Dimsum: A decentralized approach to multi-language semantics and verification. *Proceedings of the ACM on Programming Languages*, 7(POPL):775–805, 2023.

- 36 Youngju Song, Minki Cho, Dongjoo Kim, Yonghyun Kim, Jeehoon Kang, and Chung-Kil Hur. CompCertM: CompCert with C-assembly linking and lightweight modular verification. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–31, 2019.
- 37 Gordon Stewart, Lennart Beringer, Santiago Cuellar, and Andrew W Appel. Compositional CompCert. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 275–287, 2015.
- 38 Sebastian Ullrich and Leonardo de Moura. Beyond notations: Hygienic macro expansion for theorem proving languages. In *International Joint Conference on Automated Reasoning*, pages 167–182. Springer, 2020.
- 39 Nicolas Vasilache, Oleksandr Zinenko, Aart J. C. Bik, Mahesh Ravishankar, Thomas Raoux, Alexander Belyaev, Matthias Springer, Tobias Gysi, Diego Caballero, Stephan Herhut, Stella Laurenzo, and Albert Cohen. Composable and modular code generation in MLIR: A structured and retargetable approach to tensor compiler construction. *CoRR*, abs/2202.03293, 2022. [arXiv:2202.03293](https://arxiv.org/abs/2202.03293).
- 40 Alexander Viand, Patrick Jattke, Miro Haller, and Anwar Hithnawi. HECO: Automatic code optimizations for efficient fully homomorphic encryption. *arXiv preprint arXiv:2202.01649*, 2022.
- 41 Wei Wang and Xinming Huang. A novel fast modular multiplier architecture for 8,192-bit RSA cryptosystem. In *2013 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–5. IEEE, 2013.
- 42 Yuting Wang, Pierre Wilke, and Zhong Shao. An abstract stack based approach to verified compositional compilation to machine code. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–30, 2019.
- 43 Dominik Winterer, Chengyu Zhang, and Zhendong Su. Validating SMT solvers via semantic fusion. In Alastair F. Donaldson and Emina Torlak, editors, *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, pages 718–730. ACM, 2020. doi:10.1145/3385412.3385985.
- 44 Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C Pierce, and Steve Zdancewic. Interaction trees: representing recursive and impure programs in Coq. *arXiv preprint arXiv:1906.00046*, 2019.
- 45 Yannick Zakowski, Calvin Beck, Irene Yoon, Ilia Zaichuk, Vadim Zaliva, and Steve Zdancewic. Modular, compositional, and executable formal semantics for LLVM IR. *Proc. ACM Program. Lang.*, 2021. doi:10.1145/3473572.
- 46 Jianzhou Zhao, Santosh Nagarakatte, Milo MK Martin, and Steve Zdancewic. Formalizing the LLVM intermediate representation for verified program transformations. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 427–440, 2012.
- 47 Jianzhou Zhao, Santosh Nagarakatte, Milo MK Martin, and Steve Zdancewic. Formal verification of SSA-based optimizations for LLVM. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, pages 175–186, 2013.

Duper: A Proof-Producing Superposition Theorem Prover for Dependent Type Theory

Joshua Clune   

Carnegie Mellon University, Pittsburgh, PA, USA

Yicheng Qian 

Peking University, Beijing, China

Alexander Bentkamp   

Heinrich-Heine-Universität Düsseldorf, Germany

Jeremy Avigad   

Carnegie Mellon University, Pittsburgh, PA, USA

Abstract

We present Duper, a proof-producing theorem prover for Lean based on the superposition calculus. Duper can be called directly as a terminal tactic in interactive Lean proofs, but is also designed with proof reconstruction for a future Lean hammer in mind. In this paper, we describe Duper's underlying approach to proof search and proof reconstruction with a particular emphasis on the challenges of working in a dependent type theory. We also compare Duper's performance to Metis' on pre-existing benchmarks to give evidence that Duper is performant enough to be useful for proof reconstruction in a hammer.

2012 ACM Subject Classification Theory of computation → Automated reasoning; Theory of computation → Logic and verification; Theory of computation → Higher order logic; Theory of computation → Type theory

Keywords and phrases proof search, automatic theorem proving, interactive theorem proving, Lean, dependent type theory

Digital Object Identifier 10.4230/LIPIcs.ITP.2024.10

Supplementary Material *Software*: https://github.com/JOSHCLUNE/Duper_ITP_Paper_Artifact
archived at `swh:1:dir:ece08eae9c83476bea9fa47d80c266d06d008bed`

Acknowledgements We thank Mario Carneiro, Rob Lewis, and Gabriel Ebner for their advice in the early stages of the project. We thank Lydia Kondylidou for her input on our evaluation section and Jasmin Blanchette for his feedback on this paper and input on our evaluation section. We also thank Haniel Barbosa and the anonymous reviewers for their feedback on this paper.

1 Introduction

Interactive and automated theorem proving offer complementary strengths. Push-button automation is convenient but limited to small problems or restricted problem domains, and generally it does not provide strong correctness guarantees. Interactive theorem provers make it possible to verify just about any theorem with a high degree of confidence in the correctness of the result, but the work required is often unpleasant or impracticable. Most modern proof assistants therefore rely on multiple forms of supporting automation. Domain-specific tactics polish off goals in linear or linear integer arithmetic [17, 37], carry out calculations in any ring [19], and even perform inferences involving abstract geometric structures [39]. Term rewriters carry out general equational simplification [33], and tools like Isabelle's *Auto* [34], HOL Light's *Meson* [20], and Lean's *Aesop* [26] implement general tableau search.

Isabelle's celebrated *Sledgehammer* [31, 32, 35, 36] provides powerful domain-general automation by exporting problems to external provers and then harvesting enough information to reconstruct and verify the results. A proof-producing ordered resolution prover,



© Joshua Clune, Yicheng Qian, Alexander Bentkamp, and Jeremy Avigad;
licensed under Creative Commons License CC-BY 4.0

15th International Conference on Interactive Theorem Proving (ITP 2024).

Editors: Yves Bertot, Temur Kutsia, and Michael Norrish; Article No. 10; pp. 10:1–10:20

Leibniz International Proceedings in Informatics



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

Metis [21], is frequently used for the latter purpose. Similar hammers for proof assistants based on first-order logic or simple type theory include HOL(y) Hammer [22], a hammer for Mizar [23], and a hammer for Metamath [12]. (See also [9] for a survey overview.) Dependently typed foundations like those of Coq [7], Agda [10], and Lean [15] offer better support for algebraic reasoning, but the added expressivity comes at a cost, and developing domain-general automation for dependently typed foundations is notoriously hard. Even applying a single lemma can require complex unification and elaboration processes, including definitional reductions, type class search, and other means of inferring implicit arguments. There is a hammer for Coq, CoqHammer [14], which uses a custom-designed inhabitation procedure for proof reconstruction [13], but that reconstruction procedure is tuned to Coq’s intuitionistic framework. A prototype hammer for Agda [18] uses only equational reasoning for reconstruction. There is currently no hammer for Lean.

This paper introduces *Duper*, a proof-producing theorem prover for Lean based on the superposition calculus [3]. *Duper* is written in the Lean programming language and it directly generates proofs as expressions in Lean’s axiomatic foundation. It can be called as a tactic when writing proofs interactively in Lean, but it is also intended to serve as a means of proof reconstruction for external automation. Most notably, *Duper* is designed to work in a dependently typed setting. Some of the challenges of working in a dependent type theory, such as the need to use type class inference to instantiate algebraic structures, are best handled in a monomorphization and preprocessing phase. For that, we use a tool called *LeanAuto*, developed by the second author, which will be described in detail elsewhere. Other challenges, such as Skolemizing formulas in an axiomatic framework in which quantifiers can, in principle, range over empty types, have to be handled natively. We deal with remaining challenges, such as polymorphic reasoning over types, polymorphic reasoning over type universes, and higher-order reasoning, using a flexible combination of preprocessing and native handling. *Duper* uses methods inspired by the Zipperposition prover [45] to carry out higher-order inferences, and the framework is flexible enough to accommodate, in the main loop, inferences that are specific to dependent types.

Our contributions are as follows:

- We have developed new ways of incorporating important aspects of dependently typed reasoning in a superposition theorem prover.
- We have developed ways of generating proofs directly in a dependently typed axiomatic framework.
- We make effective use of *LeanAuto*’s preprocessing to prove the kinds of goals that arise in practice when working with Lean and its library, *Mathlib*.
- We show that *Duper*’s performance is comparable to *Metis*’ on standard benchmarks in the interactive theorem proving community, specifically the *Seventeen provers* [16] and *GRUNGE* benchmarks [11].

Duper is available online at <https://github.com/leanprover-community/duper>.

2 Proof Search

Duper is a superposition theorem prover implemented in Lean. It accepts as input a goal state of the form $E(\Gamma \vdash p : \text{Prop})$ where Γ is a local context, or list of hypotheses with corresponding types, E is a global environment, or list of declarations, and p is the target proposition. *Duper* also accepts as input a list of lemmas from E that may be relevant to the provided goal. Since *Duper* is not yet equipped with a relevance filter, these lemmas must currently be supplied manually. Given a goal state and lemma list, *Duper* will attempt to produce a proof term t such that $E(\Gamma \vdash t : p)$.

Although Duper’s input and output formats are tailored for closing Lean goals, its approach to proof search more closely resembles that of standalone automatic theorem provers than other forms of Lean automation. In this section, we describe the core aspects of this approach that are essential to Duper’s functioning as a superposition theorem prover.

2.1 Main Saturation Loop

Given a goal state $E(\Gamma \vdash p : \text{Prop})$, the first thing Duper does is negate the target p and add it to the local context Γ , resulting in the modified goal state $E(\Gamma, \neg p \vdash \text{False} : \text{Prop})$. Having a goal state of this form allows Duper to enter a main saturation loop in which it attempts to deduce all possible inferences from Γ , $\neg p$, and the user-provided lemmas until either a contradiction is derived or no further inferences can be performed. In the former case, Duper uses the set of hypotheses that yielded a contradiction to produce the desired proof term t , and in the latter case, Duper informs the user that it was unable to prove the goal with the available lemmas. This high-level approach to proof search makes Duper a *saturation-based* theorem prover.

The procedure that drives the main saturation loop in most saturation-based theorem provers is called the *given clause procedure* [28]. Since different theorem provers implement different calculi and search heuristics, there are multiple variants of this procedure [8], the most common of which are the Otter loop [30] and the DISCOUNT loop [1]. Duper implements a variant of the DISCOUNT loop, described by Vukmirović et al. [45], that is designed to address complications that arise in higher-order unification.

At a high level, Duper’s given clause procedure functions by partitioning its derived facts into a set of fully processed clauses called the active set and a set of unprocessed formulas called the passive set. In each iteration of the main saturation loop, the procedure selects a new formula from the passive set called the “given clause,” clasifies it, simplifies it, and then uses generating rules to produce conclusions from it before transferring it to the active set and proceeding to the next iteration of the loop. Conceptually, this control flow is simple, though there are a variety of complications, such as the fact that some of Duper’s inference rules can generate infinitely many clauses. Solutions to complications arising from higher-order reasoning are described by Vukmirović et al. [45], and our solutions to complications arising specifically from dependent type theory are described in Section 4.2.2.

Aside from implementing a prover’s central control flow, one of the primary purposes of a given clause procedure is to manage redundancy. Since the search space that an automatic theorem prover can consider is so large, reducing it as much as possible is an extremely important part of building a performant prover. Toward this end, modern provers implement a variety of heuristics, strategies, and features to help minimize the time spent reasoning about clauses that are no longer necessary to obtain a contradiction.

One of the main ways the given clause procedure manages redundancy is by performing simplification rules. Simplification rules are like generating rules in that they can produce new conclusions for Duper to reason about, but they are unlike generating rules in that they also eliminate one of the rule’s premises from future consideration. Simplification rules can be applied as forward simplification rules, meaning they use clauses from the active set to modify or eliminate the current given clause, or as backward simplification rules, meaning they use the current given clause to modify or eliminate clauses in the active set.

Typically, in each iteration of the main saturation loop, forward simplification rules are applied first, then backward simplification rules, and finally inference rules. This order is ideal because it ensures that the given clause is simplified before it is used to modify the active set and that the active set is reduced as much as possible before it is used to generate inferences. Although this order is preferred, there are some special cases in which it must be violated. Such cases are discussed in Section 4.2.2.

2.2 Core Calculus

The core calculus Duper uses to generate new facts in its main saturation loop is based on Zipperposition’s $\text{o}\lambda\text{Sup}$ calculus [5]. There are two primary reasons we chose to use this calculus. First, Zipperposition is a state-of-the-art theorem prover with an established track record of high performance, especially on higher-order problems [40, 41, 42, 44]. Second, Zipperposition’s calculus is designed to be a graceful generalization of first-order superposition, rather than a translation-based approach that transforms higher-order problems into first-order logic before calling an essentially first-order procedure. Since Duper operates in a dependently typed setting, we preferred an approach that supports native higher-order reasoning and can be extended to support native dependent type theory reasoning over an approach that requires translating to a less expressive logic.

Duper’s implementation of the superposition calculus has three primary components. First, there is the implementation of each individual rule and its corresponding proof-reconstruction procedure. For the most part, there are few substantive differences between how Duper implements its inference rules and how other provers might implement the same rules.

One aspect of rule implementation worth mentioning is how Duper defines the clauses that its inference rules act on. Most of Duper’s inference rules involve unification. Since that involves mapping variables to specific values, a clause’s variables may take on particular values for the sake of one inference but not others. It is therefore convenient to store clauses both in a permanent format that will be unchanged by inference generation as well as in a temporary format that can be modified by the unification procedure. To avoid confusion, we refer to facts in the first format as clauses and facts in the second format as mclauses. This naming convention arises from the fact that mclauses represent their variables with Lean metavariables whose types and values can easily be modified by the unification procedure.

The second component of Duper’s core calculus is a strict order on terms, literals, and mclauses. Having such an order enables Duper to impose side conditions on its inference rules which drastically improve performance by limiting the number of possible valid inferences. For this, Duper uses the Knuth-Bendix order (KBO) [24, 27] extended to be compatible with higher-order logic [5].

To implement this order for Lean expressions, which come in more categories than just symbols and variables, only a few additions and modifications are necessary. In mclauses, Duper uses Lean metavariables as free variables, and treats Lean constants, free variables, projections, strings, and natural numbers as “symbols.” Of these symbols, Duper treats constants as greatest, followed by free variables, projections, strings, and natural numbers, in that order. Lean expressions with additional metadata are definitionally equivalent to those same expressions without the metadata, so Duper simply removes them from all expressions prior to calling the KBO procedure. To handle λ -expressions, Duper follows the approach described by Bentkamp et al. [6]. This involves exhaustively applying β - and η -reduction rules to get rid of as many λ -expressions as possible, and then treating any remaining λ -expressions of the form $(\lambda x : t. e[x])$ as expressions of the form $(\text{LAM } t \ e[\#0])$ where “LAM” is a constant and $\#0$ is a bound variable with De Bruijn index 0. Finally, to handle let expressions, Duper’s KBO procedure applies ζ -reduction exhaustively along with β - and η -reduction which guarantees that no let expressions need to be considered.

The final component of Duper’s core calculus is a higher-order unification procedure. We extend the approach outlined by Vukmirović et al. [46] to dependent type theory. This enables Duper to generate arbitrarily many solutions to unification problems where infinitely many incomparable unifiers may exist. The specifics of how we extend this approach to dependent type theory are described in Section 4.1.

3 Proof Reconstruction

Once Duper derives a contradiction in its main saturation loop, it uses that contradiction to construct a proof term t such that $E(\Gamma, \neg p \vdash t : \text{False})$. Since Duper always begins by applying double-negation elimination to transform its input goal $E(\Gamma \vdash p : \text{Prop})$ into the modified goal $E(\Gamma, \neg p \vdash \text{False} : \text{Prop})$, this suffices to close the original goal. We note that although this initial transformation is standard for many automatic theorem provers, it is only sound classically, as are many of the rules in Duper’s underlying superposition calculus. Consequently, Duper only generates classical proofs, even when given a goal that could be proved intuitionistically.

To construct t , Duper begins by collecting the list of clauses that were used to derive the contradiction it found. Duper then generates proof terms for each clause in this list using proof terms from earlier clauses or facts supplied as input. Since the final clause that Duper generates is the empty clause, the final proof term that Duper generates is t .

Since the addition of a new clause is justified by a rule and a list of parent clauses, one can construct the proof term for the result of an inference from those of the inference’s premises. The specifics of how each result is justified depend on which rule is used to produce it, so each of Duper’s rules is paired with a unique proof reconstruction procedure. In this section, we discuss two ideas relevant to the proof reconstruction procedure of several of our rules.

3.1 Clause Instantiation

Each of Duper’s clauses is a disjunction of literals preceded by arbitrarily many universal quantifiers. So if conclusion D is meant to follow from premises C_1 and C_2 , the first step to constructing D ’s proof term is usually to instantiate all of the universal quantifiers at the heads of C_1 and C_2 . To guarantee that this is possible, if a rule’s proof reconstruction procedure is known to require this first step, the rule itself will output conclusions whose universal quantifiers include all of its parents’ universal quantifiers. Duper can then instantiate C_1 ’s and C_2 ’s universal quantifiers using free variables introduced from the target D .

Although this restriction on Duper’s inference rules ensures that Duper will always be able to instantiate C_1 ’s and C_2 ’s universal quantifiers with free variables introduced from D , there are some instances in which Duper must adopt a different instantiation strategy. This happens primarily when metavariables in mclauses are instantiated with particular values by Duper’s unification procedure. Since metavariables in mclauses correspond to bound variables in regular clauses, instantiating an mclause’s metavariable corresponds to instantiating a clause’s universal quantifier. When this instantiation strategy is used, the term used to instantiate the relevant quantifier is provided by the unification procedure.

This approach has some subtle consequences. The first is that Duper’s inference rules can generate conclusions containing universal quantifiers whose variables do not appear in the conclusion’s clause body. For example, given the premise $\forall x : \alpha, f x \neq f x \vee a = b$, Duper will recognize that the literal $f x \neq f x$ is always false and eliminate it to produce the conclusion $\forall x : \alpha, a = b$. Even though x does not appear in the clause body $a = b$, the conclusion includes x ’s universal quantifier since x appears in the premise and is not assigned a value by the unification procedure.

To compensate for this behavior, Duper implements an additional simplification rule, called `removeVanishedVars`, which takes an arbitrary premise and attempts to produce a conclusion containing an identical clause body and strictly fewer universal quantifiers. Applying this rule requires knowing that the removed quantifiers range over inhabited types, so Duper must perform additional reasoning to determine which types are provably inhabited. The specifics of this additional reasoning are discussed in Section 4.2.1.

The second subtle consequence of this approach is that although most of Duper’s proof reconstruction can be delayed until after it is known which clauses appear in the final proof, some amount of proof reconstruction must be performed for every generated clause. Specifically, after any rule is carried out, all metavariable instantiations performed by the unification procedure must be recorded before the rule’s mclauses are forgotten. For this reason, Duper constructs partially instantiated parent terms immediately rather than wait until the end of the proof search. Transforming the conclusion’s mclause into a clause also requires taking the unification procedure’s instantiations into account, so performing these partial instantiations immediately does not result in significant additional overhead.

3.2 Transfer Expressions

The only input required by most rules’ proof reconstruction procedures is a list of proof terms corresponding to each premise and the type of the desired conclusion. In many cases, a rule’s proof reconstruction procedure amounts to applying a single polymorphic Lean theorem that justifies the rule’s soundness. However, some rules require additional information.

One example of such a rule is `ForallHoist`. This rule takes a clause C containing an expression e that can be unified with $(\forall x : ?m1, (?m2 x))$ and cases on whether e is true. It does this by replacing e with `False` in C and giving C a new literal of the form $(?m2 ?m3 = \text{True})$ where $?m3$ is a fresh metavariable of type $?m1$. The idea is that if e is false, then replacing it with `False` in C is sound, and if e is true, then $(?m2 ?m3)$ will also evaluate to true.

The issue is that although $?m2$ appears in the parent clause that `ForallHoist` takes, $?m3$ does not. So it’s not possible to produce a proof term for the correct conclusion by merely passing the parent’s proof term into even a polymorphic Lean theorem. The result type would contain an unassigned metavariable, which is disallowed by Lean’s application procedure. It is therefore necessary to, in addition to supplying the parent’s proof term, supply an expression corresponding to $?m3$. In principle, this expression can be derived from the type of the desired conclusion, but in practice, it is both simpler and more efficient to pass the necessary expression to the proof reconstruction procedure directly.

We call expressions that are passed directly to proof reconstruction procedures but are not proof terms for parent clauses “transfer expressions.” Several of the higher-order rules that Duper implements have proof reconstruction procedures that benefit from such transfer expressions, usually because these rules produce more involved unification problems. If Duper were a standalone automatic theorem prover that needed to interface with a separate interactive theorem prover, passing along transfer expressions might be prohibitively difficult. But because Duper is implemented in Lean, has clauses defined in terms of Lean expressions, and makes use of several of Lean’s metaprogramming functionalities, implementing transfer expressions is relatively straightforward.

4 Native Dependent Type Theory Reasoning

Lean’s type theory is based on the calculus of constructions (CoC) with a countable hierarchy of non-cumulative universes and inductive types [2]. Lean moreover allows users to mark arguments to be instantiated by type class inference. Developing an automatic theorem prover that can operate in the presence of these features poses a variety of challenges. Here, we discuss the problems that Duper handles natively. Additional issues that are addressed during preprocessing are discussed in Section 5.

4.1 Unification

Many of Duper’s inference rules involve unification. To ensure these rules can be applied even when dependent types are at play, it is important that Duper can support unifying dependently typed terms. Consider the following example which involves matrices defined in terms of $\text{Fin } n$, the (dependent) type of values less than n :

```
example (a b : Nat) (matrix : Fin a → Fin b → Nat)
  (transpose : ∀ n m : Nat, (Fin n → Fin m → Nat) → (Fin m → Fin n → Nat))
  (h : ∀ n m : Nat, (fun x => transpose n m (transpose m n x)) = (fun x => x)) :
  transpose b a (transpose a b matrix) = matrix := by
  duper [h]
```

This example can be proved by using h to rewrite $\text{transpose } b \ a \ (\text{transpose } a \ b \ \text{matrix})$ as $(\text{fun } x \Rightarrow x) \ \text{matrix}$. But in order to perform this straightforward rewrite, it is necessary to unify matrix of type $\text{Fin } a \rightarrow \text{Fin } b \rightarrow \text{Nat}$ with x of type $\text{Fin } ?m \rightarrow \text{Fin } ?n \rightarrow \text{Nat}$ (where $?m$ and $?n$ are assignable metavariables). From this, we can see that even simple examples may require Duper to use a unification procedure that supports dependently typed terms.

The unification procedure in Duper is based on that of Vukmirović et al. [46]. To unify two terms s and t , the unification procedure builds a tree with nodes of the form (E, σ) , where E is a multiset of unification constraints $\{(s_1 \stackrel{?}{=} t_1), \dots, (s_n \stackrel{?}{=} t_n)\}$ and σ is a substitution. The tree is built by starting from the root $(\{s \stackrel{?}{=} t\}, \emptyset)$ and progressively creating nodes according to transition rules. A transition rule $(E, \sigma) \rightarrow (E', \sigma')$ allows one to attach a new node (E', σ') to an existing node (E, σ) . It is expected that the solutions represented by (E', σ') is a subset of that of (E, σ) , and that either $\sigma' = \sigma$ or σ' is a refinement of σ .

Among all the transition rules, the Bind rule is of great importance because it is the main rule that refines the substitution. The Bind rule is defined as

$$(\{s \stackrel{?}{=} t\} \uplus E, \sigma) \rightarrow (\{s \stackrel{?}{=} t\} \uplus E, \rho\sigma) \quad \rho \in \mathcal{P}(s \stackrel{?}{=} t)$$

where $\mathcal{P}(s \stackrel{?}{=} t)$ is the set of bindings applicable to $s \stackrel{?}{=} t$.

Both the transitions and bindings implemented by Duper are a superset of the ones described by Vukmirović et al., hence Duper’s procedure is complete for the higher-order fragment of Lean. In the following discussion, we use the notations and conventions established by Vukmirović et al., including describing the unification procedure as acting on free variables. Note that because Duper’s unification procedure operates on terms taken from mclauses, free variables in the following discussion correspond with metavariables in our implementation.

4.1.1 Transitions

In dependent type theory, types are also terms, and types can contain \forall binders. Consider the unification equation $\forall x. F \ x \stackrel{?}{=} \forall x. g \ x$ where F is a free variable and g is rigid. It is easy to see that $F \mapsto g$ is a solution. However, there are no transitions in Vukmirović et al.’s higher-order procedure applicable to this unification equation. To address this issue, we introduce a new transition:

$$\text{ForallToLambda} : (\{\forall x. f \ x \stackrel{?}{=} \forall x. g \ x\} \uplus E, \sigma) \rightarrow (\{\lambda x. f \ x \stackrel{?}{=} \lambda x. g \ x\} \uplus E, \sigma)$$

Apart from this, the Fail and Decompose transitions from Vukmirović et al. are extended to handle dependent type theory features.

4.1.2 Bindings

In this section, we explain how *Imitation Binding* of Vukmirović et al. is extended to account for dependent types. Other bindings are extended in a similar way. Apart from these, a new binding *ImitForall* is introduced to deal with \forall quantifiers.

For unification equations of the form $\lambda \bar{x}. F \bar{s}_n \stackrel{?}{=} \lambda \bar{x}. a \bar{t}_m$ where $F : \alpha_1 \rightarrow \dots \rightarrow \alpha_n \rightarrow \beta$ and $a : \gamma_1 \rightarrow \dots \rightarrow \gamma_n \rightarrow \beta$, Vukmirović et al. allows the binding *Imitation of a for F* to be applied:

$$F \mapsto \lambda \bar{x}_n. a (F_1 \bar{x}_n) \dots (F_m \bar{x}_n).$$

Here \bar{F}_m are fresh free variables. This binding attempts to find a substitution for F such that the *head* of the two sides of the equations are identical.

Three issues are introduced when we take dependent type theory into account. Resolving these issues yields an imitation binding suitable for dependent type theory:

- F and a might be dependently typed. Hence, even when the number of λ binders in the LHS and RHS of the unification equation are different, solutions might still exist. For example, consider unification equation $\lambda x. F x \stackrel{?}{=} g U$ where F and U are free variables and $F : \gamma \rightarrow \gamma$, $U : \text{Type}$, and $g : \forall \alpha : \text{Type}. \alpha$. Note that both LHS and RHS are η -expanded, and that the number of λ binders are different. However, the substitution $\{F \mapsto g (\gamma \rightarrow \gamma), U \mapsto (\gamma \rightarrow \gamma)\}$ is clearly a solution to the unification equation. Thus, we need to extend the scope of the imitation binding to unification equations of form $\lambda \bar{x}_u. F \bar{s}_n \stackrel{?}{=} \lambda \bar{x}_v. a \bar{t}_m$, where u and v are not required to be equal.
- Since F and a might be dependently typed, their types should be written as

$$F : \forall (x_1 : \alpha_1) \dots (x_k : \alpha_k). \beta \bar{x}_k$$

$$a : \forall (y_1 : \gamma_1) \dots (y_l : \gamma_l). \delta \bar{y}_l$$

Again, we cannot assert that $k = n$ or $l = m$ because types can contain free variables. However, since both $\lambda \bar{x}_u. F \bar{s}_n$ and $\lambda \bar{x}_v. a \bar{t}_m$ are η -expanded, we know that $k \leq n$ and $l \leq m$.

- For $1 \leq i < j \leq n$, the type of the bound variable x_j can depend on x_i , hence the binding for F should be written as

$$F \mapsto \lambda (x_1 : \alpha_1) \dots (x_k : \alpha_k). a (F_1 \bar{x}_n) \dots (F_h \bar{x}_k)$$

Plugging the above binding into the unification equation and comparing the number of λ -binders on the two sides, we get $h + n - k - u = m - v$, hence $h = m + k + u - n - v$.

4.2 Inhabitation Reasoning

As noted in Section 3.1, sound proof reconstruction in a dependently typed setting requires reasoning about the presence of empty types. If any of a clause's universal quantifiers range over an empty type, then the clause is vacuously true and has an irrelevant clause body. One way we might respond to this issue is to observe that in many of the theorems that people actually care to prove, all of the nonpropositional types at play are known to be inhabited. In practice, it is often fine to have Duper temporarily assume that all nonpropositional types it encounters are inhabited and throw an error at the end if this results in an unsound inference. This is exactly what Duper does if passed in the option `inhabitationReasoning := False`.

When Duper is not passed in this option, it takes care throughout the reasoning process to ensure that empty types and potentially vacuous clauses are properly accounted for. In broad strokes, the differences in Duper's behavior when `inhabitationReasoning` is enabled or

disabled can be classified into two categories: there is the additional reasoning needed to determine whether any given type is inhabited, and there are the modifications to the main saturation loop needed to ensure that Duper correctly handles potentially vacuous clauses.

4.2.1 Determining Whether a Type Is Inhabited

Lean has a built-in type class `Inhabited` α that indicates that α is nonempty and supplies a witness attesting to this fact. When a type is known to be inhabited and is either built-in or part of the popular Mathlib library [29], it will typically already have an instance of this type class. Additionally, user-defined datatypes can often generate instances of this type class automatically with the syntax `deriving Inhabited`, and if a type is composed of inhabited types in certain preapproved ways, then Lean can automatically infer that the produced type is also inhabited. For example, from the instances `[Inhabited α]` and `[Inhabited β]`, type class inference will automatically yield `[Inhabited $(\alpha \times \beta)$]`.

So in most cases, when Duper encounters a type and needs to determine whether it is nonempty, Duper can simply make use of Lean's built-in type class inference. However, there are some goals, particularly those involving polymorphic types, for which this approach is insufficient. Consider the following example:

```
example (f :  $\alpha \rightarrow \alpha$ ) (h :  $\exists x : \alpha, f x \neq x$ ) :  $\exists x y : \alpha, x \neq y$  := by duper [h]
```

If we manually inspect this example, it is clear from hypothesis `h` that α must be inhabited, otherwise it would be impossible for there to exist a value x of type α . But Lean's type class inference on its own is insufficient to make use of this observation. In order for Duper to handle this example, it must explicitly note that $\exists x : \alpha, f x \neq x$ entails that α is inhabited.

More generally, when `inhabitationReasoning` is enabled, the simplification rules used to clausify quantifiers must do more than apply Skolemization to eliminate said quantifiers. We note that given a clause of the form $(\exists x : \alpha, P(x)) \vee R$, it is not always possible to generate a Skolem symbol of type α since α may not be inhabited. So instead, Duper generates a Skolem symbol `skS` of type $\alpha \rightarrow \alpha$, and produces the Skolemized clause $\forall z : \alpha, (P(\text{skS } z) \vee R)$. This approach is effective in pushing all quantifiers to the head of the clause, but it does not preserve the information that if R is false, then α must be inhabited. To ensure that this information remains derivable, the simplification rules Duper uses to clausify quantifiers are expanded from just `Clausify \exists 1` and `Clausify \forall 1` to all of the rules in Figure 1.

$$\frac{\forall x_1 : \alpha_1, \dots \forall x_n : \alpha_n, (\exists y : \beta, p) = \text{True} \vee R}{\forall x_1 : \alpha_1, \dots \forall x_n : \alpha_n, \forall z : \beta, p[y/(\text{skS } x_1 x_2 \dots x_n z)] = \text{True} \vee R} \text{Clausify}_{\exists 1}^{\exists}$$

$$\frac{\forall x_1 : \alpha_1, \dots \forall x_n : \alpha_n, (\exists y : \beta, p) = \text{True} \vee R}{\forall x_1 : \alpha_1, \dots \forall x_n : \alpha_n, \text{Nonempty } \beta = \text{True} \vee R} \text{Clausify}_{\exists 2}^{\exists}$$

$$\frac{\forall x_1 : \alpha_1, \dots \forall x_n : \alpha_n, (\forall y : \beta, p) = \text{False} \vee R}{\forall x_1 : \alpha_1, \dots \forall x_n : \alpha_n, \forall z : \beta, \neg p[y/(\text{skS } x_1 x_2 \dots x_n z)] = \text{True} \vee R} \text{Clausify}_{\forall 1}^{\forall}$$

$$\frac{\forall x_1 : \alpha_1, \dots \forall x_n : \alpha_n, (\forall y : \beta, p) = \text{False} \vee R}{\forall x_1 : \alpha_1, \dots \forall x_n : \alpha_n, \text{Nonempty } \beta = \text{True} \vee R} \text{Clausify}_{\forall 2}^{\forall}$$

■ **Figure 1** Quantifier clausification rules. The constant `skS` which appears in `Clausify \exists 1` and `Clausify \forall 1` is a fresh Skolem function of type $\alpha_1 \rightarrow \dots \rightarrow \alpha_n \rightarrow \beta \rightarrow \beta$.

Note that although some preprocessing can be done to extract inhabited types from a goal state's original hypotheses, it is not possible in general to derive all type inhabitation information at the preprocessing stage. As an example, if a goal state contains the hypothesis

$(\exists x : \alpha, P) \vee Q$, then after this hypothesis is classified to $(\exists x : \alpha, P) = \text{True} \vee Q = \text{True}$, $\text{Classify}_{\frac{3}{2}}$ can be used to derive $\text{Nonempty } \alpha = \text{True} \vee Q = \text{True}$. But this does not yet settle the question of whether α is actually inhabited. Until Q is refuted, which may not happen until a later stage in the reasoning process, Duper cannot use this clause to determine α 's inhabitation status. The fact that type inhabitation information must be derived throughout the reasoning process, rather than all at once during preprocessing, has consequences on the structure of the main saturation loop that are discussed in Section 4.2.2.

4.2.2 Modifications to the Main Saturation Loop

The main saturation loop described in Section 2.1 accurately reflects Duper's behavior when `inhabitationReasoning` is disabled. As long as all types are known or assumed to be inhabited, a dependently typed setting does not require any substantive changes from this procedure. However, when there are potentially vacuous clauses, meaning clauses with leading universal quantifiers whose types are possibly empty, the approach described in Section 2.1 can cause Duper to remove or ignore clauses that are necessary to obtain a contradiction.

To avoid removing or ignoring clauses that are necessary to obtain a contradiction, Duper must refrain from using potentially vacuous clauses to simplify away nonvacuous clauses. However, Duper cannot avoid reasoning about potentially vacuous clauses entirely, as they may be discovered to be nonvacuous at a later stage in the reasoning process. Additionally, Duper cannot fully defer reasoning about potentially vacuous clauses until they are determined to be nonvacuous because they may be needed to derive certain type inhabitation facts. For example, even if α is a possibly empty type, it may be necessary to reason about the clause $\forall x : \alpha, \text{Nonempty } \beta = \text{True} \vee P$ because if P can be refuted, then the resulting clause will entail that $\alpha \rightarrow \beta$ is nonempty.

To satisfy these various requirements, when `inhabitationReasoning` is enabled, Duper adopts an alternative given clause procedure that has been modified in the following ways:

- When a given clause is identified as potentially vacuous, forward simplification rules and inference rules are still applied to it, but it is not used for backward simplification rules.
- When a potentially vacuous clause is added to the active set, it is still added to the data structures that allow it to be retrieved for inference rules and backward simplification rules. However, it is not added to any of the data structures that would enable it to be used in future forward simplification rules.
- After any clause is fully simplified, a check is run to see if any type inhabitation facts can be derived from the clause.
- When a new type is discovered to be nonempty, the set of clauses that Duper classifies as potentially vacuous is revisited and updated.
- When a clause previously classified as potentially vacuous is discovered to be nonvacuous, it is immediately used for backward simplification rules and is subsequently added to the data structures that enable it to be used in future forward simplification rules.

4.3 Universe Levels

Some problems require Duper to reason about universe polymorphic theorems and inductive types with universe level parameters. In many cases, these complications can be eliminated at the preprocessing stage by way of the monomorphization procedure described in Section 5. When feasible, this is Duper's preferred method of addressing universe polymorphism. Unfortunately, the procedure described in Section 5 cannot be applied to all problems. Even when it can be used, it is still possible for some of Duper's higher-order inference rules to

produce clauses containing universe polymorphic constants such as the equality and inequality predicates $\text{@Eq}\{u\}$ and $\text{@Ne}\{u\}$. So Duper requires the ability to carry out some universe polymorphic reasoning natively.

During the main saturation loop, there are three areas where universe levels are relevant. First, Duper’s unification procedure must take universe levels into account. For the most part, this only requires calling Lean’s built-in level unifier in cases where a level metavariable determines whether two sorts or constants are unifiable. Second, converting between clauses and mclauses requires interchanging parameter names in clauses with level metavariables in mclauses. Finally, the Skolem symbols that Duper generates in Figure 1’s $\text{Clausify}_1^{\exists}$ and $\text{Clausify}_1^{\forall}$ are universe polymorphic, so some machinery is needed both to generate the initial Skolem symbols and to keep track of their parameters across inferences.

During proof reconstruction, Duper must occasionally instantiate universe polymorphic facts with specific universe levels. In some cases, Duper must even instantiate the same fact multiple times with different universe levels. Consider the following example:

```
theorem singletonListNotEmpty.{u} : ∀ α : Type u, ∀ z : α, ¬[z].isEmpty := ...

example (t1 : Type 1) (t2 : Type 2) (x : t1) (y : t2) :
  ¬[x].isEmpty ∧ ¬[y].isEmpty := by duper [singletonListNotEmpty]
```

In this example, Duper must instantiate `singletonListNotEmpty` with universe level 1 so that `¬[x].isEmpty` can be derived and with universe level 2 so that `¬[y].isEmpty` can be derived. During proof reconstruction, Duper determines this by examining the children clauses generated via inferences involving `singletonListNotEmpty`. If these clauses have specific universe levels instead of `singletonListNotEmpty`’s universe variable, then those universe levels determine how `singletonListNotEmpty` is instantiated. If the children clauses are also universe polymorphic, then their children clauses are recursively examined until a clause without `singletonListNotEmpty`’s universe variable is found. This is guaranteed to happen because repeatedly examining children will inevitably lead to the empty clause. The information concerning how universe levels must be instantiated can then be propagated from children to parents until it is known how to instantiate each of Duper’s universe polymorphic clauses.

5 Monomorphization

An advantage to using dependent type theory as a foundation is that it provides powerful mechanisms to support algebraic reasoning. When a user types an expression like $x + y$ in a context where x and y are inferred to have type `Nat`, the expression is parsed as `HAdd.hAdd Nat _`, where the third argument, represented here by an underscore, is expected to be an *instance* of a type class for the addition notation. Lean will synthesize a suitable value, such as `@instHAdd Nat AddSemigroup.toAdd`, by searching through a database of instances that have been registered with the system. Type classes are similarly used to handle algebraic structures and generic theorems about them.

Type class inference is at odds with automated reasoning in a number of respects. First, it renders expressions quite verbose, increasing processing time and storage during search. Second, searching for instances is generally too expensive to carry out in Duper’s main saturation loop. Third, two fully elaborated instances of an expression like $x + y$ may be only *definitionally equal* rather than syntactically equal, which means that Lean can determine they are the same only by unfolding definitions and simplifying them. This can happen when the system infers implicit arguments, like the addition function that is appropriate to the natural numbers, in different ways. Testing equivalence of terms up to definitional equality is also generally too expensive to carry out during the main saturation loop.

When using Duper as a tactic in Lean, we therefore rely on a preprocessing phase, implemented in a separate tool called LeanAuto,¹ that does all of the following:

- abstracts the proof goal as a smaller higher-order problem with dependently typed parameters that are kept separate;
- heuristically instantiates types and type classes in generic lemmas in the context; and
- identifies definitionally equal expressions with a single representative.

We refer to this as *monomorphization* although that term is more properly used to describe the second component above. The monomorphization procedure is complex and will be described in greater detail elsewhere. Here we sketch the main ideas.

5.1 Reduction to Essentially Higher-Order Problems

The first observation is that many goals that arise in Lean are *essentially higher-order* validities. For example, suppose we have variables $n : \text{Nat}$ and $a\ b\ c : \text{Fin } n$, where $\text{Fin } n$ is the (dependent) type of values less than n . Suppose further that we are trying to prove

$$a + (b + c) = (c + b) + a$$

from

$$\forall x\ y : \text{Fin } n, x + y = y + x.$$

If we abstract $\text{Fin } n$ to a generic type α , we get a first-order problem of which our original goal is an instance.

► **Definition 1.** Let φ be a proposition in Lean. φ is an *essentially higher-order validity* iff there exists a valid higher-order formula ψ and a mapping σ such that:

- $\sigma(\psi) = \varphi$.
- σ maps free variables to expressions in Lean.
- σ maps constants, function symbols and type symbols to closed expressions in Lean. Moreover, we require that the equality symbol $=$ in higher-order logic is mapped to $=$ in Lean.
- σ is type consistent, i.e. for each constant, function symbol, type symbol, or free variable x of type T , $\sigma(x)$ is of type $\sigma(T)$.

A Lean goal $h_1 : T_1, h_2 : T_2, \dots, h_n : T_n \vdash T$ is *essentially higher(first)-order* iff $\forall (h_1 : T_1) (h_2 : T_2) \dots (h_n : T_n), T$ is *essentially higher(first)-order*.

Given a Lean goal $h_1, h_2, \dots, h_n \vdash \text{False}$, our monomorphization procedure will instantiate the quantifiers of the premises and construct a new goal G with the resulting instances as premises and False as conclusion. Then, it attempts to find a higher-order formula ψ and a substitution σ such that $\sigma(\psi)$ is equivalent to G . If successful, ψ is passed to the main saturation loop. Finally, if the saturation loop manages to find a proof of ψ , a proof of G will be reconstructed using σ .

5.2 Type Classes and Definitional Equality

Our monomorphization procedure implements a mechanism to check whether functions with different type class instance arguments are definitionally equal. For each function f with such arguments, the monomorphization procedure keeps a list of mutually (definitionally)

¹ <https://github.com/leanprover-community/lean-auto>

unequal expressions of the form $f \bar{t}$. Each time a new expression e of the form $f \bar{t}$ is found by the monomorphization procedure, it compares e to the previously recorded expressions associated with f and checks whether they are definitionally equal. This mechanism cannot recognize all definitional equalities in Lean, since expressions with different heads can also be definitionally equal. However, this approach is effective in handling the sorts of type classes that tend to appear in Mathlib.

6 Evaluation

In this section, we evaluate Duper’s performance on first-order and higher-order benchmarks in the TPTP format [43]. Duper requires a Lean goal state as input, so to run Duper on these benchmarks, we wrote a parser to convert TPTP problems into Lean goals. Said parser is included in the Duper repository. The goal of our evaluation is to answer the following:

1. How does Duper compare against other automatic theorem provers? In particular, how does Duper compare against Metis, an automatic theorem prover frequently used for Sledgehammer’s proof reconstruction? This is a metric for Duper’s current strength as a general automatic theorem prover.
2. What is the impact of using external automation to minimize benchmarks on Duper’s performance? And is this impact dependent on the nature of the external automation? This is a metric for Duper’s current potential as proof reconstruction for a future hammer.
3. Some of Duper’s rules are marked as expensive based on their behavior in Zipperposition. What is the impact of enabling or disabling expensive rules on Duper’s performance? And is this impact dependent on whether Duper is given a problem in a first-order or higher-order format? This is a metric for determining in which circumstances Duper’s expensive rules should be enabled.

6.1 Experimental Methodology

6.1.1 Benchmarks

We borrow benchmarks from *Seventeen Provers under the Hammer* [16] and *GRUNGE: A Grand Unified ATP Challenge* [11]. Both of these sources provide TPTP benchmarks in multiple formats, so we evaluate on the same set of benchmarks translated to a first-order form (FOF) and a typed higher-order form (THF). Specifically, we use the TH0⁻ encodings from the *Seventeen* benchmarks and TH0-II encodings from the *GRUNGE* benchmarks.

The *Seventeen* benchmarks contain multiple versions of each problem with different numbers of additional facts supplied by Sledgehammer. We test on the same *Seventeen* benchmarks with 16 facts included and 256 facts included. The *GRUNGE* benchmarks are not duplicated in this manner, so for these, we just test on the available benchmarks in their “bushy” format, meaning the format containing exactly the facts needed to prove the original HOL4 [38] theorems from which the benchmarks were generated.

6.1.2 Provers

In addition to Duper, we evaluate the benchmarks with Metis, Zipperposition, and Vampire [25]. Metis was selected as the standard for Sledgehammer-style proof reconstruction, Zipperposition was selected as a powerful automatic theorem prover that implements the same core calculus as Duper, and Vampire was selected as a powerful automatic theorem prover that implements a different core calculus from Duper.

We use Duper version v0.0.9, Metis version 2.4, Zipperposition version 2.1, and Vampire version 4.6.1. All provers except Duper are given arguments indicating they have a 30 second timeout, and all provers are externally terminated after 30 seconds of wall-clock time. Duper is run both with and without expensive rules enabled. Duper (–) is used to refer to Duper without expensive rules and Duper (+) is used to refer to Duper with “expensive” rules. Metis is run with default settings, while Vampire and Zipperposition are run with options replicating those in the *Seventeen* paper as closely as possible. Specifically, Vampire is run in portfolio mode with the “casc” schedule for FOF problems and the “casc_hol_2020” schedule for THF problems, while Zipperposition is run with the script `portfolio.lams.parallel.py` available in Zipperposition’s repository. We note that the Python code used to run Zipperposition’s portfolio mode on multiple cores is not Mac-compatible, so Zipperposition is only able to use one core of the computer used to run these experiments. Vampire’s portfolio mode code does not have such issues, so it was run unaltered. The computer used to run these experiments is a Mac with a 3.8GHz processor and 16GB of RAM.

6.2 Experimental Results

■ **Table 1** Original benchmark problems solved.

	FOF Format			THF Format		
	Seventeen (16 Facts)	Seventeen (256 Facts)	GRUNGE (Bushy)	Seventeen (16 Facts)	Seventeen (256 Facts)	GRUNGE (Bushy)
Duper (–)	1176/5000	1086/5000	64/1000	1111/5000	934/5000	231/1000
Duper (+)	1167/5000	1050/5000	64/1000	997/5000	670/5000	78/1000
Metis	1195/5000	1120/5000	202/1000	–	–	–
Vampire	1285/5000	2521/5000	262/1000	1331/5000	2341/5000	459/1000
Zipperposition	1277/5000	2209/5000	277/1000	1314/5000	2122/5000	354/1000

Table 1 shows the number of original benchmark problems solved by Duper, Metis, Vampire, and Zipperposition when given a 30 second timeout. Vampire and Zipperposition outperform Metis by a significant margin, especially as more facts are made available, and Metis outperforms Duper by a slimmer margin. We note that except for Metis, which only accepts FOF problems, all provers perform significantly better on *GRUNGE* THF problems than *GRUNGE* FOF problems. The difference in results between *GRUNGE*’s formats is significantly greater than the difference in results between *Seventeen*’s formats. This is likely explained by the fact that the *GRUNGE* benchmarks use a more inefficient encoding of polymorphism into the FOF format, apparently resulting in harder FOF problems [16].

■ **Table 2** Vampire-minimized Seventeen benchmark problems solved.

	FOF Format		THF Format	
	16 Facts	256 Facts	16 Facts	256 Facts
Duper (–)	1246/1285	2372/2521	1214/1331	2121/2341
Duper (+)	1244/1285	2368/2521	1149/1331	2025/2341
Metis	1268/1285	2382/2521	–	–

Tables 2 and 3 show the number of *Seventeen* benchmark problems that Duper and Metis can solve after they are minimized by Vampire and Zipperposition respectively. Minimized benchmark problems are generated by removing all axioms that do not appear in the proofs

■ **Table 3** Zipperposition-minimized Seventeen benchmark problems solved.

	FOF Format		THF Format	
	16 Facts	256 Facts	16 Facts	256 Facts
Duper (–)	1249/1277	2169/2209	1231/1314	2061/2122
Duper (+)	1248/1277	2166/2209	1214/1314	2014/2122
Metis	1267/1277	2165/2209	–	–

output by Vampire and Zipperposition. Thus, a minimized benchmark problem can only be generated if Vampire or Zipperposition solved the original benchmark problem. We only test on minimized *Seventeen* benchmark problems because the original *GRUNGE* benchmark problems already include exactly the axioms necessary to solve them.

Although Metis observably outperforms Duper on original benchmark problems, neither Duper nor Metis significantly outperforms the other on minimized problems. For both Vampire-minimized problems and Zipperposition-minimized problems, there is less than a 2% disparity in the number of reconstructed 16-Facts FOF problems, and less than a 1% disparity in the number of reconstructed 256-Facts FOF problems. Duper appears to do slightly better reconstructing Zipperposition’s proofs than Vampire’s, but the difference is marginal. We take this as a evidence that Metis’ and Duper’s abilities to reconstruct proofs generated by external provers are comparable.

In all benchmark categories, Duper performs better on average with its expensive rules disabled. We note that there are no FOF benchmarks that Duper can only solve with its expensive rules enabled, but there are 46 THF problems across the various categories that Duper requires expensive rules to solve. The fact that expensive rules appear to have some benefit for THF problems but no benefit for FOF problems is explained by the fact that most of Duper’s expensive rules are higher-order.

Overall, we conclude that on raw TPTP problems, Metis performs slightly better than Duper, but that the two tools perform extremely similarly on problems minimized by a more powerful external prover. We note that this evaluation of Duper’s performance is restricted to first-order and higher-order problems. Ideally, the features Duper implements that are oriented toward reasoning in a dependently typed setting would be evaluated using benchmarks from Mathlib, but until Duper is equipped with a relevance filter, such an evaluation is not feasible.

7 Related Work

Section 6 provides a quantitative evaluation comparing Duper against other automatic theorem provers. In this section, we discuss some of the qualitative differences between Duper and other general-purpose proof automation in various interactive theorem provers.

In Coq, the tactic most similar to Duper in purpose is `sauto` [13]. Just as Duper was designed with proof reconstruction for a future Lean hammer in mind, `sauto` was created to be CoqHammer’s [14] proof reconstruction procedure of choice. The primary difference between Duper and `sauto` is that Duper produces classical proofs and `sauto` produces constructive proofs. While Duper’s initial goal transformation and underlying superposition calculus are only sound classically, `sauto`’s direct search for type inhabitants in an appropriate normal form is fundamentally intuitionistic. For many Lean users, the benefit of being able to prove more facts outweighs the benefit of staying in Lean’s intuitionistic fragment, since many Lean users are formalizing classical mathematics in any case. On the other hand, many Coq

users are firmly committed to constructive proofs and their computational interpretations, so it makes sense that there would be a greater desire for Coq’s general-purpose automation to remain intuitionistic, even if it means fewer problems can be solved.

In Lean 3, the tactic most similar to Duper is Super.² Super is a prototype proof-producing superposition theorem prover implemented with Lean 3’s metaprogramming language. Due to inherent limits of Lean 3 metaprogramming, Super was not performant enough to make it into common use, but it was nonetheless an important proof of concept. The name “Duper” is an homage to that project.

In Lean 4, the most notable general-purpose proof automation tactic currently available is Aesop [26]. The primary difference between Duper and Aesop is that Duper is designed to require as little user input as possible, whereas Aesop is designed to be a highly customizable white-box automation tactic. There are benefits to both approaches. Aesop’s white-box approach gives users more control over how the proof search is performed, while Duper’s black-box approach is more amenable to push-button automation and has a lower barrier to entry for users. Overall, we hope that Duper and Aesop will prove to be complementary tactics well suited to different use cases.

In Isabelle/HOL, HOL4, and HOL Light, the proof automation most similar to Duper is Metis. Although the evaluation given in Section 6 is useful for gauging Metis’ and Duper’s relative performances, it is limited in that it considers only TPTP problems that can be expressed in both first- and higher-order logic. A fact that the evaluation does not capture is that there is a class of problems accessible to Duper but not Metis. This is the class of problems that fundamentally require native higher-order reasoning.

Even though Metis is a first-order prover, it can still solve some higher-order problems by first translating them into first-order logic. For many problems, this approach is sufficient, as evidenced by Metis’ frequent use in higher-order interactive theorem provers. However, as noted in *Mechanical Mathematics* [4], when problems critically involve higher-order constructions, their first-order translations can quickly become intractable. Consider the following problem which is presented both as an Isabelle/HOL lemma and as a Lean example:

```
lemma "( $\sum i::\text{nat}=0..n. i$ ) + ( $\sum i::\text{nat}=0..n. i$ ) = ( $\sum i::\text{nat}=0..n. i + i$ )"
  by (metis sum.distrib)

example :  $\sum i$  in range n, i +  $\sum i$  in range n, i =  $\sum i$  in range n, (i + i) :=
  by duper [Finset.sum_add_distrib]
```

When this problem is given to Isabelle/HOL’s Sledgehammer, Zipperposition is able to find a proof and suggests the above Metis call. However, Metis is unable to reconstruct Zipperposition’s proof because the summation notation involved is significantly more complex when encoded in first-order logic. On the other hand, Duper has no issue with the equivalent Lean example because it does not need to translate the problem into a less expressive logic.

8 Conclusion and Future Work

We have presented Duper, a proof-producing superposition theorem prover for Lean. Duper’s underlying approach to proof search adapts classical methods in automatic theorem proving to a dependently typed setting using a flexible combination of preprocessing and native reasoning. Since Duper directly generates proofs in Lean’s axiomatic foundation, it can be called as a terminal tactic in interactive Lean proofs.

² <https://github.com/leanprover/super>

In the future, we hope to use Duper for proof reconstruction in a Lean hammer. Equipping Duper with a relevance filter and integrating Duper into a full hammer pipeline will drastically increase Duper’s usefulness. The experimental results given in Section 6.2 show that Duper is performant enough to reconstruct proofs from a variety of state-of-the-art automatic theorem provers.

References

- 1 Jürgen Avenhaus, Jörg Denzinger, and Matthias Fuchs. DISCOUNT: A system for distributed equational deduction. In Jieh Hsiang, editor, *Rewriting Techniques and Applications, 6th International Conference, RTA-95, Kaiserslautern, Germany, April 5-7, 1995, Proceedings*, volume 914 of *Lecture Notes in Computer Science*, pages 397–402. Springer, 1995. doi: 10.1007/3-540-59200-8_72.
- 2 Jeremy Avigad, Leonardo de Moura, Soonho Kong, and Sebastian Ullrich. Theorem proving in Lean 4. URL: https://leanprover.github.io/theorem_proving_in_lean4/.
- 3 Leo Bachmair and Harald Ganzinger. Rewrite-based equational theorem proving with selection and simplification. *J. Log. Comput.*, 4(3):217–247, 1994. doi:10.1093/LOGCOM/4.3.217.
- 4 Alexander Bentkamp, Jasmin Blanchette, Visa Nummelin, Sophie Tourret, Petar Vukmirović, and Uwe Waldmann. Mechanical mathematicians. *Communications of the ACM*, 66(4):80–90, March 2023. doi:10.1145/3557998.
- 5 Alexander Bentkamp, Jasmin Blanchette, Sophie Tourret, and Petar Vukmirovic. Superposition for higher-order logic. *J. Autom. Reason.*, 67(1):10, 2023. doi:10.1007/S10817-022-09649-9.
- 6 Alexander Bentkamp, Jasmin Blanchette, Sophie Tourret, Petar Vukmirovic, and Uwe Waldmann. Superposition with lambdas. *J. Autom. Reason.*, 65(7):893–940, 2021. doi: 10.1007/S10817-021-09595-Y.
- 7 Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development – Coq’Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004. doi:10.1007/978-3-662-07964-5.
- 8 Jasmin Blanchette, Qi Qiu, and Sophie Tourret. Verified given clause procedures. In Brigitte Pientka and Cesare Tinelli, editors, *Automated Deduction - CADE 29 - 29th International Conference on Automated Deduction, Rome, Italy, July 1-4, 2023, Proceedings*, volume 14132 of *Lecture Notes in Computer Science*, pages 61–77. Springer, 2023. doi:10.1007/978-3-031-38499-8_4.
- 9 Jasmin C. Blanchette, Cezary Kaliszyk, Lawrence C. Paulson, and Josef Urban. Hammering towards QED. *J. Formaliz. Reason.*, 9(1):101–148, 2016. doi:10.6092/ISSN.1972-5787/4593.
- 10 Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of Agda – A functional language with dependent types. 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 73–78. Springer, 2009. doi:10.1007/978-3-642-03359-9_6.
- 11 Chad E. Brown, Thibault Gauthier, Cezary Kaliszyk, Geoff Sutcliffe, and Josef Urban. GRUNGE: A grand unified ATP challenge. In Pascal Fontaine, editor, *Automated Deduction - CADE 27 - 27th International Conference on Automated Deduction, Natal, Brazil, August 27-30, 2019, Proceedings*, volume 11716 of *Lecture Notes in Computer Science*, pages 123–141. Springer, 2019. doi:10.1007/978-3-030-29436-6_8.
- 12 Mario Carneiro, Chad E. Brown, and Josef Urban. Automated theorem proving for metamath. In Adam Naumowicz and René Thiemann, editors, *14th International Conference on Interactive Theorem Proving, ITP 2023, July 31 to August 4, 2023, Białystok, Poland*, volume 268 of *LIPICs*, pages 9:1–9:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi: 10.4230/LIPICs.ITP.2023.9.
- 13 Lukasz Czajka. Practical proof search for Coq by type inhabitation. In Nicolas Peltier and Viorica Sofronie-Stokkermans, editors, *Automated Reasoning - 10th International Joint Conference*,

- IJCAR 2020, Paris, France, July 1-4, 2020, Proceedings, Part II*, volume 12167 of *Lecture Notes in Computer Science*, pages 28–57. Springer, 2020. doi:10.1007/978-3-030-51054-1_3.
- 14 Lukasz Czajka and Cezary Kaliszyk. Hammer for coq: Automation for dependent type theory. *J. Autom. Reason.*, 61(1-4):423–453, 2018. doi:10.1007/S10817-018-9458-4.
 - 15 Leonardo de Moura and Sebastian Ullrich. The lean 4 theorem prover and programming language. In André Platzer and Geoff Sutcliffe, editors, *Automated Deduction - CADE 28 - 28th International Conference on Automated Deduction, Virtual Event, July 12-15, 2021, Proceedings*, volume 12699 of *Lecture Notes in Computer Science*, pages 625–635. Springer, 2021. doi:10.1007/978-3-030-79876-5_37.
 - 16 Martin Desharnais, Petar Vukmirovic, Jasmin Blanchette, and Makarius Wenzel. Seventeen provers under the hammer. In June Andronick and Leonardo de Moura, editors, *13th International Conference on Interactive Theorem Proving, ITP 2022, August 7-10, 2022, Haifa, Israel*, volume 237 of *LIPICs*, pages 8:1–8:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPICs.ITP.2022.8.
 - 17 Jeanne Ferrante and Charles Rackoff. A decision procedure for the first order theory of real addition with order. *SIAM J. Comput.*, 4(1):69–76, 1975. doi:10.1137/0204006.
 - 18 Simon Foster and Georg Struth. Integrating an automated theorem prover into Agda. In Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings*, volume 6617 of *Lecture Notes in Computer Science*, pages 116–130. Springer, 2011. doi:10.1007/978-3-642-20398-5_10.
 - 19 Benjamin Grégoire and Assia Mahboubi. Proving equalities in a commutative ring done right in Coq. In Joe Hurd and Thomas F. Melham, editors, *Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005, Oxford, UK, August 22-25, 2005, Proceedings*, volume 3603 of *Lecture Notes in Computer Science*, pages 98–113. Springer, 2005. doi:10.1007/11541868_7.
 - 20 John Harrison. Optimizing proof search in model elimination. In Michael A. McRobbie and John K. Slaney, editors, *Automated Deduction - CADE-13, 13th International Conference on Automated Deduction, New Brunswick, NJ, USA, July 30 - August 3, 1996, Proceedings*, volume 1104 of *Lecture Notes in Computer Science*, pages 313–327. Springer, 1996. doi:10.1007/3-540-61511-3_97.
 - 21 Joe Hurd. First-order proof tactics in higher-order logic theorem provers. In *Design and Application of Strategies/Tactics in Higher Order Logics (STRATA 2003)*, pages 56–68, 2003. URL: <http://www.gilith.com/papers>.
 - 22 Cezary Kaliszyk and Josef Urban. Hol(y)hammer: Online ATP service for HOL light. *Math. Comput. Sci.*, 9(1):5–22, 2015. doi:10.1007/S11786-014-0182-0.
 - 23 Cezary Kaliszyk and Josef Urban. Mizar 40 for mizar 40. *J. Autom. Reason.*, 55(3):245–256, 2015. doi:10.1007/S10817-015-9330-8.
 - 24 Donald E. Knuth and Peter B. Bendix. Simple word problems in universal algebras. In John Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon, 1970. doi:10.1016/b978-0-08-012975-4.50028-x.
 - 25 Laura Kovács and Andrei Voronkov. First-order theorem proving and vampire. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, volume 8044 of *Lecture Notes in Computer Science*, pages 1–35. Springer, 2013. doi:10.1007/978-3-642-39799-8_1.
 - 26 Jannis Limperg and Asta Halkjær From. Aesop: White-box best-first proof search for lean. In Robbert Krebbers, Dmitriy Traytel, Brigitte Pientka, and Steve Zdancewic, editors, *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2023, Boston, MA, USA, January 16-17, 2023*, pages 253–266. ACM, 2023. doi:10.1145/3573105.3575671.


- 27 Bernd Löchner. Things to know when implementing KBO. *J. Autom. Reason.*, 36(4):289–310, 2006. doi:10.1007/S10817-006-9031-4.
- 28 Ewing L. Lusk. Controlling redundancy in large search spaces: Argonne-style theorem proving through the years. In Andrei Voronkov, editor, *Logic Programming and Automated Reasoning, International Conference LPAR'92, St. Petersburg, Russia, July 15-20, 1992, Proceedings*, volume 624 of *Lecture Notes in Computer Science*, pages 96–106. Springer, 1992. doi:10.1007/BFB0013052.
- 29 The mathlib Community. The Lean mathematical library. In Jasmin Blanchette and Catalin Hritcu, editors, *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020*, pages 367–381. ACM, 2020. doi:10.1145/3372885.3373824.
- 30 William McCune and Larry Wos. Otter—the CADE-13 competition incarnations. *Journal of Automated Reasoning*, 18(2):211–220, 1997. doi:10.1023/a:1005843632307.
- 31 Jia Meng and Lawrence C. Paulson. Translating higher-order clauses to first-order clauses. *J. Autom. Reason.*, 40(1):35–60, 2008. doi:10.1007/S10817-007-9085-Y.
- 32 Jia Meng and Lawrence C. Paulson. Lightweight relevance filtering for machine-generated resolution problems. *J. Appl. Log.*, 7(1):41–57, 2009. doi:10.1016/J.JAL.2007.07.004.
- 33 Tobias Nipkow. Term rewriting and beyond - theorem proving in Isabelle. *Formal Aspects Comput.*, 1(4):320–338, 1989. doi:10.1007/BF01887212.
- 34 Lawrence C. Paulson. A generic tableau prover and its integration with Isabelle. *J. Univers. Comput. Sci.*, 5(3):73–87, 1999. doi:10.3217/JUCS-005-03-0073.
- 35 Lawrence C. Paulson and Jasmin Christian Blanchette. Three years of experience with sledgehammer, a practical link between automatic and interactive theorem provers. In Geoff Sutcliffe, Stephan Schulz, and Eugenia Ternovska, editors, *The 8th International Workshop on the Implementation of Logics, IWIL 2010, Yogyakarta, Indonesia, October 9, 2011*, volume 2 of *EPiC Series in Computing*, pages 1–11. EasyChair, 2010. doi:10.29007/36DT.
- 36 Lawrence C. Paulson and Kong Woei Susanto. Source-level proof reconstruction for interactive theorem proving. In Klaus Schneider and Jens Brandt, editors, *Theorem Proving in Higher Order Logics, 20th International Conference, TPHOLs 2007, Kaiserslautern, Germany, September 10-13, 2007, Proceedings*, volume 4732 of *Lecture Notes in Computer Science*, pages 232–245. Springer, 2007. doi:10.1007/978-3-540-74591-4_18.
- 37 William W. Pugh. The test: a fast and practical integer programming algorithm for dependence analysis. In Joanne L. Martin, editor, *Proceedings Supercomputing '91, Albuquerque, NM, USA, November 18-22, 1991*, pages 4–13. ACM, 1991. doi:10.1145/125826.125848.
- 38 Konrad Slind and Michael Norrish. A brief overview of HOL4. In Otmane Ait Mohamed, César A. Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings*, volume 5170 of *Lecture Notes in Computer Science*, pages 28–32. Springer, 2008. doi:10.1007/978-3-540-71067-7_6.
- 39 Robert Solovay, R. D. Arthan, and John Harrison. Some new results on decidability for elementary algebra and geometry. *Ann. Pure Appl. Log.*, 163(12):1765–1802, 2012. doi:10.1016/J.APAL.2012.04.003.
- 40 G. Sutcliffe. The CADE-27 Automated Theorem Proving System Competition - CASC-27. *AI Communications*, 32(5-6):373–389, 2020.
- 41 G. Sutcliffe. The 10th IJCAR Automated Theorem Proving System Competition - CASC-J10. *AI Communications*, 34(2):163–177, 2021.
- 42 G. Sutcliffe and M. Desharnais. The CADE-28 Automated Theorem Proving System Competition - CASC-28. *AI Communications*, 34(4):259–276, 2022.
- 43 Geoff Sutcliffe. The logic languages of the TPTP world. *Log. J. IGPL*, 31(6):1153–1169, 2023. doi:10.1093/JIGPAL/JZAC068.
- 44 Geoff Sutcliffe and Martin Desharnais. The 11th IJCAR automated theorem proving system competition - CASC-J11. *AI Commun.*, 36(2):73–91, 2023. doi:10.3233/AIC-220244.

- 45 Petar Vukmirović, Alexander Bentkamp, Jasmin Blanchette, Simon Cruanes, Visa Nummelin, and Sophie Tourret. *Making Higher-Order Superposition Work*, pages 415–432. Springer International Publishing, 2021. doi:10.1007/978-3-030-79876-5_24.
- 46 Petar Vukmirović, Alexander Bentkamp, and Visa Nummelin. Efficient full higher-order unification. *Logical Methods in Computer Science*, Volume 17, Issue 4, December 2021. doi:10.46298/lmcs-17(4:18)2021.

A Formalization of the General Theory of Quaternions

Thaynara Arielly de Lima ✉ 

Universidade Federal de Goiás, Goiânia, Brazil

André Luiz Galdino ✉ 

Universidade Federal de Catalão, Catalão, Brazil

Bruno Berto de Oliveira Ribeiro

Universidade de Brasília, Brasília D.F., Brazil

Mauricio Ayala-Rincón ✉ 

Universidade de Brasília, Brasília D.F., Brazil

Abstract

This paper discusses the formalization of the theory of quaternions in the Prototype Verification System (PVS). The general approach in this mechanization relies on specifying quaternion structures using any arbitrary field as a parameter. The approach allows the inheritance of formalized properties on quaternions when the parameters of the general theory are instantiated with specific fields such as reals or rationals. The theory includes characterizing algebraic properties that lead to constructing quaternions as division rings. In particular, we illustrate how the general theory is applied to formalize Hamilton's quaternions using the field of reals as a parameter, for which we also mechanized theorems that show the completeness of three-dimensional rotations, proving that Hamilton's quaternions mimic any 3D rotation.

2012 ACM Subject Classification Computing methodologies → Symbolic and algebraic manipulation; Theory of computation → Automated reasoning; Theory of computation → Logic and verification

Keywords and phrases Theory of quaternions, Hamilton's quaternions, Algebraic formalizations, PVS

Digital Object Identifier 10.4230/LIPIcs.ITP.2024.11

Funding *Thaynara Arielly de Lima*: Project supported by FAPEG 202310267000223.

Mauricio Ayala-Rincón: Project supported by FAPDF DE 00193.00001175/21-11 and CNPq Universal 409003/21-2 grants. The author was partially funded by the CNPq grant 313290/21-0.

1 Introduction

Quaternions can be identified with the general theory of algebraic structures consisting of quadruples built over a field, $\langle \mathbb{F}, +_{\mathbb{F}}, *_{\mathbb{F}}, \text{zero}_{\mathbb{F}}, \text{one}_{\mathbb{F}} \rangle$ and two selected elements of the field $a, b \in \mathbb{F}$, where the quaternion addition is built from the field addition component to component, and the product quaternion is a distributive product, that satisfies a series of axioms, including

$$(\text{zero}_{\mathbb{F}}, \text{one}_{\mathbb{F}}, \text{zero}_{\mathbb{F}}, \text{zero}_{\mathbb{F}})^2 = (a, \text{zero}_{\mathbb{F}}, \text{zero}_{\mathbb{F}}, \text{zero}_{\mathbb{F}})$$

$$(\text{zero}_{\mathbb{F}}, \text{zero}_{\mathbb{F}}, \text{one}_{\mathbb{F}}, \text{zero}_{\mathbb{F}})^2 = (b, \text{zero}_{\mathbb{F}}, \text{zero}_{\mathbb{F}}, \text{zero}_{\mathbb{F}})$$

$$(\text{zero}_{\mathbb{F}}, \text{zero}_{\mathbb{F}}, \text{one}_{\mathbb{F}}, \text{zero}_{\mathbb{F}}) * (\text{zero}_{\mathbb{F}}, \text{one}_{\mathbb{F}}, \text{zero}_{\mathbb{F}}, \text{zero}_{\mathbb{F}}) = (\text{zero}_{\mathbb{F}}, \text{zero}_{\mathbb{F}}, \text{zero}_{\mathbb{F}}, \text{one}_{\mathbb{F}})$$

among others, from which all properties of addition and multiplication of quaternions are inferred. In general, given a field \mathbb{F} , and elements $a, b \in \mathbb{F}$, the quaternion algebra is



© Thaynara Arielly de Lima, André Luiz Galdino, Bruno Berto de Oliveira Ribeiro, and Mauricio Ayala-Rincón;

licensed under Creative Commons License CC-BY 4.0

15th International Conference on Interactive Theorem Proving (ITP 2024).

Editors: Yves Bertot, Temur Kutsia, and Michael Norrish; Article No. 11; pp. 11:1–11:18



Leibniz International Proceedings in Informatics

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

11:2 A Formalization of the General Theory of Quaternions

represented as $\left(\frac{a, b}{\mathbb{F}}\right)$. It is a vector space in \mathbb{F} , with the basis

$$\begin{aligned} 1 &= (\text{one}_{\mathbb{F}}, \text{zero}_{\mathbb{F}}, \text{zero}_{\mathbb{F}}, \text{zero}_{\mathbb{F}}) & i &= (\text{zero}_{\mathbb{F}}, \text{one}_{\mathbb{F}}, \text{zero}_{\mathbb{F}}, \text{zero}_{\mathbb{F}}) \\ j &= (\text{zero}_{\mathbb{F}}, \text{zero}_{\mathbb{F}}, \text{one}_{\mathbb{F}}, \text{zero}_{\mathbb{F}}) & k &= (\text{zero}_{\mathbb{F}}, \text{zero}_{\mathbb{F}}, \text{zero}_{\mathbb{F}}, \text{one}_{\mathbb{F}}) \end{aligned}$$

and a distributive product, such that : $i^2 = a, j^2 = b, ij = k$ (cf. axioms above), and $ij = -ji$, for $a = (a, \text{zero}_{\mathbb{F}}, \text{zero}_{\mathbb{F}}, \text{zero}_{\mathbb{F}})$, $b = (b, \text{zero}_{\mathbb{F}}, \text{zero}_{\mathbb{F}}, \text{zero}_{\mathbb{F}})$.


Hamilton's quaternions are the first introduced structure of quaternions [11]. After its discovery, the research for structures similar to the original quaternions started, leading to a more generic and algebraic definition than the classic approach of Hamilton. Our specification in PVS uses such a generic definition.

Using the notation above, Hamilton's quaternions is the algebra $\mathbb{H} = \left(\frac{-1, -1}{\mathbb{R}}\right)$.

The structure of Hamilton's quaternions is the most popular because of its well-known efficient applicability in manipulating three-dimensional (3D) objects. Despite this fact, the interest in quaternions is not limited to Hamilton's ones but also to other structures of quaternions that are of great interest (e.g., [22]).

1.1 Main results


This paper describes the formalization of the general theory of the structures of quaternions in the interactive proof assistant PVS. It provides a characterization of quaternions as division rings based on algebraic properties of fields. The characterization is crucial to building multiplicative inverses for non-zero quaternion elements, an essential element in structures such as Hamilton's quaternions. In addition, the formalization shows how to build the structure of Hamilton's quaternions with adequate theory parameters. Finally, we formalize a completeness theorem of Hamilton's quaternions to rotate any 3D vector.

The quaternions theory is developed over the PVS nasalib theory [algebra](#) . Recent developments on this theory are reported in [4]. The theory includes complete proofs of the three isomorphism theorems for rings, characterizations of principal, prime, and maximal ideals, and an abstract algebraic-theoretical version of the Chinese Remainder Theorem for arbitrary rings [7]. Also, it includes a division algorithm for Euclidean rings and Unique Factorization Domains [6].

As far as we know, there are three solid formalizations restricted to the structure of Hamilton's quaternions, one of them in HOL Light [9], another in Coq [2], and the third one in Isabelle/HOL [18]. The HOL Light formalization applies to verify basic parts of theories related to slicing regular functions and Pythagorean-Hodograph curves; the second one in Coq has been applied to formalize 3D robot manipulators; and the one in Isabelle/HOL inspired Koutsoukou-Argraki's formalization of octonions [13]. In contrast, some elements of the general theory of quaternions built over any abstract field, as in our case, were only developed as part of the Lean mathlib library [16].

1.2 Organization

Section 2 is divided into subsections discussing the basic elements used in the specification and axiomatization of the general theory of quaternions (2.1), discussing how the algebraic properties of such structures are inferred from the axiomatization (2.2), and how quaternions are characterized as division rings (2.3). Section 3 is divided into two subsections presenting the theory parameters used to obtain Hamilton's quaternions (3.1), and the formalization


■ **Specification 1** Quaternion addition and scalar multiplication [quaternion_def](#) 

```

+(u,v): quat = ( u'x + v'x, u'y + v'y, u'z + v'z, u't + v't ) ;
*(c,v): quat = ( c * v'x, c * v'y, c * v'z, c * v't ) ;
                                                    %scalar multiplication
* :[quat,quat -> quat] ;
                                                    %quaternion multiplication

```

of the completeness of this structure to deal with 3D vector rotations (3.2). Finally, before concluding and discussing future lines of research in Section 5, Section 4 briefly discusses how other structures of quaternions can be specified.


The paper includes links to the specific points of the specification. The formalization is part of the PVS nasalib theory [algebra](#) . Formalizations in PVS are given in two files with extensions *.pvs* and *.prf*. The former contains the specifications, whereas the latter contains the proofs. The system itself, as well as relevant documentation about it, can be found in [1]. Also, an extension for PVS is available for VSCode [15].

2 Mechanization of the theory of quaternions

This section presents the formalization of the theory of quaternions using as a parameter an algebraic field and two constants: $\langle \mathbb{F}, +_{\mathbb{F}}, *_{\mathbb{F}}, \text{zero}_{\mathbb{F}}, \text{one}_{\mathbb{F}}, a, b \rangle$.

2.1 Specification of Basic Notions

The general theory of quaternions is built from any abstract type T , with binary operators for addition and multiplication $+, * : [T, T] \rightarrow T$, with constants $\text{zero}, \text{one}, a, b : T$.

Initially, in the theory defining the structure and type `quat`, [quaternion_def](#) , it is only assumed that $[T, +, \text{zero}]$ is a group: `group?(fullset[T])`. An element q of type `quat` is a quadruple of elements of type T , represented as $q = (x, y, z, t)$, and through the use of a macro, components of q can be accessed, for instance $q.y = y$. Quadruples for the quaternion basis $1, i, j, k$, and for quaternions a and b are defined; distinguishing them with names `one_q, i, j, k, a_q, b_q`. The substring `_q` refers to quaternions. Thus, field elements with the suffix `_q` refer to the associated quaternions; for instance, `a_q` refers to the quaternion $(a, \text{zero}, \text{zero}, \text{zero})$, and `zero_q` specifies the zero quaternion. The conjugate and the additive inverse of a quaternion are specified in the usual manner: they are well-defined since $[T, +, \text{zero}]$ is a group, and each element of the quadruple has an additive inverse. Tuple addition and scalar multiplication are defined in Specification 1. Finally, note that quaternion multiplication is defined as a binary operator over quaternions.

The required axioms of the theory of quaternions are given in Specification 2, where variable types are $u, v : \text{quat}$, and $c, d : T$. Notice that the axioms include associativity and (right and left) distributivity of the quaternion multiplication over the addition (`q_assoc, q_distr` and `q_distr1`), and associativity and commutativity regarding scalar multiplication over quaternion multiplication (`sc_quat_assoc, sc_comm` and `sc_assoc`). Also, it is required that `one_q` be the identity for quaternion multiplication: the axioms `one_q_times` and `times_one_q` are essential to prove the characterization of the quaternion multiplication provided in the Subsection 2.2.

■ **Specification 2** [Axioms for the Theory of Quaternion](#) [↗](#)

```

sqr_i      :AXIOM i * i = a_q
sqr_j      :AXIOM j * j = b_q
ij_is_k    :AXIOM i * j = k
ji_prod    :AXIOM j * i = inv(k)
sc_quat_assoc :AXIOM c*(u*v) = (c*u)*v
sc_comm    :AXIOM (c*u)*v = u*(c*v)
sc_assoc   :AXIOM c*(d*u) = (c*d)*u
q_distr    :AXIOM distributive?[quat](*, +)
q_distr1   :AXIOM (u + v) * w = u * w + v * w
q_assoc    :AXIOM associative?[quat](*)
one_q_times :AXIOM one_q * u = u
times_one_q :AXIOM u * one_q = u

```

■ **Specification 3** [Quaternion Basis](#) [↗](#)

```

basis_quat: LEMMA
  FORALL (q: quat): q = q'x * one_q + q'y * i + q'z * j + q't * k

```

2.2 Inference of Algebraic Properties of Quaternions

The PVS theory [quaternions](#) [↗](#) completes the basic structure of quaternions, refining the parameters in such a manner that a and b are different from zero, and $[T, +, *, \text{zero}, \text{one}]$ is a field (specified in theory [field_def](#) [↗](#)). So, the type T with addition and zero , as well as $T - \{\text{zero}\}$ with multiplication and one are Abelian groups.

From this basis, it is now possible to infer a series of lemmas about quaternions, such as $j*i = -(i*j)$, $k*k = -a_q * b_q$, $k * i = -a_q * j$, $k * j = b_q * i$, $i * k = a_q * j$, and $j * k = -b_q * i$ (see [basic lemmas](#) [↗](#)).

Such lemmas allow us to infer that quaternions one_q , i , j , and k act as a basis as given in Specification 3, and the characterization of quaternion multiplication as given in Specification 4. The proof of this characterization uses the decomposition according to the lemma [basis_quaternion](#) and requires exhaustive algebraic manipulation using quaternion axioms, a series of auxiliary lemmas, including the previous ones mentioned, and others about the algebra of quaternions, such as lemmas for the scalar product. The advantage of such formulation is that the characterization of quaternion multiplication, usually presented as a definition, is obtained from a minimal axiomatization.

Further results include the formalization of the fact that any quaternion abstract structure, $\text{quat}[T, +, *, \text{zero}, \text{one}, a, b]$, is a ring with identity as given in the Specification 5. A ring is not necessarily commutative regarding multiplication. The proof requires expanding the field definition for $[T, +, *, \text{zero}, \text{one}]$, then using that it is a commutative division ring, a commutative group with identity. From this, and the

■ **Specification 4** [Quaternion Multiplication Characterization](#) [↗](#)

```

q_prod_charac: LEMMA FORALL (u,v:quat):
  u * v = (u'x * v'x + u'y * v'y * a + u'z * v'z * b + u't * v't * inv(a)*b,
          u'x * v'y + u'y * v'x + (inv(b)) * u'z * v't + b * u't * v'z,
          u'x * v'z + u'z * v'x + a * u'y * v't + inv(a) * u't * v'y,
          u'x * v't + u'y * v'z + inv(u'z * v'y) + u't * v'x );

```


■ **Specification 5** [Quaternions are Rings with identity](#) 


```
quat_is_ring_w_one: LEMMA
  ring_with_one?[quat,+,*,zero_q,one_q](fullset[quat])
```

■ **Specification 6** [Conjugate of Multiplication of Quaternions](#) 

```
conj_product_quat : LEMMA FORALL(q, u : quat) :
  conjugate(q * u) = conjugate(u) * conjugate(q)
```



algebraic properties inferred until this point, it is possible to prove that the structure of quaternions given as $[\text{quat}[T,+,*,\text{zero},\text{one},a,b], +, *, \text{zero}_q, \text{one}_q]$ is indeed a ring with identity. The last is done expanding the notion of ring-with-identity and proving first that $[\text{quat}[T,+,*,\text{zero},\text{one},a,b], +, *, \text{zero}_q]$ is a ring, and then that $[\text{quat}[T,+,*,\text{zero},\text{one},a,b], *, \text{one}_q]$ is a monoid.

Some of the formalizations benefit from PVS strategies to automatize manipulation of the algebra of quaternions. For instance, the lemma in Specification 6 states that for quaternions q, u , $\text{conjugate}(q * u) = \text{conjugate}(u) * \text{conjugate}(q)$, where [conjugate\(u\)](#)  is given by the quaternion $(u'x, -u'y, -u'z, -u't)$. The proof of this lemma is done by applying the theorem of characterization of quaternion multiplication [q_prod_charac](#), showing that each pair of corresponding components of the resulting quadruples are equal. Quaternions' operations are defined from addition and multiplication over arbitrary fields. PVS allows the manipulation of numerical algebraic structures, such as the field of reals. Indeed, Manip is a package of PVS tactics that simplify numerical manipulation [8]. However, it does not support algebraic manipulations over arbitrary fields.

Simple strategies were developed to handle quaternions' operations [PVS strategies](#) . Roughly, a strategy in PVS is a proof script that can be applied as a PVS proof command to improve automation. For instance, at some point in the proof, one must show that the quadruples' first components coincide with the corresponding equation presented below. However, proving this equality is not straightforward, requiring exhaustive applications of quaternions' addition and multiplication properties, which justified the development of such strategies.

$$\begin{aligned} &-(q'x * u't + q'y * u'z + -(q'z * u'y) + q't * u'x) = \\ &-(u'x * q't) + u'y * q'z + -(u'z * q'y) + -(u't * q'x) \end{aligned}$$

Some additional lemmas and definitions are formalized to characterize quaternions as division rings.

Two important predicates and subtypes of `quat` are defined, the type of pure quaternions, [pure_quat](#) , and the type of scalar quaternions, [scalar_F](#) , which consists of quaternions with null scalar component and with null components i, j, k , respectively. Also, we specify the *reduced norm* of a quaternion q as $\text{red_norm}(q) = q * \text{conjugate}(q)$. The lemmas obtained for such definitions cover the properties in the Specification 7, among others.

The lemma [center_quat_is_sc_F](#) expresses the fact that if the characteristic of the ring $[T, +, *, \text{zero}]$ is different from two, i.e., there exists an element $x \in T$ such that $x + x \neq \text{zero}$, the center of the structure built with the quaternions and its multiplication is exactly the subtype of all the scalar quaternions.

■ **Specification 7** Pure and Scalar Quaternions Conjugate and Norm Properties [↗](#)

```

red_norm_charac: LEMMA FORALL (q: quat):
  red_norm(q) = (q'x * q'x +
                inv(a) * (q'y * q'y) +
                inv(b) * (q'z * q'z) +
                (a * b) * (q't * q't),
                zero, zero, zero)

conj_product_quat_scalar : LEMMA FORALL(s : T, q : quat) :
  conjugate(s * q) = s * conjugate(q)

red_norm_conj: LEMMA FORALL(q:quat):
  red_norm(conjugate(q)) = red_norm(q)

center_quat_is_sc_F: LEMMA charac(fullset[T]) /= 2 IMPLIES
  center[(quat),*](fullset[quat]) = scalar_F

q_x_v_cq : LEMMA FORALL (q:quat, v:(pure_quat)) :
  pure_quat(q * v * conjugate(q))

```

■ **Specification 8** $T_q(q)(v)$ Operator [↗](#)

```

T_q(q: quat)(v:(pure_quat)): (pure_quat) = q * v * conjugate(q)

T_q_is_linear: LEMMA FORALL (c,d: T, q: quat, v,w: (pure_quat)):
  T_q(q)(c * v + d * w) = c * T_q(q)(v) + d * T_q(q)(w)

T_q_red_norm_invariant: LEMMA FORALL (q: quat, v:(pure_quat)):
  red_norm(q) = one_q IMPLIES red_norm(T_q(q)(v)) = red_norm(v)

T_q_invariant_red_norm: LEMMA FORALL (c: T, q: quat):
  red_norm(q) = one_q IMPLIES T_q(q)(c * pure_part(q)) = c * pure_part(q)

```

The center of such structure is given by the quaternions that multiplicatively commute with all other quaternions: $\{ q \mid \forall u : q * u = u * q \}$. This theorem is obtained, proving that for any quaternion q in the center, commutativity with the basis quaternions i , j , k implies the pure components of x should be zero.

Finally, from the last lemma in Specification 7, $q_x_v_cq$, the transformation given as the curried operator $T_q(q:quat)(v:(pure_quat))$ is specified, and crucial properties about it are proved, as presented in Specification 8. Such properties express the linearity of the operator, $T_q_is_linear$; the fact that if the red_norm of q is one, the resulting transformation of the pure quaternion v , $T_q(q)(v)$, has the same norm as v ; and, that the transformation over the pure quaternion $pure_part(q)$, obtained from q , does not affect any multiple of it. In fact, the last lemma could be obtained by proving that $T_q(q)(pure_part(q))=pure_part(q)$ and by the fact that $T_q(q)$ is linear.

Quaternions of characteristic two require specialized definitions but are not the subject of this paper (e.g., Chapter six of [22]).

2.3 Characterization of Quaternions as Division Rings

The characterization of quaternions as division rings is given by a series of six lemmas presented in Specification 9.

The first lemma, `nz_red_norm_if_inv_exist`, is proved constructively. Assuming $\text{red_norm}(q) \neq \text{zero_q}$, using the characterization of `red_norm` in Specification 7, one has that the scalar component of $\text{red_norm}(q) = q'x * q'x + -(a) * (q'y * q'y) + -(b) * (q'z * q'z) + (a * b) * (q't * q't)$ is not null and consequently has a multiplicative inverse in the field, say y . From this, one builds the desired quaternion multiplicative inverse of q as the quaternion $\text{conjugate}(q) * (y * \text{one_q})$. We have to consider the quaternion $y * \text{one_q}$ in the previous multiplication since y is a scalar. The exhaustive job is once again related to the algebraic manipulation to prove that $q * (\text{conjugate}(q) * (y * \text{one_q})) = \text{one_q}$ and vice-versa. This involves repeated applications of the characterization of quaternion multiplication, the definition and characterization of `red_norm`, and several algebraic properties of quaternions.

The second lemma in Specification 9, `div_ring_iff_nz_rednorm`, established that a quaternion is a division ring exactly when all non-zero quaternions have a reduced norm different from `zero_q`. Necessity is proved by contradiction, assuming the existence of an inverse for q , say $y * q = \text{one_q}$. Then, by expanding the definition of reduced norm, one obtains $q * \text{conjugate}(q) = \text{zero_q}$. From these equations, by simple algebraic manipulations, one obtains $y * (q * \text{conjugate}(q)) = \text{one_q} * \text{conjugate}(q)$, and finally one obtains $\text{zero_q} = \text{conjugate}(q)$, which contradicts the assumption that $q \neq \text{zero_q}$. The proof of sufficiency is obtained by applying the first lemma.

The third lemma in Specification 9, `inv_q_prod_charac`, characterizes the inverse of a non `zero_q` quaternion q through the equation $\text{inv}(q) = \text{conjugate}(q) * \text{inv}(\text{red_norm}(q))$ whenever the quaternion structure is a division ring. This lemma uses the previous one and exhaustive algebraic manipulation. The key of the proof is to show that $\text{conjugate}(q) * (\text{red_norm}(q))^{-1}$ is the inverse of q . This is proved showing that $q * (\text{conjugate}(q) * (\text{red_norm}(q))^{-1}) = \text{one_q}$ and $(\text{conjugate}(q) * (\text{red_norm}(q))^{-1}) * q = \text{one_q}$. The former equation requires only associativity and expansion of the definition of `red_norm` to obtain the equation $(q * \text{conjugate}(q)) * (q * \text{conjugate}(q))^{-1} = \text{one_q}$, from which one concludes. The latter equation requires the application of the previous lemma to obtain the multiplicative inverse of `red_norm`, say y , such that $\text{red_norm}(q) * y = \text{one_q}$. Expanding the definition of `red_norm`, one obtains the equation $(q * \text{conjugate}(q)) * y = \text{one_q}$. In this manner, one obtains the equation $q * ((\text{conjugate}(q) * y) * q) = q * \text{one_q}$, from which one concludes.

The fourth lemma in Specification 9, `quat_div_ring_aux1`, is a simple auxiliary result from the theory of fields. If $t = \text{zero}$, the type of a implies $-a \neq \text{zero}$. For the case in which $t \neq \text{zero}$, after Skolemization, one obtains the premise $t*t = a$; also, t has a multiplicative inverse, say y . Then, by instantiating the premise with y and zero , one obtains objective equality $a*(y*y) + b * \text{zero} = \text{one}$. By replacing a with $t*t$, one obtains $(t*t)*(y*y) = \text{one}$. The formalization, as expected, requires simple field algebraic manipulations.

The fifth lemma, `quat_div_ring_aux2`, is another auxiliary result on fields. When $t = \text{zero}$, one concludes by the inequation results from the type of b . Otherwise, let y and $y1$ be the multiplicative inverses of t and $a + a$, respectively. Notice that since the characteristic of the field is different from two, $a + a \neq \text{zero}$, allowing the use of the latter inverse. The second premise is then instantiated with $(\text{one} + a) * y1$ and $(\text{one} - a) * y1 * y$ giving the objective

$$a((\text{one} + a) * y1)^2 + b((\text{one} - a) * y1 * y)^2 = \text{one}$$

Algebraic manipulation transforms the left-hand side of this equation into the term below,

■ **Specification 9** Characterization of Quaternions as Division Rings [↗](#)

```

nz_red_norm_iff_inv_exist: LEMMA
  (FORALL (q:nz_quat):
    red_norm(q) /= zero_q) IFF
    inv_exists?[quat,*,one_q](remove(zero_q, fullset[quat]))

div_ring_iff_nz_rednorm: LEMMA
  division_ring?[quat,+,*,zero_q,one_q](fullset[quat]) IFF
  (FORALL (q: nz_quat): red_norm(q) /= zero_q)

inv_q_prod_charac: LEMMA
  division_ring?[quat,+,*,zero_q,one_q](fullset[quat]) IMPLIES
  (FORALL (q: nz_quat):
    inv[nz_quat,*,one_q](q) = conjugate(q)*inv[nz_quat,*,one_q](red_norm(q)))

quat_div_ring_aux1: LEMMA
  (FORALL (x,y:T): a * (x*x) + b * (y*y) /= one) IMPLIES
  (FORALL (t:T): t*t + inv[T,+,zero](a) /= zero)

quat_div_ring_aux2: LEMMA
  (charac(fullset[T]) /= 2 AND (FORALL (x,y:T): a * (x*x)+b * (y*y) /= one))
  IMPLIES
  (FORALL (t:T): a*(t*t) + b /= zero)

quat_div_ring_char: LEMMA
  charac(fullset[T]) /= 2 IMPLIES
  ((FORALL (x,y:T): a*(x*x) + b*(y*y) /= one) IFF
  division_ring?[quat,+,*,zero_q,one_q](fullset[quat]))

```

where for the integer k , k t abbreviates $t+t+\dots+t$ k times.

$$a * y1^2 + 2(a^2 * y1^2) + a^3 * y1^2 + b * y1^2 * y^2 + 2(b * (-a) * y1^2 * y^2) + b * (-a)^2 * y1^2 * y^2$$

By multiplying $a*(t*t) + b = \text{zero}$ by $y * y$, one obtains the equation $a + b (y * y) = \text{zero}$, which allows the elimination of the first and second component of the above term; indeed

$$a * y1^2 + b * y1^2 * y^2 = (a + b * y^2)y1^2 = \text{zero}$$

The third and last components are also eliminated:

$$a^3 * y1^2 + b * (-a)^2 * y1^2 * y^2 = (a + b * y^2) * a^2 * y1^2 = \text{zero}$$

Finally, the remaining four components are proved equal to **one** using the equation $-b * (y * y) = a$:

$$2(a^2 * y1^2) + 2(b * (-a) * y1^2 * y^2) = 4(a^2 * y1^2) = (a + a) * (a + a) * y1^2 = \text{one}$$

The final lemma, `quat_div_ring_char`, states that the structure of quaternions with multiplication is a division ring whenever the characteristic of the ring $[T, +, *, \text{zero}]$ with field multiplication is different from two and the condition $\forall x, y \in T : a * x^2 + b * y^2 \neq \text{one}$, used in previous two lemmas, holds. The proof applies the second lemma in the series of lemmas given in Specification 9, `div_ring_iff_nz_rednorm`, thus, changing the objective to proving that $\text{red_norm}(q) \neq \text{zero}_q$, for any $q \neq \text{zero}_q$ under these conditions.

On the one side, if there exists x, y in the field such that $a * x^2 + b * y^2 = \text{one}$, one can select the quaternion element $q = \text{one}_q + x * i + y * j$. So, $q \neq \text{zero}_q$, and its reduced norm, $1 - a * x^2 - b * y^2$ is different from **zero**. Therefore, the quaternion cannot be a

division ring. On the other side, suppose the quaternion is not a division ring, but the condition $\forall x, y \in \mathbb{T} : a * x^2 + b * y^2 \neq \text{one}$ holds. Then, there exists $q \neq \text{zero}_q$ such that $\text{red_norm}(q) = q'x^2 - a * q'y^2 - b * q'z^2 + a * b * q't^2 = \text{zero}_q$. For short, let this q be equal to (x, y, z, t) .

The first component of the reduced norm gives the field equation:

$$x^2 - a * y^2 - b * z^2 + a * b * t^2 = \text{zero} \quad (1)$$

From the last equation, one has that $x^2 - a * y^2 = b * (z^2 - a * t^2)$. From this equation, one obtains $(x^2 - a * y^2) * (z^2 - a * t^2) = b * (z^2 - a * t^2)^2$. This equation gives

$$(x^2 * z^2 + a^2 * y^2 * t^2 - a * x^2 * t^2 - a * y^2 * z^2) = b * (z^2 - a * t^2)^2$$

From the last equation, one obtains

$$a * (x * t + y * z)^2 + b * (z^2 - a * t^2)^2 = (x * z + a * y * t)^2 \quad (2)$$

Notice that $(x * z + a * y * t) = \text{zero}$; otherwise, multiplying the equation by the square of the inverse of this term, one contradicts the hypothesis $\forall x, y \in \mathbb{T} : a * x^2 + b * y^2 \neq \text{one}$. Therefore, equation (2) becomes:

$$a * (x * t + y * z)^2 + b * (z^2 - a * t^2)^2 = \text{zero} \quad (3)$$

Suppose now that $z^2 - a * t^2 \neq \text{zero}$. Thus, multiplying the equation by the square of the inverse of this term, one obtains an equation of the form $a * t'^2 + b = \text{zero}$, which gives a contradiction by lemma `quat_div_ring_aux2`. Thus, $z^2 - a * t^2 = \text{zero}$.

Assume now that $t \neq \text{zero}$. Multiplying by the square of the inverse of t , one obtains an equation of the form $t'^2 - a = \text{zero}$, which gives a contradiction by lemma `quat_div_ring_aux1`. Therefore, the fourth component of the quaternion element q is zero: $t = \text{zero}$, which also implies the third component $z = \text{zero}$.

Thus the reduced norm of q is equal to $x^2 - ay^2$, and by hypothesis, $x^2 - ay^2 = \text{zero}$. Once again, if $y \neq \text{zero}$, multiplying the equation by the square of the inverse of y , one obtains an equation of the form $t'^2 - a = \text{zero}$, which gives a contradiction by lemma `quat_div_ring_aux1`. So, $y = \text{zero}$, and also $x = \text{zero}$.

This completes the proof.

3 Parameterization of the Algebra of Hamilton's Quaternions

By providing parameters `quaternions[real,+,*,0,1,-1,-1]` to the theory `quaternions` [↗](#), one obtains Hamilton's quaternions, \mathbb{H} , mentioned in the introduction. This structure is usually characterized in textbooks by the identities $i^2 = j^2 = k^2 = ijk = -1$ (e.g., [22]). In this section, we will present the completeness of 3D rotation by using Hamilton's quaternion, as well as the main properties to achieve such results formalized in the PVS theory `quaternions_Hamilton` [↗](#). In this section, "quaternions" reference elements of the structure of Hamilton's quaternions.

3.1 Specification of Basic Properties

The structure given by $(\mathbb{H}, +_{\mathbb{H}}, \text{zero}_q, *_R)$, where $*_R$ indicates the scalar product induced by the multiplication over real numbers, is a vector space isomorphic to \mathbb{R}^4 equipped with their standard operations. A pure part of a quaternion can be mimicked by a vector from

11:10 A Formalization of the General Theory of Quaternions

■ Specification 10 [Connection between quaternions and vectors](#)

```
Real_part(q: quat): real = q.x

Vector_part(q: quat): Vect3 = (q.y, q.z, q.t)

conversion_quot: LEMMA
  FORALL(r: real, nz: nzreal): r/nz = number_fields./(r,nz)

quat_is_Real_p_Vector_part: LEMMA
  FORALL (q: quat):
    q = (Real_part(q), Vector_part(q).x, Vector_part(q).y, Vector_part(q).z)

decompose_eq_Real_Vector_part: LEMMA
  FORALL (q, p : quat):
    Real_part(q) = Real_part(p) AND Vector_part(q) = Vector_part(p) IFF
    q = p

Vector_part_scalar: LEMMA
  FORALL (k:real, q: quat): Vector_part(k*q) = k * Vector_part(q)
```

\mathbb{R}^3 and has a fundamental role in the theorems regarding the completeness of 3D rotations. To reuse results about real vectors, formalized in theory [vectors](#) in PVS nasalib, we specified operators that return the real and pure part of a quaternion as a real number and a three-dimensional vector, respectively, and formalized basic properties about them (see Specification 10).

3.2 Rotational completeness of Hamilton's Quaternions

Hamilton's quaternions is a suitable structure to perform rotations in \mathbb{R}^3 , and it has some advantages when compared with techniques based on rotating by Euler angles:

- The rotation using quaternions relies on the application of the linear transformation $T_q(q)(v)$, defined in Specification 8. This operator is based on the multiplication of three quaternions which, in the light of the lemma [q_prod_charac](#), is computed using multiplication and sum of real numbers in this context. On the other hand, rotating by Euler angles relies on the multiplication of three matrices of order 3, whose entries contain trigonometric functions, each one of these matrices represents a rotation around the axes x, y , and z of a 3D coordinate system (e.g., Chapter 4 in [3], and [19]). Thus, Hamilton's quaternions provide a computational, more efficient manner to perform rotations.
- Rotating by Euler angles can lead to a *gimbal lock*. This well-known phenomenon occurs when two axes align, causing the loss of one degree of freedom and *locking* the system to rotate in a degenerated two-dimensional space [10]. Hamilton's quaternions avoid *gimbal lock*.
- A rotation by Euler angles is based on the composition of rotations around three axes, e.g., yaw, pitch, and roll. In contrast, only the pure part of a quaternion element q defines the axis of a rotation using Hamilton's quaternions [10]. Therefore, it is easier to visualize the transformation by quaternions.

The landmark results of this section, presented in the Specification 11, are the formalizations of theorems [Quaternions_Rotation](#) and [Quaternions_Rotation_Deform](#). The former states that given two pure quaternions a and b , which can be identified as vectors of \mathbb{R}^3 of the same norm, there is a quaternion $q = \text{rot_quat}(a, b)$ such that the operator

■ **Specification 11** Completion of rotation using Hamilton's quaternions [↗](#)

```

Quaternions_Rotation: THEOREM
FORALL (a:(pure_quat), b:(pure_quat) |
      norm(Vector_part(a)) = norm(Vector_part(b)) AND
      linearly_independent?(Vector_part(a), Vector_part(b))):
  LET q = rot_quat(a,b) IN b = T_q(q)(a)

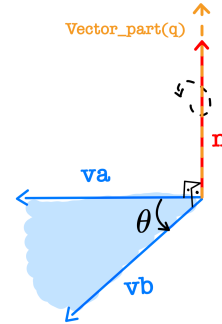
Quaternions_Rotation_Deform: THEOREM
FORALL (a:(pure_quat), b:(pure_quat) |
      linearly_independent?(Vector_part(a), Vector_part(b))):
  LET q =
    (sqrt(number_fields./(norm(Vector_part(b)),norm(Vector_part(a))))*
     rot_quat(a,
     number_fields./(norm(Vector_part(a)),norm(Vector_part(b))) * b)
  IN b = T_q(q)(a)

```

$T_q(q)$ rotates a into b . The latter theorem ensures the existence of a quaternion q such that the operator $T_q(q)$ transforms a into b , even when they are not, necessarily, of the same length. For the second transformation, it is only needed multiplying $\text{rot_quat}\left(a, \frac{|a|}{|b|}b\right)$ by

the scalar $\sqrt{\frac{|a|}{|b|}}$, where $|v|$ denotes the usual norm of v in \mathbb{R}^3 .

The following will highlight the main steps to formalize those theorems. Initially, consider two pure quaternions a and b such that $va = \text{Vector_part}(a)$ and $vb = \text{Vector_part}(b)$ are linearly independent; i.e., such vectors are nonparallel and non-null. Let θ be the smallest angle between va and vb and consider $n = \frac{va \times vb}{|va||vb|}$, where $va \times vb$ denotes the usual cross product of vectors in \mathbb{R}^3 . The idea is to consider n as the rotation axis and built the quaternion q that leads a into b from θ and n , as follows:



$$q = \left(\cos\left(\frac{\theta}{2}\right), n'_x * \sin\left(\frac{\theta}{2}\right), n'_y * \sin\left(\frac{\theta}{2}\right), n'_z * \sin\left(\frac{\theta}{2}\right) \right)$$

The elements θ , n and q were specified as [r_angle\(a,b\)](#) [↗](#), [n_rot_axis\(a,b\)](#) [↗](#), and [rot_quat\(a,b\)](#) [↗](#), respectively (See Specification 12). They use some structures formalized in the theories [vectors](#) [↗](#) and [trig](#) [↗](#) in the PVS nasalib. For example, [r_angle\(a,b\)](#) is formalized from the operator [angle_between\(Vector_part\(a\),Vector_part\(b\)\)](#) [↗](#), which, in turn, is specified by using the [arccosine](#) function and the usual *inner product* of \mathbb{R}^3 ; whereas, [n_rot_axis\(a,b\)](#) uses the specification of *cross product* defined as the vector [cross\(a,b\)](#) [↗](#). Notice that [r_angle\(a,b\)](#) [↗](#) has type [nnreal_le_pi](#), inheriting the adequate type (reals in the interval $(0, \pi]$) in which the function [acos](#) is specified in the PVS trigonometry library.

Four main lemmas are needed to formalize the Theorem [Quaternions_Rotation](#) [↗](#).

The first one consists of a characterization of the operator $T_q(q)(a)$ specified as the lemma [T_q_Real_charac](#) [↗](#). According to this result, for any quaternion q and any pure quaternion a , the following equality holds:

11:12 A Formalization of the General Theory of Quaternions

■ Specification 12 Basic elements to built a rotation by quaternions [↗](#)

```

r_angle(a,b:(nzpure_quat)): nreal_le_pi =
    angle_between(Vector_part(a),Vector_part(b))

n_rot_axis(a:(pure_quat),b:(pure_quat) |
    linearly_independent?(Vector_part(a), Vector_part(b))): Vect3 =
    normalize(cross(Vector_part(a), Vector_part(b)))

rot_quat(a:(pure_quat),b:(pure_quat) |
    linearly_independent?(Vector_part(a), Vector_part(b))): quat =
    LET rot_angl_halve : nreal_le_pi = number_fields./(r_angle(a,b), 2),
        sin_ha = sin(rot_angl_halve),
        cos_ha = cos(rot_angl_halve),
        n = n_rot_axis(a,b)
    IN (cos_ha, sin_ha * n'x, sin_ha * n'y, sin_ha * n'z)

```

$$\begin{aligned} \text{Vector_part}(T_q(q)(a)) &= ((q'x)^2 - |\text{Vector_part}(q)|^2) * va && + \\ & (2 * (\text{Vector_part}(q) * va)) * \text{Vector_part}(q) && + \quad (4) \\ & (2 * q'x) * (\text{Vector_part}(q) \times va) \end{aligned}$$

In Equation 4, the multiplication $v * w$ between vectors is interpreted as the dot product.

The vector part of $T_q(q)(a)$ expresses all the relevant information of the resulting quaternion: since the type established for $T_q(q)(a)$ is `pure_quat`, see Specification 8, the prover automatically generates a *proof obligation*, called in PVS *Type Correctness Condition (TCC)*, to verify that the first component of this quaternion is zero. Also, according to the lemma `T_q_is_linear`, showed in Specification 8, $T_q(q)(a)$ is a linear transformation. And since $|q| = 1$, it preserves the norm of $|a|$, acting as a rotation.

The other three key lemmas consist of established equivalent expressions for each term in the addition appearing in `T_q_Real_charac`, see Equation 4.

The lemma `Quat_Rot_Aux1` [↗](#) ensures that $\text{Vector_part}(q) * va = 0$. Consequently, the equation $(2 * (\text{Vector_part}(q) * va)) * \text{Vector_part}(q) = 0$ also holds.

The formalization of this lemma applies the lemma `orth_cross` [↗](#), of the PVS theory `vectors`, that guarantees that the vectors $(va \times vb)$ and va are orthogonal. This is a consequence of the equalities $\text{Vector_part}(q) = \sin\left(\frac{\theta}{2}\right) * n = \frac{\sin\left(\frac{\theta}{2}\right)}{|va||vb|} * (va \times vb)$.

The lemma `Quat_Rot_Aux2` [↗](#) establishes the equality

$$((q'x)^2 - |\text{Vector_part}(q)|^2) * va = \cos(\theta) * va$$

By definition of q and since $|n| = 1$,

$$(q'x)^2 - |\text{Vector_part}(q)|^2 = \cos^2\left(\frac{\theta}{2}\right) - \sin^2\left(\frac{\theta}{2}\right) * |n|^2 = \cos^2\left(\frac{\theta}{2}\right) - \sin^2\left(\frac{\theta}{2}\right)$$

Thus, `Quat_Rot_Aux2` follows as a consequence of the lemma `cos_2a` [↗](#), formalized in the theory `trig@trig_basic`, from which one can infer that $\cos^2\left(\frac{\theta}{2}\right) - \sin^2\left(\frac{\theta}{2}\right) = \cos(\theta)$.

Finally, in the lemma `Quat_Rot_Aux3` [↗](#), it is formalized that

$$(2 * q'x) * (\text{Vector_part}(q) \times va) = vb - \cos(\theta) * va$$

In fact, by definition of q and n , and the associative property for scalar elements, one can infer that:

$$(2 * q'x) * (\text{Vector_part}(q) \times va) = \left(2 \cos\left(\frac{\theta}{2}\right) \sin\left(\frac{\theta}{2}\right) \frac{1}{|va \times vb|} \right) ((va \times vb) \times va)$$

Applying the lemmas [cross_cross](#) and [sin_2a](#), specified in theories `vectors@cross_3D` and `trig@trig_basic`, respectively, one obtains the equality

$$\left(2 \cos\left(\frac{\theta}{2}\right) \sin\left(\frac{\theta}{2}\right) \frac{1}{|va \times vb|} \right) ((va \times vb) \times va) = \frac{\sin(\theta)}{|va \times vb|} ((va * va) * vb - (vb * va) * va)$$

Since, $(va * va) = |va|^2$ and $(vb * va) = \cos(\theta)|va||vb|$, it holds that

$$\frac{\sin(\theta)}{|va \times vb|} ((va * va) * vb - (vb * va) * va) = \frac{\sin(\theta)}{|va \times vb|} (|va|^2 * vb - (\cos(\theta) * |va||vb|) * a)$$

Thus, by using the fact the $|va| = |vb|$ and applying the identity $|va \times vb| = |va||vb|\sin(\theta)$, formalized in the lemma [norm_cross_charac](#) of the theory `vectors`, one obtains the equality

$$\frac{\sin(\theta)}{|va \times vb|} (|va|^2 * vb - (\cos(\theta) * |va||vb|) * va) = vb - \cos(\theta)va$$

The Theorem [Quaternions_Rotation](#) is then obtained as a direct consequence of the lemmas `T_q_Real_charac`, `Quat_Rot_Aux1`, `Quat_Rot_Aux2` and `Quat_Rot_Aux3`.

The formalization of the Theorem [Quaternions_Rotation_Deform](#) ensures that Hamilton's quaternions are useful to promote not only rotations in \mathbb{R}^3 but also linear scaling since the transformation $T_q(q)(a)$ maps a into b even when they are not of the same length.

For this, we have only to consider $q = \sqrt{\frac{|b|}{|a|}} * \text{rot_quat}\left(a, \frac{|a|}{|b|}b\right)$. In fact, using this q as argument of the transformation,

$$T_q(q)(a) = \sqrt{\frac{|b|}{|a|}} * \text{rot_quat}\left(a, \frac{|a|}{|b|}b\right) * a * \text{conjugate}\left(\sqrt{\frac{|b|}{|a|}} * \text{rot_quat}\left(a, \frac{|a|}{|b|}b\right)\right)$$

Then, applying the lemma [conj_product_quat_scalar](#), behind some algebraic manipulations, it holds that

$$\begin{aligned} T_q(q)(a) &= \sqrt{\frac{|b|}{|a|}} * \sqrt{\frac{|b|}{|a|}} * \text{rot_quat}\left(a, \frac{|a|}{|b|}b\right) * a * \text{conjugate}\left(\text{rot_quat}\left(a, \frac{|a|}{|b|}b\right)\right) \\ &= \frac{|b|}{|a|} * T_q\left(\text{rot_quat}\left(a, \frac{|a|}{|b|}b\right)\right)(a) \end{aligned}$$

Finally, since $|\text{Vector_part}(a)| = \left|\text{Vector_part}\left(\frac{|a|}{|b|}b\right)\right|$, the proof of the Theorem [Quaternions_Rotation_Deform](#) is completed instantiating [Quaternions_Rotation](#) with the pure quaternions a and $\frac{|a|}{|b|}b$, which guarantees that

$$T_q\left(\text{rot_quat}\left(a, \frac{|a|}{|b|}b\right)\right)(a) = \frac{|a|}{|b|}b,$$

and, consequently, that $T_q(q)(a) = b$.

It is important to note that only the crucial lemmas for formalizing the previous results were highlighted. Although the automation for the simplification of equations over reals is in an advanced stage in PVS, several algebraic manipulations involving associative property for scalars, characterization of the norm of a vector, and properties derived from linear independence, among others, were necessary to conclude the formal proofs.

4 Theory Parameters to Specify other Quaternions

Quaternion theory, as defined in Section 1, can describe many algebraic structures. Depending on the field \mathbb{F} and $a, b \in \mathbb{F}^\times$, the subset of invertible elements of the field, some quaternion algebra can be isomorphic to the matrix ring $M_2(\mathbb{F})$. In these cases, we say that the quaternion algebra *splits over* \mathbb{F} . In fact, it has been proved that a quaternion algebra $\left(\frac{a, b}{\mathbb{F}}\right)$, which is not a division ring, is indeed isomorphic to $M_2(\mathbb{F})$ [5].

An example is given by the quaternion built over the complex field: $\left(\frac{a, b}{\mathbb{C}}\right) \xrightarrow{\sim} M_2(\mathbb{C})$, in which not only, it splits for some values $a, b \in \mathbb{C} \setminus \{0\} = \mathbb{C}^\times$.

On the other hand, all $\left(\frac{a, b}{\mathbb{F}}\right)$ that are not isomorphic to $M_2(\mathbb{F})$ are division rings; an example are Hamilton's quaternions.

Another case of a quaternion that is a division ring is $\left(\frac{a, p}{\mathbb{Q}}\right)$, where p is an odd prime and a is a quadratic non-residue, or $\left(\frac{a, p}{\mathbb{Q}_p}\right)$, where \mathbb{Q}_p are the p -adic numbers and a, p having the same restrictions [22].

The formalization of the general theory of quaternions constitutes a starting point for dealing with other interesting applications of the theory of quaternions. Surveying only a few of the applications covered in Voight's book [22], we can mention the following: applications of quaternion algebras in analytic number theory, geometry (hyperbolic geometry and low-dimensional topology), arithmetic geometry, and supersingular elliptic curves. Also, Lewis surveys relevant applications of quaternion theory in several areas [14].

Many of these application topics use these different types of quaternions or their order. In this case, an order is understood as a subring of the quaternion algebra, which is also a lattice. In Voight's book [22], a more detailed description of interesting orders such as maximal order, Eichler order, and more general orders is given.

The Hurwitz quaternion order is one such maximal order used for proving theorems. This quaternion order is a subring of the quaternions \mathbb{H} and $\left(\frac{-1, -1}{\mathbb{Q}}\right)$, and is given by

$$H = \{\alpha\zeta + \beta i + \gamma j + \delta k \mid \alpha, \beta, \gamma, \delta \in \mathbb{Z}\}, \text{ where } \zeta = \frac{1}{2}(1 + i + j + k).$$

It is used to prove Lagrange's theorem that every positive integer is a sum of four squares. Furthermore, it is possible to prove that, short of commutativity, H has all the properties of Euclidean rings.

In the aforementioned proof of Lagrange's four-square theorem. Considering $u, v \in \mathbb{H}$:

$$u = a_0 + a_1 i + a_2 j + a_3 k, \text{ and } v = b_0 + b_1 i + b_2 j + b_3 k$$

Since $\text{Red_norm}(uv) = \text{Red_norm}(u) * \text{Red_norm}(v)$ [↗](#), the reduced norm in \mathbb{H} can be used to prove the Lagrange Identity in \mathbb{Z} :

$$(a_0^2 + a_1^2 + a_2^2 + a_3^2)(b_0^2 + b_1^2 + b_2^2 + b_3^2) = c_0^2 + c_1^2 + c_2^2 + c_3^2$$

where, by the characterization of quaternion multiplication:

$$\begin{aligned} c_0 &= a_0b_0 - a_1b_1 - a_2b_2 - a_3b_3 & c_1 &= a_0b_1 + a_1b_0 + a_2b_3 - a_3b_2 \\ c_2 &= a_0b_2 - a_1b_3 + a_2b_0 + a_3b_1 & c_3 &= a_0b_3 + a_1b_2 - a_2b_1 + a_3b_0 \end{aligned}$$

With this identity and by restricting the domain from \mathbb{H} to H , we can change the original problem from finding a solution for all positive integers into finding it for all primes. In this manner, the four integer square problem is expressed using only quaternions, which turns the Number Theory problem into an easier algebraic one. A didactic proof approach appears in Chapter 7 of Herstein's textbook [12]. Among other formalized properties available in the PVS nasalib theory algebra [\[4\]](#), the mechanization of this theorem uses the first isomorphism theorem for rings and results about maximal ideals [7].

Among the interesting applications in physics, it is possible to express gravity as part of a simple quaternion wave equation [21], the four Maxwell equations as a nonhomogeneous quaternion wave equation, as well as the Klein-Gordon equation as a quaternion simple harmonic oscillator [20]. Furthermore, under some restrictions, it is possible to express a quaternion analog to the Schrödinger equation, a well-known differential equation that governs the behavior of wave functions in quantum mechanics. The Schrödinger equation gives the kinetic energy plus the potential. To do this, we first look at the quaternions as the external tensor product of a scalar and an \mathbb{R}^3 -vector, denoted by $(\mathbf{s}, \tilde{\mathbf{v}})$, and write the quaternion in its polar form, namely:

$$\mathbf{q} = (\mathbf{s}, \tilde{\mathbf{v}}) = \|\mathbf{q}\| e^{\theta * \mathbf{I}} = \|\mathbf{q}\| (\cos(\theta) + \mathbf{I} * \sin(\theta)),$$

where $\|\mathbf{q}\| = \sqrt{\mathbf{q} * \text{conjugate}(\mathbf{q})}$, $\theta = \arccos\left(\frac{\mathbf{s}}{\|\mathbf{q}\|}\right)$, and $\mathbf{I} = \frac{\tilde{\mathbf{v}}}{\|\tilde{\mathbf{v}}\|}$. Note that $\mathbf{I}^2 = -1$.

Next, it is necessary to determine the quaternion wave function, ψ . Therefore, consider the quaternion $(\mathbf{t}, \tilde{\mathbf{R}})$ representing time and space, the quaternion $(\mathbf{E}, \tilde{\mathbf{P}})$ representing the electric field and momentum, and the quaternion $\mathbf{V}(0, \mathbf{X})$ representing the potential. Thus, with \hbar being the reduced Planck constant, we have:

$$\psi \equiv \frac{(\mathbf{t}, \tilde{\mathbf{R}}) * (\mathbf{E}, \tilde{\mathbf{P}})}{\hbar} = \frac{(\mathbf{E}\mathbf{t} - \tilde{\mathbf{R}} * \tilde{\mathbf{P}}, \mathbf{E} * \tilde{\mathbf{R}} + \tilde{\mathbf{P}} * \mathbf{t} + \tilde{\mathbf{R}} \times \tilde{\mathbf{P}})}{\hbar}$$

Passing ψ to its polar form, and assuming that ψ is normalized, we have the quaternion wave function:

$$\psi = e^{(\mathbf{E} * \mathbf{t} - \tilde{\mathbf{R}} * \tilde{\mathbf{P}}) * \mathbf{I} / \hbar}, \text{ where } \mathbf{I} = \frac{\mathbf{E} * \tilde{\mathbf{R}} + \tilde{\mathbf{P}} * \mathbf{t} + \tilde{\mathbf{R}} \times \tilde{\mathbf{P}}}{\|\mathbf{E} * \tilde{\mathbf{R}} + \tilde{\mathbf{P}} * \mathbf{t} + \tilde{\mathbf{R}} \times \tilde{\mathbf{P}}\|}$$

Now, the derivatives of ψ with respect to time and space give, respectively:

$$\frac{\partial \psi}{\partial \mathbf{t}} = \frac{\mathbf{E} * \mathbf{I}}{\hbar} \frac{\psi}{\sqrt{1 + \left(\frac{\mathbf{E} * \mathbf{t} - \tilde{\mathbf{R}} * \tilde{\mathbf{P}}}{\hbar}\right)^2}} \quad \text{and} \quad \nabla \psi = -\frac{\tilde{\mathbf{P}} * \mathbf{I}}{\hbar} \frac{\psi}{\sqrt{1 + \left(\frac{\mathbf{E} * \mathbf{t} - \tilde{\mathbf{R}} * \tilde{\mathbf{P}}}{\hbar}\right)^2}}$$

To achieve the objective, which is to establish an analog to the Schrödinger equation in terms of quaternions, it is necessary to consider some assumptions and verify the behavior of the quaternion wave function ψ . Among these assumptions are, for example, the conservation of energy and momentum and the assumption that $\mathbf{E} * \mathbf{t} - \tilde{\mathbf{R}} * \tilde{\mathbf{P}} = 0$. Therefore,

$$\frac{\partial \psi}{\partial \mathbf{t}} = \frac{\mathbf{E} * \mathbf{I}}{\hbar} \psi \Rightarrow -\mathbf{I} * \hbar \frac{\partial \psi}{\partial \mathbf{t}} = \mathbf{E} \psi \Rightarrow \mathbf{E} = -\mathbf{I} * \hbar \frac{\partial}{\partial \mathbf{t}}$$

11:16 A Formalization of the General Theory of Quaternions

$$\nabla\psi = -\frac{\tilde{\mathbf{P}} * \mathbf{I}}{\hbar}\psi \Rightarrow \mathbf{I} * \hbar\nabla\psi = \tilde{\mathbf{P}}\psi \Rightarrow \tilde{\mathbf{P}} = \mathbf{I} * \hbar\nabla$$

It is known that the momentum $\tilde{\mathbf{P}}$ is the product of the mass, m , and velocity, v . Consequently,

$$\tilde{\mathbf{P}}^2 = (mv)^2 = 2m\frac{mv^2}{2} = 2m \text{ KE} = -\hbar^2\nabla^2 \Rightarrow \text{KE} = -\frac{\hbar^2}{2m}\nabla^2$$

Since the Hamiltonian \mathbf{H} corresponds to the total energy (\mathbf{E}), that is, it is equal to the sum of the kinetic energy KE and the potential energy \mathbf{V} , we obtain the following equation, which is similar to the Schrödinger equation:

$$\mathbf{H}\psi = -\frac{\hbar^2}{2m}\nabla^2\psi + \mathbf{V} * \psi.$$

5 Conclusions and Future Work

Table 1 presents the number of lines in the proofs of the crucial lemmas and theorems on the characterization of quaternions as division rings and rotational completeness of Hamilton's quaternions formalized in the theories [quaternions](#) and [quaternions_Hamilton](#), respectively.

Although the complexity of proving rotational completeness is high, PVS supplies satisfactory algebraic automation of the field of reals \mathbb{R} , which makes the formalization of rotational completeness much simpler than the formalization of characterization of an arbitrary structure of quaternion as a division ring (observe the number of proof lines). Indeed, algebraic manipulation on standard number types, such as the type `real`, has been studied and implemented during the evolution of PVS, as reported by Muñoz and Mayero in [17] and di Vito in [8], among others. Although some simple strategies were developed in this work to apply automatically commutative and associative properties of the (general) field parameter over which quaternions were defined, the improvement of tactics and the availability of techniques to detect and cancel equal terms over algebraic theories as `field` and `quat` is indispensable. This will surely make it possible to simplify substantially the length of the proofs presented in Table 1 for the case of the theory of quaternions.

Possible future work includes formalizations of applications of quaternions theory in other areas as discussed in Section 4. For instance, a formalization of Lagrange's four-square theorem (in progress) required adequate parameters to the quaternion theory, proving that Hurwitz's substructure is indeed a ring and almost a Euclidean ring, except for commutativity. After such proof, a few more auxiliary arithmetic lemmas, such as Lagrange's Identity, which can turn the problem from finding solutions to all integers into finding for all primes, can be used for proving Lagrange's Theorem using quaternions.

In addition to the availability of the abstract theory of quaternions, other available PVS theories may be useful to formalize the application of quaternions in quantum mechanics discussed in Section 4. For instance, to specify quaternions in their polar form and the quaternion wave function, the core of theorems related to quaternion arithmetic and trigonometric theory should be useful; also, to formalize the Schrödinger equation, it will be extremely relevant to develop theorems or axioms on the differentiation of quaternions, and physics concepts, for example, momentum.

Of course, another urgent line of research is extending PVS tactics, strategies, and, in general, mechanisms of arithmetic manipulation for standard types as `int`, `nat`, and `reals` to abstract algebraic structures as `ring`, `field`, and `quat`.

■ **Table 1** Quantitative information.

Theory/Formula Name	Proof Line Numbers	Number of Proved Formulas
		Lemmas/Theorems
nz_red_norm_iff_inv_exist	125	1
div_ring_iff_nz_rednorm	95	1
inv_q_prod_charac	259	1
quat_div_ring_aux1	40	1
quat_div_ring_aux2	388	1
quat_div_ring_char	487	1
quaternions.pvs	10981	63
T_q_Real_charac	190	1
Quat_Rot_Aux1	10	1
Quat_Rot_Aux2	116	1
Quat_Rot_Aux3	106	1
Quaternions_Rotation	38	1
Quaternions_Rotation_Deform	94	1
quaternions_Hamilton.pvs	3662	30




References

- 1 PVS system. <https://pvs.csl.sri.com/index.html>. Accessed: 2024-05-27.
- 2 Reynald Affeldt and Cyril Cohen. Formal Foundations of 3D Geometry to Model Robot Manipulators. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs, CPP 2017, Paris, France, January 16-17, 2017*, pages 30–42. ACM, 2017. doi: 10.1145/3018610.3018629.
- 3 Howard Anton and Chris Rorres. *Elementary Linear Algebra: Applications Version*. John Wiley & Sons, Inc., 10th edition, 2010.
- 4 Mauricio Ayala-Rincón, Thaynara Arielly de Lima, André Luiz Galdino, and Andréia Borges Avelar. Formalization of Algebraic Theorems in PVS (Invited Talk). In *Proc. 24th International Conference on Logic for Programming, Artificial Intelligence and Reasoning LPAR*, volume 94 of *EPiC Series in Computing*, pages 1–10, 2023. doi:10.29007/7jbv.
- 5 Keith Conrad. Quaternion algebras. Accessed in March 13, 2024. URL: <https://kconrad.math.uconn.edu/blurbs/ringtheory/quaternionalg.pdf>.
- 6 Thaynara Arielly de Lima, Andréia Borges Avelar, André Luiz Galdino, and Mauricio Ayala-Rincón. Formalizing factorization on euclidean domains and abstract euclidean algorithms. *CoRR*, abs/2404.14920, 2024. In *Proceedings 18th Logical and Semantic Frameworks with Applications LSFA 2023*. doi:10.48550/arXiv.2404.14920.
- 7 Thaynara Arielly de Lima, André Luiz Galdino, Andréia Borges Avelar, and Mauricio Ayala-Rincón. Formalization of Ring Theory in PVS - Isomorphism Theorems, Principal, Prime and Maximal Ideals, Chinese Remainder Theorem. *J. Autom. Reason.*, 65(8):1231–1263, 2021. doi:10.1007/s10817-021-09593-0.
- 8 Ben L. Di Vito. *Manip User's Guide, Version 1.3*, 2012. URL: <https://pvs.csl.sri.com/doc/manip-guide.pdf>.
- 9 Andrea Gabrielli and Marco Maggesi. Formalizing Basic Quaternionic Analysis. In *Proceedings of the 8th International Conference on Interactive Theorem Proving, ITP*, volume 10499 of *Lecture Notes in Computer Science*, pages 225–240. Springer, 2017. doi: 10.1007/978-3-319-66107-0_15.
- 10 Gökmen Günaşti. Quaternion algebra, their applications in rotations and beyond quaternions. Technical report, Linnaeus University, Digitala Vetenskapliga Arkivet, 2012.

11:18 A Formalization of the General Theory of Quaternions

- 11 William Rowan Hamilton. On quaternions, or on a new system of imaginaries in algebra. *Philosophical Magazine*, 25(3):489–495, 1844. doi:10.1080/14786444408645047.
- 12 Israel (Yitzchak) Nathan Herstein. *Topics in Algebra*. John Wiley and Sons, New York, Chichester, Brisbane, Toronto, Singapore, second edition, 1975.
- 13 Angeliki Koutsoukou-Argyaki. Octonions. *Arch. Formal Proofs*, 2018. URL: <https://www.isa-afp.org/entries/Octonions.html>.
- 14 David W. Lewis. Quaternion Algebras and the Algebraic Legacy of Hamilton’s Quaternions. *Irish Math. Soc. Bulletin*, 57:41–64, 2006. doi:10.33232/bims.0057.41.64.
- 15 Paolo Masci and Aaron Dutle. Proof Mate: An interactive proof helper for PVS (tool paper). In *Proceedings of the 14th International Symposium NASA Formal Methods, NFM 2022*, volume 13260 of *Lecture Notes in Computer Science*, pages 809–815. Springer International Publishing, 2022. doi:10.1007/978-3-031-06773-0_44.
- 16 The mathlib Community. The Lean Mathematical Library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020*, pages 367–381. ACM, 2020. doi:10.1145/3372885.3373824.
- 17 César Muñoz and Micaela Mayero. Real Automation in the Field. Technical Report Interim report, No 39, NASA/ICASE, 2001.
- 18 Lawrence C. Paulson. Quaternions. *Arch. Formal Proofs*, 2018. URL: <https://www.isa-afp.org/entries/Quaternions.html>.
- 19 Logah Perumal. Euler angles: conversion of arbitrary rotation sequences to specific rotation sequence. *Comput. Animat. Virtual Worlds*, 25(5-6):521–529, 2014. doi:10.1002/CAV.1529.
- 20 Douglas Sweetser. Doing Physics with Quaternions, 2005. Accessed in March 13, 2024. URL: <https://theworld.com/~sweetser/quaternions/ps/book.pdf>.
- 21 Douglas Sweetser. Three Roads to Quaternion Gravity. In *APS March Meeting Abstracts*, volume 2019 of *APS Meeting Abstracts*, page T70.008, January 2019.
- 22 John Voight. *Quaternion Algebras*, volume GTM 288 of *Graduate Texts in Mathematics*. Springer Cham, 2021. doi:10.1007/978-3-030-56694-4.




A Modular Formalization of Superposition in Isabelle/HOL

Martin Desharnais   

Max-Planck-Institut für Informatik, Saarland Informatics Campus, Saarbrücken, Germany
Graduate School of Computer Science, Saarland Informatics Campus, Saarbrücken, Germany

Balazs Toth   

Ludwig-Maximilians-Universität München, Germany

Uwe Waldmann   

Max-Planck-Institut für Informatik, Saarland Informatics Campus, Saarbrücken, Germany

Jasmin Blanchette   

Ludwig-Maximilians-Universität München, Germany

Max-Planck-Institut für Informatik, Saarland Informatics Campus, Saarbrücken, Germany

Sophie Touret   

Université de Lorraine, CNRS, Inria, LORIA, Nancy, France

Max-Planck-Institut für Informatik, Saarland Informatics Campus, Saarbrücken, Germany

Abstract

Superposition is an efficient proof calculus for reasoning about first-order logic with equality that is implemented in many automatic theorem provers. It works by saturating the given set of clauses and is refutationally complete, meaning that if the set is inconsistent, the saturation will contain a contradiction. In this work, we restructured the completeness proof to cleanly separate the ground (i.e., variable-free) and nonground aspects, and we formalized the result in Isabelle/HOL. We relied on the *IsaFoR* library for first-order terms and on the Isabelle saturation framework.

2012 ACM Subject Classification Computing methodologies → Theorem proving algorithms

Keywords and phrases Superposition, verification, first-order logic, higher-order logic

Digital Object Identifier 10.4230/LIPIcs.ITP.2024.12

Supplementary Material *Other (Isabelle/HOL theory files)*: https://github.com/IsaFoL/IsaFoL/tree/ITP2024-IsaSuperposition/Superposition_Calculus [17]

archived at [swh:1:dir:9afd6985e81383a79df5325b0ea04e4fdc528228](https://swh.1:dir:9afd6985e81383a79df5325b0ea04e4fdc528228)

Funding *Jasmin Blanchette*: Co-funded by the European Union (ERC, Nekoka, 101083038). Views and opinions expressed are however those of the authors only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them.

Acknowledgements We thank Xavier Génereux, Mark Summerfield, and the anonymous reviewers for suggesting textual improvements.

1 Introduction

Superposition is a highly successful proof calculus for reasoning about first-order logic with equality designed by Bachmair and Ganzinger [2, 3]. It is implemented in many automatic theorem provers, including E [33], SPASS [45], Vampire [22], and Zipperposition [16].

Superposition provers work by refutation and saturation. They operate on a clause set, which initially consists of the clasified input problem in which the conjecture appears negated. Inferences are performed using clauses from this set as premises; the conclusions of inferences are added to the set. The prover stops when the empty clause \perp , denoting falsehood, is derived or when no more inferences are possible.



© Martin Desharnais, Balazs Toth, Uwe Waldmann, Jasmin Blanchette, and Sophie Touret; licensed under Creative Commons License CC-BY 4.0

15th International Conference on Interactive Theorem Proving (ITP 2024).

Editors: Yves Bertot, Temur Kutsia, and Michael Norrish; Article No. 12; pp. 12:1–12:20

Leibniz International Proceedings in Informatics



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

Consider the problem of proving $f(b) \approx f(a)$ from $b \approx a$, where \approx denotes equality. After negating the conjecture, we obtain the clause set $\{b \approx a, f(b) \not\approx f(a)\}$. The superposition calculus includes an inference rule called superposition that uses the first clause to rewrite the second clause to $f(a) \not\approx f(a)$. This new clause is added to the clause set. At this point, a unary inference rule called equality resolution uses $f(a) \not\approx f(a)$ to derive \perp .

During the saturation, the prover can delete clauses considered redundant, and it does not need to perform inferences considered redundant. For example, if the clause set contains $b \approx a$, then the clauses $f(b) \approx f(a)$ and $b \approx a \vee b \not\approx c$ are redundant. Deletion of redundant clauses helps reduce the clause explosion caused by saturation.

The inference rules of the superposition calculus are *sound*, meaning that the conclusion of each rule is entailed by the premises. This is easy to prove. What is much harder to show is that the calculus is *refutationally complete*: If a clause set is unsatisfiable and saturated (up to redundancy), then it contains \perp . We care about completeness because a complete calculus is likely to yield a higher success rate in practice than an incomplete one. Moreover, the completeness proof serves as a guide during the development of the calculus: Only inferences that are needed in the proof must be performed.

When developing proof calculi for first-order logic and beyond, it often helps to first develop a calculus that works on ground (i.e., variable-free) clauses. We can then lift it to the nonground level. This approach cleanly separates concerns. It is common in the literature [7–11, 29] and is supported by the *saturation framework* developed by Bachmair and Ganzinger [4, Section 4] and extended by Waldmann et al. [44], a collection of pen-and-paper results useful to establish the refutational completeness of saturation calculi and provers.

For superposition, Bachmair and Ganzinger’s completeness proof [3] does not separate the ground and nonground aspects. Waldmann et al. give some hints on how to instantiate the framework to obtain a modular proof that separates these aspects. Our main contributions are twofold. First, we elaborated these hints into a 15-page proof text [43] (summarized here in Section 3). Second, following this detailed *blueprint*, we formalized in Isabelle/HOL [24] the refutational completeness of ground superposition (Section 4) and lifted it to derive the refutational completeness of the nonground calculus (Section 5). We also proved soundness.

The separation of concerns, apart from allowing different people to work independently on different parts of the formalization, simplifies the completeness proof. On the ground level, there is no need to rename variables apart or to perform unification. On the nonground level, an inference overapproximates a set of ground inferences. Intuitively, this means that every inference on ground clauses can be simulated by inferences on corresponding nonground clauses. For superposition inferences, this roughly means that if $D\gamma_1$ and $E\gamma_2$ are premises of a nonredundant ground inference yielding C , where γ_1, γ_2 are substitutions, then there exists an inference with D and E as premises and whose conclusion is a generalization of C .

A difficulty arises on the nonground level because the calculus is optimized to avoid superposition *into* variables. For example, given the clause set $\{b \approx a, f(x) \not\approx c\}$, a superposition inference unifying b with x would yield the conclusion $f(a) \not\approx c$, but the calculus excludes this inference. Intuitively, since $f(a) \not\approx c$ is an instance of $f(x) \not\approx c$, we would expect the inference to be unnecessary, but this must be justified in general.

The Isabelle formalization relies on the first-order terms and related notions from the *IsaFoR* library [39]. It also uses the Isabelle version of the saturation framework [42]. The formalization validates the pen-and-paper proof: We found only one easy-to-repair mistake and one unnecessary assumption. The formalization can serve as a reference for refutational completeness of superposition, an important result in automated reasoning. It could also serve as the basis of a verified executable prover.

Ours is not the first formalization of superposition in a proof assistant, or even in Isabelle/HOL. Our predecessor is Peltier, who formalized a generalization of superposition and published his result in the *Archive of Formal Proofs (AFP)* [27]. However, his proof is monolithic, mixing ground and nonground aspects. By using the saturation framework, we get a clearer proof structure and immediately obtain the completeness of an abstract prover based on superposition [44, Lemma 10] as well as the completeness of various saturation procedures [44, Section 4].

Our Isabelle formalization and the underlying pen-and-paper proof are available online [17,43]. The formalization will soon be submitted to the *AFP*. Our work is part of the IsaFoL (Isabelle Formalization of Logic) effort [12].¹

2 Background

Prerequisites. We consider an untyped first-order logic with equality. A *term* is defined inductively as either a variable x or a function application $f(t_1, \dots, t_n)$ for a function symbol f and a (possibly empty) list of terms t_1, \dots, t_n . An *atom* is an unordered pair of terms, typically written as an equation $t \approx t'$. A *literal* is an atom $t \approx t'$ or a negated atom $t \not\approx t'$. A *clause* is a finite multiset $\{L_1, \dots, L_n\}$ of literals, typically written as a disjunction $L_1 \vee \dots \vee L_n$. The symbol \perp denotes the empty clause (or empty disjunction), which is false. All variables in a clause are to be understood as implicitly universally quantified in that clause.

A *context* κ is a term with one designated position that is to be filled by another term – in other words, a term with a hole. We use the syntax $\kappa[t]$ to represent the term consisting of a subterm t in a context κ . We write \square for the empty context.

Substitutions are total unary functions that let us replace variables with terms. We can apply a substitution σ to a syntactic entity X (e.g., a term or literal) by writing $X\sigma$. A substitution γ is a *grounding* substitution for a syntactic entity X if $X\gamma$ is ground, i.e., if it does not contain variables. A substitution ρ is a *renaming* if it is injective and $x\rho$ is a variable for every variable x . The composition of two substitutions σ_1 and σ_2 is defined as the function $\sigma_1 \circ \sigma_2 = (\lambda x. x\sigma_1\sigma_2)$. A substitution μ is an *idempotent most general unifier (IMGU)* for a set of terms T if μ is a unifier for T and $\mu \circ v = v$ for every unifier v for T .

An element x is *maximal* in a finite multiset \mathcal{X} w.r.t. a strict partial ordering \prec on \mathcal{X} if $x \in \mathcal{X} \wedge (\forall y \in \mathcal{X}. y \neq x \rightarrow x \not\prec y)$. An element x is *strictly maximal* in a finite multiset \mathcal{X} w.r.t. a strict partial ordering \prec on \mathcal{X} if $x \in \mathcal{X} \wedge (\forall y \in \mathcal{X} \setminus \{x\}. x \not\preceq y)$, where \preceq is the reflexive closure of \prec . The two notions coincide except for their handling of duplicates: A maximal element can have duplicates, whereas a strictly maximal element cannot. If the ordering is not total, a multiset can have multiple maximal or strictly maximal elements.

The Superposition Calculus. Bachmair and Ganzinger’s superposition calculus [2,3] belongs to a class of proof calculi for automatic provers known as saturation calculi. A saturation prover takes a set of formulas, usually clauses, as input and processes it by performing two operations: First, it derives new formulas from the old ones and adds them to the set. Second, it deletes superfluous formulas from the set. This process is repeated until the prover either finds \perp or reaches a state in which it is not required to add further formulas.

Abstractly, the calculus can be defined by two components: a set of inferences

$$\frac{C_n \cdots C_1}{C_0}$$

indicating that the formula C_0 (the *conclusion*) must be added to the set whenever the formulas C_n, \dots, C_1 (the *premises*) are already present, and a redundancy criterion that describes which inferences are unnecessary and which formulas may be deleted from the set.

¹ <https://github.com/IsaFoL/IsaFoL>

For the superposition calculus, the inferences are given by three schematic inference rules. The first one is

$$\frac{\overbrace{t \approx t' \vee D'}^D \quad \overbrace{\kappa[u] \bowtie u' \vee E'}^E}{\underbrace{(\kappa[t'\rho] \bowtie u' \vee E' \vee D'\rho)\mu}_C} \text{superposition}$$

where the clauses D and E are the premises, C is the conclusion, \bowtie is either \approx or $\not\approx$, u is a nonvariable subterm occurring in a context κ in clause E , ρ is an arbitrary but fixed renaming that is chosen so that $D\rho$ and E are variable-disjoint, and μ is an IMGU of $t\rho$ and u .

The other two rules are

$$\frac{\overbrace{t \not\approx t' \vee D'}^D}{\underbrace{D'\mu}_C} \text{equality resolution}$$

where μ is an IMGU of t and t' , and

$$\frac{\overbrace{u \approx u' \vee t \approx t' \vee D'}^D}{\underbrace{(u' \not\approx t' \vee u \approx t' \vee D')\mu}_C} \text{equality factoring}$$

where μ is an IMGU of t and u .

To reduce the number of inferences that need to be computed during the saturation, the inference rules above are equipped with ordering restrictions. Let \prec_t be an ordering on terms that is stable under grounding substitutions, and whose ground restriction is well-founded, total, and compatible with contexts, and has the subterm property. The term ordering \prec_t is extended to a literal ordering and a clause ordering in the following way: To every positive literal $t \approx t'$, we assign the multiset $\{t, t'\}$, to every negative literal $t \not\approx t'$, we assign the multiset $\{t, t, t', t'\}$. The literal ordering \prec_{lit} compares these multisets using the multiset extension of \prec_t . The clause ordering \prec_c compares clauses by comparing their multisets of literals using the multiset extension of \prec_{lit} .

We impose the following ordering restrictions on the inferences above: **(1)** If L is the first literal in a premise D or E , it must be maximal in that premise w.r.t. \prec_{lit} (after applying the substitution); **(2)** if additionally L is a positive equation in a superposition inference, it must be strictly maximal; **(3)** except in equality resolution inferences, the right-hand side of the equation or negated equation L may not be larger than or equal to the left-hand side w.r.t. \prec_t ; and **(4)** in superposition inferences, $D\rho\mu$ may not be larger than or equal to $C\mu$ w.r.t. \prec_c .

In the worst case, all literals in a clause can be incomparable and hence maximal. For clauses with negative literals, this effect can be remedied using a *selection function* that overrides the ordering restrictions. This is a function that maps every clause to a submultiset of its negative literals. The ordering conditions above are then modified so that if at least one literal in a clause is selected, then the maximality conditions for literals are applied to the selected submultiset instead of the original clause. This means that only inferences that involve literals that are maximal among the selected literals need to be performed.

These local restrictions are supplemented by a global redundancy criterion for clauses and inferences. Bachmair and Ganzinger's *standard redundancy criterion* is defined as follows: A ground clause C is redundant w.r.t. a set N of ground clauses if it is entailed by clauses in N that are smaller than C w.r.t. \prec_c . A nonground clause C is redundant w.r.t. a set N of nonground clauses if every ground instance of C is redundant w.r.t. the set of all ground instances of clauses in N . A ground inference (i.e., an inference with ground premises

and ground conclusion) is redundant w.r.t. a set N of ground clauses if its conclusion is entailed by clauses in N that are smaller than the maximal premise. A nonground inference is redundant w.r.t. a set N of nonground clauses if every ground instance of the inference is redundant w.r.t. the set of all ground instances of clauses in N .

Redundant clauses may be deleted from the clause set during a saturation; redundant inferences need not be computed. In particular, inferences whose conclusion is already contained in the clause set are always redundant.

The Saturation Framework. In their article in the *Handbook of Automated Reasoning* [4], Bachmair and Ganzinger gave a general account of components and properties of saturation calculi. The framework by Waldmann et al. [44] extended this to include a general treatment of lifting, subsumption, and prover architectures. We summarize the main results.

Let F be a set of formulas, and \models be a consequence relation on F . An F -inference is an inference with premises and conclusion in F . An F -inference system Inf is a set of F -inferences. If $N \subseteq F$, we write $Inf(N)$ for the set of all inferences in Inf with premises in N .

Let Red_1 be a function from sets of formulas to sets of inferences; let Red_F be a function from sets of formulas to sets of formulas. The pair $Red = \langle Red_1, Red_F \rangle$ is a *redundancy criterion* for Inf if it satisfies the following conditions:

1. if $N \models \{\perp\}$, then $N \setminus Red_F(N) \models \{\perp\}$;
2. if $N \subseteq N'$, then $Red_F(N) \subseteq Red_F(N')$ and $Red_1(N) \subseteq Red_1(N')$;
3. if $N' \subseteq Red_F(N)$, then $Red_F(N) \subseteq Red_F(N \setminus N')$ and $Red_1(N) \subseteq Red_1(N \setminus N')$; and
4. if the conclusion of an inference in Inf is in N , then the inference is in $Red_1(N)$.

Inferences in $Red_1(N)$ and formulas in $Red_F(N)$ are called redundant w.r.t. N .

A saturation prover for a calculus $\langle Inf, Red \rangle$ gets a set of formulas $N_0 \subseteq F$ as input and generates a sequence N_0, N_1, \dots of sets of formulas by adding newly computed formulas and by deleting unnecessary formulas. We require that in every step the deleted formulas are redundant w.r.t. the remaining ones. We call the sequence N_0, N_1, \dots a *derivation*. The set $N_\infty = \bigcup_i \bigcap_{j \geq i} N_j$ of persistent formulas is called the *limit* of the derivation. The derivation is *fair* if every inference from persistent formulas eventually becomes redundant. The calculus $\langle Inf, Red \rangle$ is *dynamically refutationally complete* if for every set N_0 with $N_0 \models \{\perp\}$ and every fair derivation N_0, N_1, \dots , the formula \perp is eventually derived, that is, $\perp \in \bigcup_i N_i$.

Proving the dynamic refutational completeness of the calculus $\langle Inf, Red \rangle$ directly is usually difficult. Fortunately, dynamic refutational completeness can be shown to be equivalent to another property, namely static refutational completeness: A set $N \subseteq F$ is *saturated* w.r.t. Inf and Red if $Inf(N) \subseteq Red_1(N)$. The calculus $\langle Inf, Red \rangle$ is *statically refutationally complete* if for every saturated set N we have that $N \models \perp$ implies $\perp \in N$.

To prove the static (and thus dynamic) refutational completeness of a calculus, it is usually convenient to start with a ground version of the calculus. The completeness result for the nonground calculus can then be obtained from the completeness result for the ground calculus by lifting, using a suitable *grounding* function that maps nonground formulas to sets of ground formulas and nonground inferences to sets of ground inferences. The framework also shows how to deal with redundancy criteria that are defined as intersections of other redundancy criteria (a technique that we will need to handle selection functions in the lifting process), how to integrate *subsumption* into the redundancy criterion (so that, e.g., $x \approx a$ makes its instance $b \approx a$ redundant), and how to obtain completeness results for implementations of the calculus in various prover architectures.

The framework has been formalized in Isabelle/HOL and extended by Tourret and Blanchette [14, 40, 42]. The present work builds on this formalization.

3 Proof Outline

Static refutational completeness can be stated as follows:

► **Theorem 1.** *For every set N that is saturated w.r.t. the superposition calculus, if N entails \perp , then $\perp \in N$.*

Equivalently: For every saturated set N such that $\perp \notin N$, there exists a model of N . Bachmair and Ganzinger’s original proof [3, Section 4] uses a monolithic approach. Our proof is more modular and proceeds in two clearly separated steps:

1. Given a ground clause set M saturated w.r.t. ground inferences, we build a model of M .
2. We show that if a clause set N is saturated w.r.t. nonground inferences, then its grounding $N_G = \{C\gamma \mid C \in N \text{ and } C\gamma \text{ is ground}\}$ is saturated w.r.t. ground inferences. Hence, by step 1, there exists a model of N_G , which is also a model of N .

In step 1, we construct a confluent and terminating term rewriting system R_∞ and use it to define an interpretation that equates all terms that share the same normal form w.r.t. R_∞ , and no others. For example, if $R_\infty = \{b \rightarrow a\}$, then the associated interpretation makes $f(b) \approx f(a)$ true and $c \approx a$ false. The system R_∞ is built incrementally. We start with $\{\}$ and traverse the clauses in M from the smallest clause following the ordering \prec_c . For each clause $C \in M$, if C is true in the current interpretation, there is nothing to do. Otherwise, we add a rewrite rule that attempts to make C true without affecting the truth of earlier, smaller clauses. While this process might fail in general, it will always produce a model of M if M is saturated.

In step 2, we must show that saturation on the nonground level implies saturation on the ground level. Via a result from the saturation framework, this amounts to showing that there exist nonground inferences corresponding to all nonredundant ground inferences of the calculus. A subtlety is that the calculus avoids superposition inferences into variables. Thus there might exist ground inferences that are not reflected on the nonground level. However, we can show that all such inferences are redundant. Another concern is the selection function. In general, we cannot assume that it is stable under substitutions, but without this assumption it is hard to relate the ground and nonground levels. The solution is provided by the saturation framework, which allows us to simultaneously lift all ground selection functions to the nonground level.

4 The Ground Proof

On the ground level, we reuse theories [23] from the `lsaFoR` project for terms, of type `'f gterm`, and term contexts, of type `'f gtxt`. The type variable `'f` represents function symbols. Isabelle types use a postfix notation. Ground atoms have type `'f gatom`, which is a synonym for `'f gterm uprod`, i.e., unordered pair of ground terms. Ground literals have type `'f gatom literal`. Ground clauses have type `'f gatom clause`, which is a synonym for `'f gatom literal multiset`. Isabelle multisets are always finite.

We start the formalization by introducing a locale, or module, that fixes an ordering on ground terms:

```

locale ground_ordering =
  fixes ( $\prec_t$ ) :: 'f gterm  $\Rightarrow$  'f gterm  $\Rightarrow$  bool
  assumes
    transp ( $\prec_t$ ) and asymp ( $\prec_t$ ) and totalp ( $\prec_t$ ) and wfp ( $\prec_t$ ) and
     $\forall \kappa :: 'f gtxt. \forall t_1 t_2. t_1 \prec_t t_2 \longrightarrow \kappa[t_1] \prec_t \kappa[t_2]$  and
     $\forall \kappa :: 'f gtxt. \forall t. \kappa \neq \square \longrightarrow t \prec_t \kappa[t]$ 

```


In Isabelle, a locale consists of parameters (here, \prec_t) that may depend on type variables (here, $'f$) paired with assumptions. Locales allow us to declare parameters and assumptions once and reuse them in multiple definitions and lemmas. When we later instantiate a locale, we must supply concrete arguments for the types and parameters and then discharge the proof obligations corresponding to the assumptions.

The locale `ground_ordering` assumes that the binary relation \prec_t is a well-founded total ordering, is compatible with ground term contexts, and has the subterm property.

Inside the locale context, we lift the term ordering and its properties to literals (\prec_{lit}) and clauses (\prec_c). We also configure the little-known order Isabelle proof method [38], a decision procedure for the quantifier-free theory of partial and total orderings, so that it can solve problems for our orderings.

As a building block for this formalization, we developed a generic theory of (strictly) minimal, (strictly) maximal, least, and greatest element in sets, finite sets, and finite multisets w.r.t. any partial or total ordering.

Next, we define the notion of selection function:

```

locale select =
  fixes sel :: 'a clause  $\Rightarrow$  'a clause
  assumes
     $\forall C. \text{sel } C \subseteq C$  and
     $\forall C. \forall L \in \text{sel } C. \text{is\_neg } L$ 

```

The locale `select` fixes a function `sel` for clauses with any atom type $'a$. In this section, we instantiate $'a$ with $'f \text{ gatom}$; Section 5 will instantiate it with its own atom type. Our assumptions on a selection function are that it always returns a submultiset of the argument C and only returns negative literals.

We can now assemble the parameters and assumption for the ground calculus:

```

locale ground_superposition_calculus = ground_ordering ( $\prec_t$ ) + select selG
for
  ( $\prec_t$ ) :: 'f gterm  $\Rightarrow$  'f gterm  $\Rightarrow$  bool and
  selG :: 'f gatom clause  $\Rightarrow$  'f gatom clause +
  assumes  $\forall R :: ('f \text{ gterm} \times 'f \text{ gterm}) \text{ set. ground\_critical\_pair\_theorem } R$ 

```

The locale `ground_superposition_calculus` extends both `ground_ordering` and `select`, inheriting all their assumptions as well as the definitions and theorems from their locale contexts. The parameters \prec_t and `selG` are provided with type annotations to control the instantiation of the type parameters. We also assume that the critical pair theorem [1, Theorem 6.2.4] holds for ground terms. As a sanity check, we proved this theorem in Isabelle by adapting a similar, but license-incompatible, result from the `IsaFoR` project [39].

We can now specify the ground version of the inference rules presented in Section 2, using inductive predicates. Compared with their nonground counterparts, the ground rules benefit from two simplifications. First, neither renamings nor unifiers are needed because ground terms contain no variables. Second, terms and clauses can be compared directly using \prec_t and \prec_c instead of using a reversed negated form since the orderings are total.

We show the ground superposition rule as an example. The rule notation below defines an inductive predicate `ground_superposition DEC` with the rule's premises D, E as assumptions and the rule's conclusion C as conclusion:

$$\frac{\overbrace{t \approx t' \vee D'}^D \quad \overbrace{\kappa[t] \bowtie u \vee E'}^E}{\underbrace{\kappa[t'] \bowtie u \vee D' \vee E'}_C} \text{ground_superposition } D E C$$

Side conditions:

1. $\bowtie \in \{\approx, \not\approx\}$;
2. $D \prec_c E$;
3. $t' \prec_t t$;
4. $u \prec_t \kappa[t]$;
5. if $\bowtie = \approx$, then $\text{sel}_G E = \{\}$ and $\kappa[t] \bowtie u$ is strictly maximal in E ;
6. if $\bowtie = \not\approx$, then $\text{sel}_G E = \{\}$ and $\kappa[t] \bowtie u$ is maximal in E or $\kappa[t] \bowtie u$ is maximal in $\text{sel}_G E$;
7. $\text{sel}_G D = \{\}$;
8. $t \approx t'$ is strictly maximal in D .

Following the structure required by the saturation framework, we define an inference system Inf_G and a consequence relation entails_G . For formulas, we use the type of ground clauses *'f gatom clause*, and for contradictions, we use the empty clause \perp .

► **Definition 2.** The set $\text{Inf}_G :: \text{'f gatom clause inference set}$ consists of all inferences of the ground superposition calculus:

$$\text{Inf}_G = \{ \langle [D, E], C \rangle \mid \text{ground_superposition } D E C \} \cup \{ \langle [D], C \rangle \mid \text{ground_eq_resolution } D C \} \cup \{ \langle [D], C \rangle \mid \text{ground_eq_factoring } D C \}$$

For the consequence relation, we reuse the theory of Herbrand interpretation developed for a formalization of ordered resolution [32]. This theory considers an interpretation to be a set of true atoms and defines the relation $\mathcal{I} \models_{\text{lit}} L$ expressing that the interpretation \mathcal{I} models the literal L , i.e., that L 's atom is in \mathcal{I} iff L is positive. The predicate is lifted to clauses (\models_c) and clause sets (\models) in the usual way.

Our ground atoms being unordered pairs of ground terms, our interpretations should be sets of *unordered* pairs. However, since Isabelle makes it easier to manipulate sets of *ordered* pairs, we use these as our interpretation and define a small wrapper with the help of the function $\text{uprod} :: 'a \times 'a \Rightarrow 'a \text{ uprod}$ to bridge the gap.

► **Definition 3.** The predicates $(\models_c) :: (\text{'f gterm} \times \text{'f gterm}) \text{ set} \Rightarrow \text{'f gatom clause} \Rightarrow \text{bool}$ and $(\models) :: (\text{'f gterm} \times \text{'f gterm}) \text{ set} \Rightarrow \text{'f gatom clause set} \Rightarrow \text{bool}$ express that an interpretation models a clause and a clause set, respectively:

$$\mathcal{I} \models_c C \iff \{ \text{uprod } r \mid r \in \mathcal{I} \} \models_c C \quad \mathcal{I} \models N \iff \{ \text{uprod } r \mid r \in \mathcal{I} \} \models N$$

We cannot use arbitrary sets of pairs as interpretations because the pairs should represent term equality. We require a valid interpretation \mathcal{I} to be reflexive, symmetric, and transitive and to be compatible with ground context application (i.e., $\forall \kappa :: \text{'f gtxt}. \forall t t'. \langle t, t' \rangle \in \mathcal{I} \longrightarrow \langle \kappa[t], \kappa[t'] \rangle \in \mathcal{I}$). We encode these requirements in the entails_G predicate:

► **Definition 4.** The predicate $\text{entails}_G :: \text{'f gatom clause set} \Rightarrow \text{'f gatom clause set} \Rightarrow \text{bool}$ expresses that a clause set N_1 entails another clause set N_2 , i.e., every valid interpretation of N_1 is also a valid interpretation of N_2 :

$$\begin{aligned} \text{entails}_G N_1 N_2 \iff & (\forall \mathcal{I} :: (\text{'f gterm} \times \text{'f gterm}) \text{ set}. \\ & \text{refl } \mathcal{I} \longrightarrow \text{sym } \mathcal{I} \longrightarrow \text{trans } \mathcal{I} \longrightarrow \text{compatible_with_gtxt } \mathcal{I} \longrightarrow \\ & \mathcal{I} \models N_1 \longrightarrow \mathcal{I} \models N_2) \end{aligned}$$

Equipped with Inf_G and entails_G , we can start to use the saturation framework. We first instantiate the `sound_inference_system` locale to make sure that the ground superposition calculus is sound and that our definitions correspond to what the framework expects:

sublocale `ground_superposition_calculus` \subseteq `sound_inference_system` **where**
 $\text{Inf} = \text{Inf}_G$ **and** $\text{Bot} = \{\perp\}$ **and** $\text{entails} = \text{entails}_G$

The sublocale notation means that definitions and theorems from `ground_superposition_calculus` are sufficient to prove the assumptions of `sound_inference_system` w.r.t. the given parameter instantiations. At this point, Isabelle requires us to actually prove the assumptions.

As the redundancy criterion, we reuse the standard redundancy criterion defined in the Isabelle saturation framework [14]:

sublocale `ground_superposition_calculus` \subseteq
`calculus_with_finitary_standard_redundancy` **where**
 $\text{Inf} = \text{Inf}_G$ **and** $\text{Bot} = \{\perp\}$ **and** $\text{entails} = \text{entails}_G$ **and** $\text{less} = (\prec_c)$
defines $\text{Red}_{IG} = \text{Red}_I$ **and** $\text{Red}_{FG} = \text{Red}_F$

The locale `calculus_with_finitary_standard_redundancy` defines the functions $\text{Red}_I :: 'f\text{gatom clause set} \Rightarrow 'f\text{gatom clause inference set}$, identifying redundant inferences, and $\text{Red}_F :: 'f\text{gatom clause set} \Rightarrow 'f\text{gatom clause set}$, identifying redundant formulas. We rename them to Red_{IG} and Red_{FG} , respectively.

To prove refutational completeness, we will exhibit a valid interpretation for a given saturated clause set. We build this interpretation by defining a confluent and terminating set of rewrite rules R_∞ , which we lift to an interpretation $\llbracket R_\infty \rrbracket^\downarrow$ that defines term equality. Each rewrite rule is a pair $\langle t, t' \rangle$, written $t \rightarrow t'$.

► **Definition 5.** The function $\llbracket \cdot \rrbracket :: ('f\text{gterm} \times 'f\text{gterm}) \text{ set} \Rightarrow ('f\text{gterm} \times 'f\text{gterm}) \text{ set}$ expands a rewrite rule set to all term contexts: $\llbracket R \rrbracket = \{\kappa[t] \rightarrow \kappa[t'] \mid t \rightarrow t' \in R\}$.

► **Definition 6.** The function $\cdot^\downarrow :: ('f\text{gterm} \times 'f\text{gterm}) \text{ set} \Rightarrow ('f\text{gterm} \times 'f\text{gterm}) \text{ set}$ produces the set of all term pairs considered equal w.r.t. a set of rewrite rules: $R^\downarrow = \{\langle t, t' \rangle \mid \exists t''. t \rightarrow t'' \in R^* \wedge t' \rightarrow t'' \in R^*\}$.

Now that we can lift a set of rewrite rules to a model, we define two mutually recursive functions that construct such a set for a given clause set.

► **Definition 7.** Let $N^{\prec_c D} = \{C \in N \mid C \prec_c D\}$ for any N and D . The functions $\text{epsilon} :: 'f\text{gatom clause set} \Rightarrow 'f\text{gatom clause} \Rightarrow ('f\text{gterm} \times 'f\text{gterm}) \text{ set}$ and $\text{rewrite_sys} :: 'f\text{gatom clause set} \Rightarrow ('f\text{gterm} \times 'f\text{gterm}) \text{ set}$ generate a term rewriting system for a given clause set:

$$\begin{aligned} \text{epsilon } N C &= \{t \rightarrow t' \mid \exists C'. C \in N \wedge C = (t \approx t' \vee C') \wedge \text{sel}_G C = \{\} \wedge \\ &\quad t \approx t' \text{ is strictly maximal in } C \wedge t' \prec_t t \wedge \\ &\quad \llbracket \text{rewrite_sys } N^{\prec_c C} \rrbracket^\downarrow \not\models_c C \wedge \\ &\quad \llbracket \text{rewrite_sys } N^{\prec_c C} \cup \{t \rightarrow t'\} \rrbracket^\downarrow \not\models_c C' \wedge \\ &\quad t \text{ is in normal form w.r.t. } \llbracket \text{rewrite_sys } N^{\prec_c C} \rrbracket^\downarrow\} \\ \text{rewrite_sys } N &= \bigcup_{C \in N} \text{epsilon } N C \end{aligned}$$

We reuse the definitions of joinability (\cdot^\downarrow) and of normal form (i.e., irreducibility) from a formalization of abstract rewriting systems [35].

The model construction iterates over the clause set, starting from the smallest clause following the ordering \prec_c , and collects a set of rewrite rules. At any point, we can use $\llbracket \cdot \rrbracket^\downarrow$ to obtain the candidate model. At each iteration, epsilon returns a set of rewrite rules that

12:10 A Modular Formalization of Superposition in Isabelle/HOL

are added to the term rewriting system: Either the considered clause is already true w.r.t. to the candidate model, in which case `epsilon` returns the empty set, or `epsilon` returns a single new rewrite rule that should make the clause true.

► **Example 8.** Assume \prec_t is the lexicographic path ordering with the precedence $a \prec b \prec c \prec d \prec e \prec f$. Let $N = \{d \approx c, b \approx a \vee e \not\approx c, b \not\approx b \vee f(b) \approx a, f(c) \approx b, f(b) \approx a \vee f(c) \not\approx b, f(b) \approx a \vee f(d) \not\approx b\}$ be a clause set saturated w.r.t. the ground superposition calculus. The following table shows the result of each iteration of the model construction:

Iteration	Clause C	<code>rewrite_sys</code> $N^{\prec_c C}$	<code>epsilon</code> $N C$
1	$d \approx c$	$\{\}$	$\{d \rightarrow c\}$
2	$b \approx a \vee e \not\approx c$	$\{d \rightarrow c\}$	$\{\}$
3	$b \not\approx b \vee f(b) \approx a$	$\{d \rightarrow c\}$	$\{f(b) \rightarrow a\}$
4	$f(c) \approx b$	$\{d \rightarrow c, f(b) \rightarrow a\}$	$\{f(c) \rightarrow b\}$
5	$f(b) \approx a \vee f(c) \not\approx b$	$\{d \rightarrow c, f(b) \rightarrow a, f(c) \rightarrow b\}$	$\{\}$
6	$f(b) \approx a \vee f(d) \not\approx b$	$\{d \rightarrow c, f(b) \rightarrow a, f(c) \rightarrow b\}$	$\{\}$

At each iteration $i + 1$, the term rewriting system consists of the union of the term rewriting system of iteration i and the “epsilon” of iteration i . As expected, the interpretation after iteration 6 is a model of N .

The conditions on rewrite rule production were chosen so that the term rewriting system is confluent. Specifically, we prove strong normalization and the weak Church–Rosser property, which together imply the Church–Rosser property, which is equivalent to confluence.

► **Lemma 9.** *Let N be a ground clause set. The term equality specified by $\llbracket \text{rewrite_sys } N \rrbracket^\downarrow$ is reflexive, symmetric, transitive, and compatible with ground contexts.*

Now that we can build a valid interpretation for a clause set, it remains to show that it satisfies all clauses from this set. We first need a pair of lemmas that express monotonicity properties of the construction:

► **Lemma 10.** *Let N be a ground clause set and C be a ground clause. If $\text{epsilon } N C = \{t \rightarrow t'\}$, then*

1. $\llbracket \text{rewrite_sys } N \rrbracket^\downarrow \models_c C$;
2. $\forall D \in N. C \prec_c D \rightarrow \llbracket \text{rewrite_sys } N^{\prec_c D} \rrbracket^\downarrow \models_c C$;
3. $\llbracket \text{rewrite_sys } N \rrbracket^\downarrow \not\models_c C \setminus \{t \approx t'\}$; and
4. $\forall D \in N. C \prec_c D \rightarrow \llbracket \text{rewrite_sys } N^{\prec_c D} \rrbracket^\downarrow \not\models_c C \setminus \{t \approx t'\}$.

► **Lemma 11.** *Let N be a ground clause set and $C \in N$ be a ground clause. If $\llbracket \text{rewrite_sys } N^{\prec_c C} \rrbracket^\downarrow \models_c C$, then*

1. $\llbracket \text{rewrite_sys } N \rrbracket^\downarrow \models_c C$ and
2. $\forall D \in N. C \prec_c D \rightarrow \llbracket \text{rewrite_sys } N^{\prec_c D} \rrbracket^\downarrow \models_c C$.

We can now prove that our model construction works for all clauses.

► **Lemma 12.** *Let N be a saturated ground clause set and $C \in N$ be a ground clause. If $\perp \notin N$, then*

1. $\text{epsilon } N C = \{\} \leftrightarrow \llbracket \text{rewrite_sys } N^{\prec_c C} \rrbracket^\downarrow \models_c C$ and
2. $\forall D \in N. C \prec_c D \rightarrow \llbracket \text{rewrite_sys } N^{\prec_c D} \rrbracket^\downarrow \models_c C$.

Proof Sketch. By well-founded induction w.r.t. \prec_c . ◀

► **Lemma 13 (Ground Model Construction).** *Let N be a saturated ground clause set and $C \in N$ be a ground clause. If $\perp \notin N$, then $\llbracket \text{rewrite_sys } N \rrbracket^\downarrow \models_c C$.*

Proof Sketch. By Lemmas 11 and 12 if $\text{epsilon } N C = \{\}$ and Lemma 10 otherwise. ◀

► **Theorem 14** (Ground Refutational Completeness). *Let N be a saturated ground clause set. If $\text{entails}_G N \{\perp\}$, then $\perp \in N$.*

Proof Sketch. We assume $\perp \notin N$ and show $\neg \text{entails}_G N \{\perp\}$. We must show the existence of an interpretation \mathcal{I} such that **(1)** \mathcal{I} is reflexive, symmetric, transitive, and compatible with ground contexts; **(2)** $\mathcal{I} \models N$; and **(3)** $\mathcal{I} \not\models_c \perp$. We let $\mathcal{I} = \llbracket \text{rewrite_sys } N \rrbracket^\perp$. Step 1 follows from Lemma 9. Step 2 follows from Lemma 13. Step 3 follows from the definition of \models_c . ◀

Finally, we can provide our main result for the ground calculus by instantiating the locale `statically_complete_calculus` from the saturation framework:

```
sublocale ground_superposition_calculus ⊆ statically_complete_calculus where
  Inf = InfG and Bot = {⊥} and entails = entailsG and less = (<c) and
  RedI = RedIG and RedF = RedFG
```

We use Theorem 14 to discharge the proof obligation.

5 The Nonground Proof

On the nonground level, we reuse theories [36] from the `IsaFoR` project for terms, of type (f, v) *term*, and term contexts, of type (f, v) *ctxt*. The type variable v represents term variables. Atoms have type (f, v) *atom*, which is a synonym for (f, v) *term uprod*. Literals and clauses (type (f, v) *atom clause*) are defined analogously to ground literals and clauses. Substitutions have type $v \Rightarrow (f, v)$ *term*.

► **Definition 15.** The predicates $\text{is_ground}_t :: (f, v)$ *term* \Rightarrow *bool* and $\text{is_ground}_{\text{ctxt}} :: (f, v)$ *ctxt* \Rightarrow *bool* express that the given term or context does not contain any variables. We lift is_ground_t to substitutions ($\text{is_ground}_{\text{subst}}$), atoms (is_ground_a), literals ($\text{is_ground}_{\text{lit}}$), and clauses (is_ground_c).

► **Definition 16.** The function $\text{groundings}_c :: (f, v)$ *atom clause* \Rightarrow *f gatom clause set* maps a clause to the set of its groundings: $\text{groundings}_c C = \{C\gamma \mid \text{is_ground}_c(C\gamma)\}$.

The calculus is parameterized by a well-founded total ordering on ground terms as defined in the locale `ground_ordering`. We introduce a locale for the nonground ordering:

```
locale first_order_ordering =
  fixes (<t) :: (f, v) term ⇒ (f, v) term ⇒ bool
  assumes
    transp (<t) and asymp (<t) and
    totalp_on {t | is_ground_t t} (<t) and wfp_on {t | is_ground_t t} (<t) and
    ∀κ :: (f, v) ctxt. ∀t1 t2. is_ground_t t1 → is_ground_t t2 → is_ground_ctxt κ →
      t1 <t t2 → κ[t1] <t κ[t2] and
    ∀κ :: (f, v) ctxt. ∀t. is_ground_t t → is_ground_ctxt κ → κ ≠ □ → t <t κ[t] and
    ∀t1 t2 γ. is_ground_t (t1γ) → is_ground_t (t2γ) → t1 <t t2 → t1γ <t t2γ
```

We assume transitivity and asymmetry on the entire relation. We assume totality and well-foundedness on ground terms using `totalp_on` and `wfp_on`. We also assume compatibility with ground contexts and the subterm property for ground terms. Finally, we assume *stability under grounding substitutions*: If two terms are $<_t$ -related, then they are still $<_t$ -related after applying a grounding substitution. Most of the restrictions to ground terms are not necessary for practical orders such as the Knuth–Bendix ordering and the lexicographic path ordering, but we prefer the additional generality.

12:12 A Modular Formalization of Superposition in Isabelle/HOL

In the locale context of `first_order_ordering`, we define \prec_{tG} as \prec_t on ground terms. We lift \prec_t to literals (\prec_{lit}) and clauses (\prec_c). We then prove transitivity, asymmetry, totality, well-foundedness, and stability under grounding substitutions for \prec_{lit} and \prec_c .

The next lemma will be useful to prove that nonground inferences overapproximate ground inferences:

► **Lemma 17.** *Let C be a clause, $L \in C$ a literal, and γ a grounding substitution for C . If $L\gamma$ is (strictly) maximal in $C\gamma$, then L is (strictly) maximal in C .*

Next to the ordering, the calculus is also parameterized by a selection function. Let $(f, 'v)$ *select* abbreviate $(f, 'v)$ *atom clause* \Rightarrow $(f, 'v)$ *atom clause* and $'fgselect$ abbreviate $'fgatom clause \Rightarrow 'fgatom clause$. We define

```
locale first_order_select = select sel
  for sel :: ('f, 'v) select
```

The locale `first_order_select` extends the locale `select` and fixes the type of its selection function `sel`. We use this locale to prove some lemmas regarding `sel` and grounding substitutions. The blueprint also initially assumed *stability under renaming*: $\text{sel}(C\rho) = (\text{sel } C)\rho$ for every clause C and every renaming ρ . This assumption was taken from the formal completeness proof for the resolution calculus [30], but it turns out to be unnecessary in our formalization because we use a different approach to lift ground inferences.

The following predicate is useful to lift selection functions:

► **Definition 18.** The predicate $\text{is_grounding}_S :: ('f, 'v) \text{select} \Rightarrow 'fgselect \Rightarrow \text{bool}$ relates two selection functions, S for nonground clauses and S_G for ground clauses: $\text{is_grounding}_S S S_G \longleftrightarrow \forall C_G :: 'fgatom clause. \exists C :: ('f, 'v) atom clause. \exists \gamma. C_G = C\gamma \wedge S_G C_G = (S C)\gamma$.

Using is_grounding_S , we define the following in the context of the `first_order_select` locale:

► **Definition 19.** The set $\text{gselects} :: 'fgselect \text{ set}$ consists of all ground selection functions related to `sel`: $\text{gselects} = \{S_G \mid \text{is_grounding}_S \text{ sel } S_G\}$.

Based on `gselects`, we lift the ground calculus for all ground selection functions. The following locale associates a selection function `sel` with a grounding `selG`:

```
locale grounded_first_order_select = first_order_select sel
  for sel :: ('f, 'v) select +
  fixes sel_G :: 'fgselect
  assumes is_grounding_S sel sel_G
```

We show that the grounding `selG` fulfills the criteria for a ground-level selection function:

```
sublocale grounded_first_order_select  $\subseteq$  select where
  sel = sel_G
```

This sublocale relation establishes the lifting for the selection function.

We continue by creating the main building block for the nonground calculus:

```
locale first_order_superposition_calculus =
  first_order_ordering ( $\prec_t$ ) + first_order_select sel
  for
    ( $\prec_t$ ) :: ('f, 'v) term  $\Rightarrow$  ('f, 'v) term  $\Rightarrow$  bool and
    sel :: ('f, 'v) select +
```

fixes tiebreakers :: $'f$ gatom clause $\Rightarrow ('f, 'v)$ atom clause $\Rightarrow ('f, 'v)$ atom clause \Rightarrow bool
 assumes
 infinite (UNIV :: $'v$ set) and
 $\forall C_G. \text{wfp}(\text{tiebreakers } C_G) \wedge \text{transp}(\text{tiebreakers } C_G) \wedge \text{asyp}(\text{tiebreakers } C_G)$ and
 $\forall R :: ('f \text{ gterm} \times 'f \text{ gterm}) \text{ set. ground_critical_pair_theorem } R$

We define the locale `first_order_superposition_calculus` extending `first_order_ordering` and `first_order_select`. We need three additional assumptions: First, we assume that the set of all variables (UNIV :: $'v$ set) is infinite, so that we can generate enough fresh variables for renamings. Second, we support tiebreakers, i.e., a family of well-founded partial orderings indexed by ground clauses. The orderings can be used to implement subsumption. Third, we assume the theorem of ground critical pairs as in Section 4.

Next, we define the three inference rules `superposition`, `eq_resolution`, and `eq_factoring` presented in Section 2 inside `first_order_superposition_calculus` using inductive predicates. The `superposition` rule follows as an example:

$$\frac{\overbrace{t \approx t' \vee D'}^D \quad \overbrace{\kappa[u] \bowtie u' \vee E'}^E}{\underbrace{((\kappa\rho_2)[t'\rho_1] \bowtie u'\rho_2 \vee D'\rho_1 \vee E'\rho_2)\mu}_C} \text{superposition } D E C$$

Side conditions:

1. $\bowtie \in \{\approx, \not\approx\}$;
2. ρ_1 and ρ_2 are renamings;
3. $D\rho_1$ and $E\rho_2$ are variable-disjoint;
4. u is not a variable;
5. μ is an IMGU of $\{t\rho_1, u\rho_2\}$;
6. $E\rho_2\mu \not\prec_c D\rho_1\mu$;
7. $t\rho_1\mu \not\prec_t t'\rho_1\mu$;
8. $(\kappa[u])\rho_2\mu \not\prec_t u'\rho_2\mu$;
9. if $\bowtie = \approx$, then $\text{sel } E = \{\}$ and $(\kappa[u] \bowtie u')\rho_2\mu$ is strictly maximal in $E\rho_2\mu$;
10. if $\bowtie = \not\approx$, then $\text{sel } E = \{\}$ and $(\kappa[u] \bowtie u')\rho_2\mu$ is maximal in $E\rho_2\mu$ or $(\kappa[u] \bowtie u')\rho_2\mu$ is maximal in $(\text{sel } E)\rho_2\mu$;
11. $\text{sel } D = \{\}$;
12. $(t \approx t')\rho_1\mu$ is strictly maximal in $D\rho_1\mu$.

Unlike in the blueprint, we do not fix functions to create the renamings and the IMGUs. Instead, we use predicates that describe the properties of renamings and IMGUs. This gives more flexibility to a saturation prover based on the calculus, which could, for example, use a nondeterministic implementation.

► **Definition 20.** The set $\text{Inf}_F :: ('f, 'v)$ atom clause inference set consists of all inferences of the superposition calculus:

$$\text{Inf}_F = \{\langle [D, E], C \rangle \mid \text{superposition } D E C\} \cup \{\langle [D], C \rangle \mid \text{eq_resolution } D C\} \cup \{\langle [D], C \rangle \mid \text{eq_factoring } D C\}$$

Concluding the setup for the lifting of the calculus, we define

`locale grounded_first_order_superposition_calculus =`
`first_order_superposition_calculus + grounded_first_order_select`

12:14 A Modular Formalization of Superposition in Isabelle/HOL

By combining `first_order_superposition_calculus` and `grounded_first_order_select`, we provide an arbitrary ground select function `selG` to the nonground calculus. The resulting locale `grounded_first_order_superposition_calculus` has all the required assumptions to instantiate `ground_superposition_calculus`:

```

sublocale grounded_first_order_superposition_calculus  $\subseteq$ 
  ground_superposition_calculus where
  selG = selG and  $\prec_t = \prec_{tG}$ 

```

The selection function causes some complications. In general, `sel` is not stable under substitutions (i.e., `(sel C)σ` and `sel (Cσ)` might be different). As a result, we cannot directly use it at the ground level. Based on `sel` and a saturated set N , we would want to define a suitable ground selection function S_G . However, this definition cannot work, because N is not a priori known. The solution is as follows: For all ground selection functions in `gselects`, we lift the corresponding ground calculi to the nonground level and consider all of them together. This ensures that we perform the right lifting regardless of N .

The locale `lifting_intersection` [42, Section 3.1] of the saturation framework enables us to lift a family of ground calculi indexed by `gselects`. We instantiate the locale as a sublocale of `first_order_superposition_calculus`. Since we lift a ground calculus family and not a single calculus, we cannot have a global `ground_superposition_calculus`. However, once we have an arbitrary but fixed ground selection function $S_G \in \text{gselects}$, we can use Isabelle's facility for instantiating locales locally in a proof using `interpret`.

The locale `lifting_intersection` gives us the following lifted definition of the entailment relation for nonground clause sets:

► **Definition 21.** The predicate `entailsF` :: (f, v) atom clause set \Rightarrow (f, v) atom clause set \Rightarrow *bool* expresses that a clause set N_1 entails another clause set N_2 : `entailsF N1 N2` \longleftrightarrow `entailsG ($\bigcup_{C \in N_1}$ groundingsc C) ($\bigcup_{C \in N_2}$ groundingsc C)`.

Next, we ensure that the nonground calculus is sound and compatible with the saturation framework by instantiating the `sound_inference_system` locale:

```

sublocale first_order_superposition_calculus  $\subseteq$  sound_inference_system where
  Inf = InfF and Bot = { $\perp$ } and entails = entailsF

```

The locale `lifting_intersection` provides a lifted redundancy criterion $\langle \text{Red}_I, \text{Red}_F \rangle$ supporting full subsumption based on tiebreakers. Additionally, it reduces our proof of static refutational completeness for `first_order_superposition_calculus` to two easier proof obligations: First, every member of the ground calculus family is statically refutationally complete. We can prove this directly by the main result of Section 4. Second, there exists a selection function `grounding` that is indexing a member of the ground calculus family, with which the nonground inferences overapproximate all ground inferences. We prove this below.

► **Definition 22.** The function `groundingsInf` :: (f, v) atom clause inference \Rightarrow f gatom clause inference set maps an inference to the set of ground inferences that can arise by grounding its premises and conclusions: `groundingsInf ι` = $\{\iota\gamma \mid \iota\gamma \in \text{Inf}_G\}$.

► **Lemma 23** (Equality Resolution Lifting). *Let C and D be clauses, γ be a grounding substitution for C and D , and ι_G be a ground inference. If `selG (Dγ) = (sel D)γ` and $\iota_G = \text{ground_eq_resolution} (D\gamma) (C\gamma)$, then there exist C' and ι such that $C\gamma = C'\gamma$, $\iota = \text{eq_resolution} D C'$, and $\iota_G \in \text{groundings}_{\text{Inf}} \iota$.*

Note that the first assumption further restricts the relation between sel and sel_G . We will see in Lemmas 26 and 27 how we can discharge this assumption within the locale.

Proof Sketch. We first deconstruct the premise D using the properties of its grounding $D\gamma$ induced by the ground inference to put it in the correct form for the nonground inference. Then we construct a matching conclusion C' and show that $C\gamma = C'\gamma$. The more restricted relation between sel and sel_G is required to lift the side condition about selection. ◀

► **Lemma 24** (Equality Factoring Lifting). *Let C and D be clauses, γ be a grounding substitution for C and D , and ι_G be a ground inference. If $\text{sel}_G(D\gamma) = (\text{sel } D)\gamma$ and $\iota_G = \text{ground_eq_factoring}(D\gamma)(C\gamma)$, then there exist C' and ι such that $C\gamma = C'\gamma$, $\iota = \text{eq_factoring } D C'$, and $\iota_G \in \text{groundings}_{\text{Inf}} \iota$.*

Proof Sketch. Analogous to Lemma 23. ◀

► **Lemma 25** (Superposition Lifting). *Let C, D, E be clauses, γ be a grounding substitution for C, D, E , ρ_1, ρ_2 be renamings such that $D\rho_1$ and $E\rho_2$ are variable-disjoint, and ι_G be a ground inference. If $\text{sel}_G(D\rho_1\gamma) = (\text{sel}(D\rho_1))\gamma$, $\text{sel}_G(E\rho_2\gamma) = (\text{sel}(E\rho_2))\gamma$, $\iota_G = \text{ground_superposition}(D\rho_1\gamma)(E\rho_2\gamma)(C\gamma)$, and $\iota_G \notin \text{Red}_{\iota_G}(\text{groundings}_c D \cup \text{groundings}_c E)$, then there exist C' and ι such that $C\gamma = C'\gamma$, $\iota = \text{superposition } DEC'$, and $\iota_G \in \text{groundings}_{\text{Inf}} \iota$.*

Compared with Lemmas 23 and 24, there are two additions: First, nonground superposition inferences require their premises to be variable-disjoint. Therefore, the lemma is parameterized by renamings ρ_1 and ρ_2 . Second, we assume that ι_G is not redundant. This is unproblematic: The lemma is used only to prove that nonground inferences overapproximate ground inferences, and there we need the lifting only in the nonredundant case.

Proof Sketch. The proof is similar to those of the previous two lemmas. A subtlety is that superposition avoids inferences into variables (side condition 4). We must show that ground inferences whose lifting would result in an inference into a variable are redundant according to Red_{ι_G} . We sketch the proof with two examples:

- Consider the saturated clause set $N = \{b \approx a, g(x) \not\approx d\}$. We must show that there are no nonredundant inferences from the set of its groundings $N_G = \{b \approx a, g(a) \not\approx d, g(b) \not\approx d, g(c) \not\approx d, \dots\}$. However, we can derive the clause $g(a) \not\approx d$ using $b \approx a$ and $g(b) \not\approx d$. Fortunately, the inference is redundant since $g(a) \not\approx d$ is already contained in N_G .
- What if we have a clause with multiple occurrences of the same variable, as in $N' = \{b \approx a, g(x) \not\approx f(x)\}$? We then have $N'_G = \{b \approx a, g(a) \not\approx f(a), g(b) \not\approx f(b), g(c) \not\approx f(c), \dots\}$ and can use $b \approx a$ and $g(b) \not\approx f(b)$ to generate $g(a) \not\approx f(b)$. However, since $\{b \approx a, g(a) \not\approx f(a)\} \models \{g(a) \not\approx f(b)\}$ and $g(a) \not\approx f(a) \prec_c g(a) \not\approx f(b)$, this inference is also redundant.

The formal proof performs multiple inductions over the number of occurrences of variables. ◀

While proving the above lemmas in Isabelle, we discovered a mistake in the formulation of our blueprint. We had wrongly stated the conclusion of Lemma 23 as: “Then there exists $\iota = \text{eq_resolution } D C$ and \dots ” We had fixed the conclusions of the nonground inferences to C , even though many clauses can result in the same grounding w.r.t. γ . The same issue arose in Lemmas 24 and 25.

► **Lemma 26.** *Let N be a clause set, N_G be the set containing all groundings of all clauses of N , and S be an arbitrary function of type $(f, 'v)$ select. Then there exists a function S_G of type f gselect such that $\text{is_grounding}_S S S_G$ and $\forall C_G \in N_G. \exists C \gamma. C \in N \wedge C_G = C\gamma \wedge S_G C_G = (S C)\gamma$.*

Proof Sketch. We construct a S_G on N_G such that it fulfills the last property. This is possible since the elements of N_G are the groundings of the elements of N . It follows directly from the definition of groundings_c that S_G on N_G is a grounding of S . For clauses not in N_G , we define S_G as the ground restriction of S . ◀

► **Lemma 27 (Overapproximation).** *Let N be a clause set. Then there exists a ground calculus with a ground selection function S_G such that nonground inferences from N overapproximate all ground inferences from the groundings of N , meaning that each ground inference from the groundings $_c$ of N is either contained in the $\text{groundings}_{\text{Inf}}$ of inferences from N or redundant.*

Proof Sketch. The proof follows from the lifting lemmas (Lemmas 23–25), for which we can obtain the required ground selection functions using Lemma 26. For Lemma 25, we also need the assumption that the set of all variables is infinite to be able to provide the renamings. ◀

Using Lemma 27, we can conclude our endeavor and instantiate `statically_complete_calculus`:

```
sublocale first_order_superposition_calculus ⊆ statically_complete_calculus where
  Inf = InfF and Bot = {⊥} and entails = entailsF and less = (<c) and
  RedI = RedI and RedF = RedF
```

We have verified the static refutational completeness of first-order superposition. The saturation framework provides us with a proof of dynamic refutational completeness.

Finally, to exclude any inconsistency in our assumptions, we instantiate the locale `first_order_superposition_calculus` with a trivial select function, trivial tiebreakers, and the verified Knuth–Bendix ordering [37]. We can discharge all proof obligations. Since `first_order_superposition_calculus` transitively instantiates all the other locales, we cover all our assumptions.

6 Related Work

The saturation framework [44] has been used in the completeness proof of several new variants of superposition:

- Boolean λ -superposition [8] for higher-order logic, as well as its predecessors Boolean-free λ -superposition [9] and Boolean-free λ -free superposition [7] that operate on fragments of higher-order logic.
- superposition with delayed unification [11] for first-order logic, which adds constraints to the conclusions of inferences instead of performing full unifications.

An extended abstract by Tourret [41] discusses how these use the framework. The work described in the present paper could serve as a foundation to formalize these proofs.

The Isabelle/HOL formalization of the saturation framework was introduced together with an instance of the resolution calculus and an abstract resolution prover called RP by Tourret and Blanchette [42]. Other theorem proving techniques formalized in Isabelle/HOL include an executable SAT solver by Blanchette et al. [13] based on CDCL (conflict-driven clause learning) for propositional logic with state-of-the-art optimizations, various sequent and tableau calculi for first-order and related logics by From et al. [19, 20], and another version of resolution and RP by Schlichtkrull et al. [30] following Bachmair and Ganzinger’s original, more ad hoc proof that was extended to an executable prover [31]. Most recently, the newly created SCL calculus [18], which follows a CDCL-like approach to theorem proving in first-order logic, was also verified in Isabelle/HOL by Bromberger et al. [15] as it was being developed. Also relevant here is Paulson’s formalization of Gödel’s incompleteness theorems [25, 26].

Isabelle/HOL is possibly the most widely used system for formalizing automated reasoning results, but other proof assistants are used as well. Early results include Shankar’s proof of Gödel’s first incompleteness theorem in Nqthm [34], Persson’s completeness proof for intuitionistic predicate logic in ALF [28], and Harrison’s formalization of basic first-order model theory in HOL Light [21]. We refer to Blanchette [12, Section 5] for a survey.

Finally, the work closest to ours, already mentioned in the introduction, is the formalization of a variant of the superposition calculus in Isabelle/HOL by Peltier [27]. Our initial intent was to integrate his calculus with the saturation framework, but after months of fruitless attempts, we decided to start from scratch, which resulted in the present work.

A first obstacle we encountered was related to Peltier’s redundancy criterion. He relies on a notion that is sufficient to prove static refutational completeness but cannot be lifted to dynamic completeness because his redundancy is defined in terms of smaller or equal clauses rather than strictly smaller clauses. This makes it unsuitable for use in the saturation framework, but we managed to replace it with a suitable criterion without changing the calculus, allowing us to pursue our work in this direction for a while.

What made us switch approach was an incompatibility requiring a major modification of Peltier’s formalization itself. Peltier works with closures, i.e., pairs $\langle C, \sigma \rangle$ consisting of a *set* of literals C and a substitution σ . This calculus is defined directly on the nonground level, where static completeness is proved. For integration into the saturation framework, we wanted a ground version of the calculus, which we obtained by restricting the substitutions to groundings only and operating on clauses as sets of ground literals $C\sigma$. However, this made it impossible to overapproximate this calculus with Peltier’s calculus on the nonground level, which is needed for the lifting to be possible in the framework. The issue is that we do not want to match a literal K in a ground clause to two literals L_1, L_2 in a nonground closure $\langle L_1 \vee L_2 \vee C, \sigma \rangle$ such that $L_1\sigma = L_2\sigma = K$, because this breaks the lifting. Fixing this would require working directly on closures also at the ground level. A new proof of ground refutational completeness would have had to be provided for this new calculus. It seemed more convenient to formalize a calculus operating on *multisets* of literals instead of closures, especially that the Isabelle multiset library was already well developed for use in theorem proving formalization.

Our formalization consists of 12 000 nonblank lines, 7000 of which are for nonbackground theories. For comparison, Peltier’s formalization consists of 9000 nonblank lines, 7000 of which are for nonbackground theories. All numbers are rounded to the nearest thousand. Interestingly, the two formalizations have approximately the same size even though they are written in very different styles. To our surprise, the additional modularity of our work did not lead to a shorter proof.

7 Conclusion

We restructured the refutational completeness proof of superposition using the saturation framework. We first proved refutational completeness for the ground calculus and lifted the proof to the full, nonground calculus. Next, we formalized this pen-and-paper proof in Isabelle/HOL. The formalization can be seen as a case study for the *IsaFoR* library and the saturation framework, as well as for basic Isabelle tools such as *locales*, which facilitate modularity and proof reuse.

We see three main directions for future work. First, the proof could be extended to support simply typed or rank-1-polymorphic first-order terms. Second, the completeness proofs of variants of superposition, such as hierarchic superposition [5, 6], combinatory superposition [10], and λ -superposition [8], could be formalized as well. Third, the formalization of superposition (or that of variants) could be extended to obtain a verified executable prover.

References

- 1 Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.
- 2 Leo Bachmair and Harald Ganzinger. On restrictions of ordered paramodulation with simplification. In Mark E. Stickel, editor, *CADE-10*, volume 449 of *LNCS*, pages 427–441. Springer, 1990. doi:10.1007/3-540-52885-7_105.
- 3 Leo Bachmair and Harald Ganzinger. Rewrite-based equational theorem proving with selection and simplification. *J. Log. Comput.*, 4(3):217–247, 1994.
- 4 Leo Bachmair and Harald Ganzinger. Resolution theorem proving. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, volume I, pages 19–99. Elsevier and MIT Press, 2001.
- 5 Leo Bachmair, Harald Ganzinger, and Uwe Waldmann. Theorem proving for hierarchic first-order theories. In H el ene Kirchner and Giorgio Levi, editors, *ALP '92*, volume 632 of *LNCS*, pages 420–434. Springer, 1992.
- 6 Peter Baumgartner and Uwe Waldmann. Hierarchic superposition revisited. In Carsten Lutz, Uli Sattler, Cesare Tinelli, Anni-Yasmin Turhan, and Frank Wolter, editors, *Description Logic, Theory Combination, and All That—Essays Dedicated to Franz Baader on the Occasion of His 60th Birthday*, volume 11560 of *LNCS*, pages 15–56. Springer, 2019.
- 7 Alexander Bentkamp, Jasmin Blanchette, Simon Cruanes, and Uwe Waldmann. Superposition for lambda-free higher-order logic. *Log. Methods Comput. Sci.*, 17(2), 2021.
- 8 Alexander Bentkamp, Jasmin Blanchette, Sophie Tourret, and Petar Vukmirovic. Superposition for higher-order logic. *J. Autom. Reason.*, 67(1):10, 2023.
- 9 Alexander Bentkamp, Jasmin Blanchette, Sophie Tourret, Petar Vukmirovic, and Uwe Waldmann. Superposition with lambdas. *J. Autom. Reason.*, 65(7):893–940, 2021.
- 10 Ahmed Bhayat and Giles Reger. A combinator-based superposition calculus for higher-order logic. In Nicolas Peltier and Viorica Sofronie-Stokkermans, editors, *IJCAR 2020, Part I*, volume 12166 of *LNCS*, pages 278–296. Springer, 2020.
- 11 Ahmed Bhayat, Johannes Schoisswohl, and Michael Rawson. Superposition with delayed unification. In Brigitte Pientka and Cesare Tinelli, editors, *CADE-29*, volume 14132 of *LNCS*, pages 23–40. Springer, 2023.
- 12 Jasmin Christian Blanchette. Formalizing the metatheory of logical calculi and automatic provers in Isabelle/HOL (invited talk). In Assia Mahboubi and Magnus O. Myreen, editors, *CPP 2019*, pages 1–13. ACM, 2019.
- 13 Jasmin Christian Blanchette, Mathias Fleury, Peter Lammich, and Christoph Weidenbach. A verified SAT solver framework with learn, forget, restart, and incrementality. *J. Autom. Reason.*, 61(3):333–366, 2018.
- 14 Jasmin Christian Blanchette and Sophie Tourret. Extensions to the comprehensive framework for saturation theorem proving. *Archive of Formal Proofs*, 2020. URL: https://isa-afp.org/entries/Saturation_Framework_Extensions.html.
- 15 Martin Bromberger, Martin Desharnais, and Christoph Weidenbach. An Isabelle/HOL formalization of the SCL(FOL) calculus. In Brigitte Pientka and Cesare Tinelli, editors, *CADE-29*, volume 14132 of *LNCS*, pages 116–133. Springer, 2023.
- 16 Simon Cruanes. *Extending Superposition with Integer Arithmetic, Structural Induction, and Beyond*. Ph.D. thesis,  cole polytechnique, 2015.
- 17 Martin Desharnais and Balazs Toth. Superposition calculus, 2024. URL: https://github.com/IsaFoL/IsaFoL/tree/ITP2024-IsaSuperposition/Superposition_Calculus.
- 18 Alberto Fiori and Christoph Weidenbach. SCL clause learning from simple models. In Pascal Fontaine, editor, *CADE-27*, volume 11716 of *LNCS*, pages 233–249. Springer, 2019.
- 19 Asta Halkj er From, Patrick Blackburn, and J rgen Villadsen. Formalizing a Seligman-style tableau system for hybrid logic (short paper). In Nicolas Peltier and Viorica Sofronie-Stokkermans, editors, *IJCAR 2020, Part I*, volume 12166 of *LNCS*, pages 474–481. Springer, 2020.

- 20 Asta Halkjær From, Anders Schlichtkrull, and Jørgen Villadsen. A sequent calculus for first-order logic formalized in Isabelle/HOL. *J. Log. Comput.*, 33(4):818–836, 2023.
- 21 John Harrison. Formalizing basic first order model theory. In Jim Grundy and Malcolm C. Newey, editors, *TPHOLS '98*, volume 1479 of *LNCS*, pages 153–170. Springer, 1998.
- 22 Laura Kovács and Andrei Voronkov. First-order theorem proving and Vampire. In Natasha Sharygina and Helmut Veith, editors, *CAV 2013*, volume 8044 of *LNCS*, pages 1–35. Springer, 2013.
- 23 Alexander Lochmann, Bertram Felgenhauer, Christian Sternagel, René Thiemann, and Thomas Sternagel. Regular tree relations. *Archive of Formal Proofs*, 2021. URL: https://isa-afp.org/entries/Regular_Tree_Relations.html.
- 24 Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
- 25 Lawrence C. Paulson. A machine-assisted proof of Gödel’s incompleteness theorems for the theory of hereditarily finite sets. *Rev. Symb. Log.*, 7(3):484–498, 2014.
- 26 Lawrence C. Paulson. A mechanised proof of Gödel’s incompleteness theorems using Nominal Isabelle. *J. Autom. Reason.*, 55(1):1–37, 2015. doi:10.1007/S10817-015-9322-8.
- 27 Nicolas Peltier. A variant of the superposition calculus. *Archive of Formal Proofs*, 2016. URL: <https://www.isa-afp.org/entries/SuperCalc.html>.
- 28 Henrik Persson. Constructive completeness of intuitionistic predicate logic: A formalisation in type theory. Licentiate thesis, Chalmers tekniska högskola and Göteborgs universitet, 1996.
- 29 John Alan Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, 1965.
- 30 Anders Schlichtkrull, Jasmin Blanchette, Dmitriy Traytel, and Uwe Waldmann. Formalizing Bachmair and Ganzinger’s ordered resolution prover. *J. Autom. Reason.*, 64(7):1169–1195, 2020.
- 31 Anders Schlichtkrull, Jasmin Christian Blanchette, and Dmitriy Traytel. A verified prover based on ordered resolution. In Assia Mahboubi and Magnus O. Myreen, editors, *CPP 2019*, pages 152–165. ACM, 2019.
- 32 Anders Schlichtkrull, Jasmin Christian Blanchette, Dmitriy Traytel, and Uwe Waldmann. Formalization of Bachmair and Ganzinger’s ordered resolution prover. *Archive of Formal Proofs*, 2018. URL: https://isa-afp.org/entries/Ordered_Resolution_Prover.html.
- 33 Stephan Schulz, Simon Cruanes, and Petar Vukmirović. Faster, higher, stronger: E 2.3. In Pascal Fontaine, editor, *CADE-27*, volume 11716 of *LNCS*, pages 495–507. Springer, 2019.
- 34 Natarajan Shankar. *Metamathematics, Machines, and Gödel’s Proof*, volume 38 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1994.
- 35 Christian Sternagel and René Thiemann. Abstract rewriting. *Archive of Formal Proofs*, 2010. URL: <https://isa-afp.org/entries/Abstract-Rewriting.html>.
- 36 Christian Sternagel and René Thiemann. First-order terms. *Archive of Formal Proofs*, 2018. URL: https://isa-afp.org/entries/First_Order_Terms.html.
- 37 Christian Sternagel and René Thiemann. A formalization of Knuth–Bendix orders. *Archive of Formal Proofs*, 2020. URL: https://isa-afp.org/entries/Knuth_Bendix_Order.html.
- 38 Lukas Stevens and Tobias Nipkow. A verified decision procedure for orders in Isabelle/HOL. In Zhe Hou and Vijay Ganesh, editors, *ATVA 2021*, volume 12971 of *LNCS*, pages 127–143. Springer, 2021.
- 39 René Thiemann and Christian Sternagel. Certification of termination proofs using CeTA. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *TPHOLS 2009*, volume 5674 of *LNCS*, pages 452–468. Springer, 2009.
- 40 Sophie Tournet. A comprehensive framework for saturation theorem proving. *Archive of Formal Proofs*, 2020. URL: https://www.isa-afp.org/entries/Saturation_Framework.html.
- 41 Sophie Tournet. The spawns of the saturation framework. In Laura Kovács and Michael Rawson, editors, *7th and 8th Vampire Workshop*, volume 99 of *EPiC Series in Computing*, pages 1–6. EasyChair, 2024.

12:20 A Modular Formalization of Superposition in Isabelle/HOL

- 42 Sophie Tourret and Jasmin Blanchette. A modular Isabelle framework for verifying saturation provers. In Cătălin Hrițcu and Andrei Popescu, editors, *CPP 2021*, pages 224–237. ACM, 2021.
- 43 Uwe Waldmann. A modular completeness proof for the superposition calculus, 2024. URL: https://nekoka-project.github.io/pubs/isasup_blueprint.pdf.
- 44 Uwe Waldmann, Sophie Tourret, Simon Robillard, and Jasmin Blanchette. A comprehensive framework for saturation theorem proving. *J. Autom. Reason.*, 66(4):499–539, 2022.
- 45 Christoph Weidenbach, Dilyana Dimova, Arnaud Fietzke, Rohit Kumar, Martin Suda, and Patrick Wischnewski. SPASS version 3.5. In Renate A. Schmidt, editor, *CADE-22*, volume 5663 of *LNCS*, pages 140–145. Springer, 2009.

Completeness of Asynchronous Session Tree Subtyping in Coq

Burak Ekici   

Department of Computer Science, University of Oxford, UK

Nobuko Yoshida   

Department of Computer Science, University of Oxford, UK

Abstract

Multiparty session types (MPST) serve as a foundational framework for formally specifying and verifying message passing protocols. *Asynchronous subtyping* in MPST allows for typing optimised programs preserving type safety and deadlock freedom under asynchronous interactions where the message order is preserved and sending is non-blocking. The optimisation is obtained by message reordering, which allows for sending messages earlier or receiving them later. Sound subtyping algorithms have been extensively studied and implemented as part of various programming languages and tools including C, Rust and C-MPI. However, formalising all such permutations under sequencing, selection, branching and recursion in session types is an intricate task. Additionally, checking asynchronous subtyping has been proven to be undecidable.

This paper introduces the first formalisation of asynchronous subtyping in MPST within the Coq proof assistant. We first decompose session types into *session trees* that do not involve branching and selection, and then establish a coinductive refinement relation over them to govern subtyping. To showcase our formalisation, we prove example subtyping schemas that appear in the literature, all of which cannot be verified, at the same time, by any of the existing decidable sound algorithms.

Additionally, we take the (inductive) negation of the refinement relation from a prior work by Ghilezan et al. [22] and re-implement it, significantly reducing the number of rules (from eighteen to eight). We establish the completeness of subtyping with respect to its negation in Coq, addressing the issues concerning the negation rules outlined in the previous work [22].

In the formalisation, we use the greatest fixed point of the least fixed point technique, facilitated by the `paco` library, to define coinductive predicates. We employ parametrised coinduction to prove their properties. The formalisation consists of roughly 10K lines of Coq code, accessible at: <https://github.com/ekiciburak/sessionTreeST/tree/itp2024>.

2012 ACM Subject Classification Computing methodologies → Concurrent computing methodologies; Theory of computation → Type theory; Theory of computation → Logic and verification; Theory of computation → Proof theory

Keywords and phrases asynchronous multiparty session types, session trees, subtyping, Coq

Digital Object Identifier 10.4230/LIPIcs.ITP.2024.13

Supplementary Material

Software (Source Code): <https://github.com/ekiciburak/sessionTreeST/tree/itp2024> [17]
archived at `swh:1:dir:33823a0054801bcf4ea95f2dffe733579cbd53c8`

Funding This work is partially supported by EPSRC EP/T006544/2, EP/K011715/1, EP/K034413/1, EP/L00058X/1, EP/N027833/2, EP/N028201/1, EP/T014709/2, EP/V000462/1, EP/X015955/1, NCSS/EPSRC VeTSS, and Horizon EU TaRDIS 101093006.

Acknowledgements We would like to thank Dawit Tirore, Marco Giunti and Mukesh Tiwari for their feedback on the previous versions of this paper; Jovanka Vanja Pantovic and Alceste Scalas for a comprehensive discussion on the negation of the refinement relation. We also thank anonymous referees for their constructive input.



© Burak Ekici and Nobuko Yoshida;

licensed under Creative Commons License CC-BY 4.0

15th International Conference on Interactive Theorem Proving (ITP 2024).

Editors: Yves Bertot, Temur Kutsia, and Michael Norrish; Article No. 13; pp. 13:1–13:20

Leibniz International Proceedings in Informatics



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

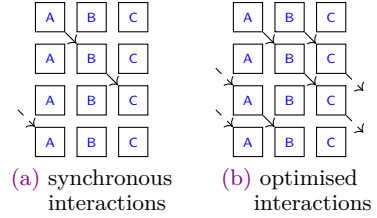
1 Introduction

Software systems often consist of concurrent and distributed components that interact through message-passing based on predefined communication protocols. Ensuring that each component adheres to the specified protocol is crucial to prevent runtime failures like communication errors and deadlocks. Session types have emerged as a successful solution to this challenge [27, 36], originally devised to two-party protocols like client-server interactions and later expanded to handle multiparty protocols as well [21, 41]. Session types offer a type-based approach to statically validate if a process conforms to a specified protocol.

A crucial challenge in employing session types lies in determining whether it is feasible to replace a part of the protocol \mathbb{T} with another \mathbb{T}' without violating safety and deadlock-freedom. This concept is referred to as session *subtyping* [18, 16], denoted by $\mathbb{T}' \leq \mathbb{T}$, when \mathbb{T}' is a subtype of \mathbb{T} .

It becomes even more challenging to formalise the precise subtyping in *asynchronous* interactions where the send operation is *non-blocking*. The asynchronous nature permits message reordering, facilitating the sending of messages earlier or delaying their reception and opening up the possibility for *protocol optimisations*. To exemplify this, we take the *ring-choice* protocol in [13], which orchestrates three participants **A**, **B** and **C**:

1. **A** sends an integer n to **B** with the label *add*.
2. **B** sends an integer m to **C**, labelled either *add* or *sub*.
 - a. If **C** receives the integer m labelled *add*, it sends an integer $m + k$ back to **A** with the *add* label, and the protocol restarts from step 1.
 - b. If **C** receives the integer m labelled *sub*, it sends an integer $m - k$ to **A** with the *sub* label, and the protocol restarts from step 1.



Source: [13]

Certainly, during *synchronous* interactions (a), no data flow would occur from **B** to **C** or from **C** to **A** before **B** receives data from **A**. However, under *asynchronous* interactions (b), assuming that each participant begins with its own initial value, **B** can *concurrently* send data (with different labels) to **C** *before* receiving data from **A**, letting **C** to start the next iteration by sending data to **A**.

The synchronous interactions from **B**'s local viewpoint could be represented by a *session type* \mathbb{T}_B , which can then be optimised into the type $\mathbb{T}_B^{\text{opt}}$ under asynchronous interactions as specified in Figure 1. The notation “!” is read as “send to” while “?” denotes “receive from”

$$\mathbb{T}_B = \mu t. A?add(i32). \oplus C! \begin{cases} add(i32).t \\ sub(i32).t \end{cases} \quad \mathbb{T}_B^{\text{opt}} = \mu t. \oplus C! \begin{cases} add(i32).A?add(i32).t \\ sub(i32).A?add(i32).t \end{cases}$$

■ **Figure 1** Local type \mathbb{T}_B and its optimised local type $\mathbb{T}_B^{\text{opt}}$ (view) of **B**.

actions, and “i32” is the integer sort of payloads. The *selection type* \oplus denotes the internal choice of actions (label, payload sort, continuation triples) directed towards a particular participant. Dually, the *branching type* $\&\mathcal{L}$ is the external choice of actions from a participant. The symbol μ denotes the recursion binder.

The optimisation pictured in Figure 1 is handled simply by reordering “send to **C**” and “receive from **A**” actions. The type of optimised interactions $\mathbb{T}_B^{\text{opt}}$ is said to be an *asynchronous subtype* [21, 22] of the type \mathbb{T}_B , and can safely replace it within the protocol while maintaining deadlock-freedom.

► **Note 1.** Throughout the paper, we hyperlink the related Coq sources to the symbol \clubsuit .

We provide the first Coq [38] library (<https://github.com/ekiciburak/sessionTreeST/tree/itp2024>) that handles the internal dynamics of asynchronous subtyping for MPST [22] and proves the optimisation summarised in Figure 1 and four more examples from the literature: Examples 3.17 3.19 and 4.14 in [22] ♣. Notice that no decidable sound subtyping algorithm in the literature [14, 5, 10, 2] can verify examples all together (see § 6).

We then prove a completeness theorem of subtyping with respect to its negation. The Coq proof of completeness involves reorganising the subtyping relation by reformulating the underlying refinement relation and its negation, proposed by Ghilezan et al. [21, 22] ♣.

In the reformulation of refinement, we accommodate the possibility of including the empty prefix ε in term syntax, leading to the definition of the relation with two fewer rules than [22, Definition 3.3] (see Figure 5). This simplification facilitates the proof of an inversion lemma, as elaborated in Remark 6 and Lemma 7.

Regarding the reformulation of the refinement negation, we reduce the number of rules from eighteen in [22, Fig. 6] to eight, thereby rendering the remaining ten rules provable. This is done by introducing a new sort of term prefixing ($\mathcal{C}^{\mathbf{p}}$ – Lemma 12) and using it to modify some of the original rules in order to adopt a better structural shape of rules and become more readily applicable within proofs. Further details are covered in Lemmata 17, 14, 18 and 20, and in Remark 19.

The accompanying Coq library can be used to certify additional asynchronous protocol optimisations in MPST. This entails defining both the original and optimised protocols, then applying either of the two main refinement rules (see Figure 5) to show that the latter is a subtype of the former. In addition, provided that the subtyping is obtained by the use of coinductive structures in Coq, applications dealing with infinite trees could also leverage the structures and lemmata present in the library.

2 Session Trees and Subtyping

The subtyping of session types [16, 18] plays a crucial role in process calculi, as a process that instantiates a session type \mathbb{T} can securely substitute another process inhabiting a supertype \mathbb{T}' of \mathbb{T} . Such substitution contributes to the development of more optimised protocols [21, 22].

Each *closed* asynchronous session type \mathbb{T} is associated with a corresponding session tree $\mathbb{T} = \mathcal{T}(\mathbb{T})$. Refer to [20, Def. A.14] for the definition of the translation function $\mathcal{T}: \mathbb{T} \rightarrow \mathbb{T}$. Therefore, the definition of a subtyping relation could be captured by the use of session trees. With this perspective, in this work we introduce a Coq library that implements asynchronous session trees together with various property proofs.

A session tree is *coinductively* defined with the following syntax that reflects in Coq ♣ in the way listed alongside:

$$\begin{array}{l} \mathbb{T} ::= \\ | \text{end} \\ | \&_{i \in I} \mathbf{p}^? \ell_i(\mathbb{S}_i). \mathbb{T}_i \\ | \bigoplus_{i \in I} \mathbf{p}! \ell_i(\mathbb{S}_i). \mathbb{T}_i \end{array}$$

```
CoInductive st: Type ≙
| st_end      : st
| st_receive: participant → list (label*sort*st) → st
| st_send    : participant → list (label*sort*st) → st.
Notation "p ??? l" ≙ (st_receive p l).
Notation "p ! l" ≙ (st_send p l).
```

The constructor $\&_{i \in I} \mathbf{p}^? \ell_i(\mathbb{S}_i). \mathbb{T}_i$ denotes *branching* (or *external choice*) interactions and represents a set of messages towards participant \mathbf{p} with labels ℓ_i , payload sorts \mathbb{S}_i and continuations \mathbb{T}_i . While $\bigoplus_{i \in I} \mathbf{p}! \ell_i(\mathbb{S}_i). \mathbb{T}_i$ stands for *selection* (or *internal choice*) and specifies a set of messages from \mathbf{p} with labels ℓ_i , payload sorts \mathbb{S}_i and continuations \mathbb{T}_i (for some $i \in I$); the constructor **end** signals termination of interactions.

13:4 Completeness of Asynchronous Session Tree Subtyping in Coq

In both the code alongside and the rest of the paper, we use the notation “?” and `st_receive` constructor interchangeably, as well as the notation ‘!’ and `st_send` constructor. We exclude the symbols $\&$ and \oplus but maintain their functionality using Coq lists. That is, labels, sorts and continuations for selections and branchings are represented in Coq by lists of label-sort-continuation triples. See the constructors `st_receive` and `st_send` in the above code snippet.

The objective in checking “whether a session tree T qualifies a subtype (subtree) of another tree T' ($T \leq T'$)” is twofold:

1. the *decomposition* of both trees into sets of *single-input-single-output* (SISO) trees, and
2. checking whether it is possible to find SISO trees W , from the decomposition of considered subtree, and W' , from the decomposition of considered supertree, such that W is a *refinement* of W' . That is, there exist certain ways to reorder the actions in W so that it matches the structure of actions in W' . See Definition 5 for further details on refinement.

2.1 SI, SO and SISO Trees

In [22], the decomposition of a given session tree T into a set of SISO trees is not accomplished all at once; instead, it involves intermediate steps. Initially, T is partitioned into a set of trees where each tree is characterised by singleton choices in their selections (referred to as *single-output* (SO) trees). Subsequently, for each individual SO tree, a further set of trees is formed where the members exhibit singleton branchings (referred to as *single-input* (SI) trees). Therefore, consecutively applying SO and SI decompositions (in any order) to a session tree eventually yields in a set of SISO trees.

In what follows, we *coinductively* present SO (denoted U), SI (denoted V) and SISO (denoted W) trees. In SO trees, there is a list of branchings but a single selection while SI trees contain a list of selections with a single branching

$$U ::= \text{end} \mid \&_{i \in I} p? \ell_i(S_i).U_i \mid p! \ell(S).U \quad V ::= \text{end} \mid p? \ell(S).V \mid \oplus_{i \in I} p! \ell_i(S_i).V_i$$

and SISO trees are made of single branching and single selection \mathfrak{P} :

$$W ::= \begin{array}{l} \text{end} \\ p? \ell(S).W \\ p! \ell(S).W \end{array}$$

```
Inductive singletonI (R: st → Prop): st → Prop ≙
| ends : singletonI R st_end
| sends: ∀ p l s w, R w → singletonI R (st_send p [(l,s,w)])
| recvs: ∀ p l s w, R w → singletonI R (st_receive p [(l,s,w)]).
Definition singleton s ≙ paco1 (singletonI) bot1 s.
Class siso: Type ≙ mk_siso { und: st; sprop: singleton und }.
```

Formalisation of SISO trees in Coq initiates with the declaration of a `Prop` valued inductive predicate `singletonI` which serves for verifying “whether the selections and branchings within a given session tree are singletons”. We then leverage the “greatest fixed point of the least fixed point” technique facilitated by the `paco` library [29], and generate the type `singleton` as the greatest fixed point of `singletonI`; so that the latter is applicable to infinite session trees. We then formulate SISO trees as a sigma type of a session tree `und` such that `und` respects `singleton`.

This technique has been employed by Zakowski et al. [43] to define weak bisimilarity on streams, and Tirore et al. [40] to define (sound and complete) projection of global session types onto local types.

► Remark 2. The use of `paco` library is beneficial – in many constructions presented through our the paper – since it allows for coinductive reasoning *parametrised* by “accumulated knowledge” so that proof goals could be closed upon encountering something that is already in

the knowledge set through out coinductive folding steps. Furthermore, `paco` utilises semantic guardedness rather than relying on syntactic guard checks, which can be problematic and compromised even through straightforward setoid rewrites.

We bypass the intermediate SO and SI decompositions and directly build a coinductive relation that inhabits SISO tree and session tree pairs such that former is obtained by decomposing the latter. This approach is slightly different from the one outlined in § 3.4 of [22], but it better aligns with Coq formalisation.

► **Definition 3.** \clubsuit *The SISO decomposition of a session tree is governed by the coinductive relation \llcorner with the following rules:*

$$\frac{\forall i \in I \quad \exists k \in I \quad \ell = \ell_k \quad W \llcorner T_k}{p?l(S).W \llcorner \&\mathcal{I}_{i \in I} p?l_i(S_i).T_i} \text{ [SISO-RCV]} \quad \frac{\forall i \in I \quad \exists k \in I \quad \ell = \ell_k \quad W \llcorner T_k}{p!l(S).W \llcorner \bigoplus_{i \in I} p!l_i(S_i).T_i} \text{ [SISO-SND]}$$

$$\frac{}{\text{end } \llcorner \text{ end}} \text{ [SISO-END]}$$

for some finite set of indices I .

The relation \llcorner provided in Definition 3 is coinductively implemented in Coq under the name `st2sisoC`, as shown below. This implementation operates at the level of session trees instead of SISO trees, to avoid the need for singleton checks at each step of rule application. However, we ensure that the relation is instantiated with the underlying `und` of a `siso` tree whenever it is called. Refer to the formal subtyping definition, `subtype`, outlined in Definition 10, for instance. We maintain this methodology until § 4.2, where we establish the negation of the refinement relation, `nRefinement`, directly over `siso` trees.

```
Inductive st2siso (R: st → st → Prop): st → st → Prop ≙
| st2siso_end: st2siso R st_end st_end
| st2siso_rcv: ∀ l s x xs p, R x (pathsel l xs) → st2siso R (p ?' [(l,s,x)]) (st_receive p xs)
| st2siso_snd: ∀ l s x xs p, R x (pathsel l xs) → st2siso R (p ! [(l,s,x)]) (st_send p xs) .
Definition st2sisoC s1 s2 ≙ paco2 (st2siso) bot2 s1 s2.
```

The function `pathsel` selects the path among the list of selections and branchings that matches the label of the current SISO action.

```
Fixpoint pathsel (u: label) (l: list (label*sort*st)): st ≙
match l with
| (lbl,s,x)::xs => if eqb u lbl then x else pathsel u xs
| nil => st_end
end.
```

It returns T_k under the condition $\ell = \ell_k$ within the context of the rules [SISO-RCV] and [SISO-SND].

2.2 SISO Tree Refinement

The other key component of checking whether a given session tree is a subtree (or supertree) of another is the support for action reordering. Conceptually, the subtree has the capability to “anticipate” certain input/output actions that are expected to take place in the supertree. This anticipation is captured by action reordering [22, Def. 3.2], namely executing actions earlier or later than their prescribed occurrence.

► **Definition 4.** (\clubsuit , \spadesuit). To elucidate action reorderings, a pair of input/output sequences are recursively defined below

$$\begin{aligned} \mathcal{A}^{(p)} &::= \varepsilon \mid q?l(S) \mid q?l(S).\mathcal{A}^{(p)} \\ \mathcal{B}^{(p)} &::= \varepsilon \mid r?l(S) \mid q!l(S) \mid r?l(S).\mathcal{B}^{(p)} \mid q!l(S).\mathcal{B}^{(p)} \quad (q \neq p) \end{aligned}$$

13:6 Completeness of Asynchronous Session Tree Subtyping in Coq

The $\mathcal{A}^{(p)}$ prefix refers to a finite sequence of actions containing all possible receives excluding those from the participant p . $\mathcal{B}^{(p)}$, on the other hand, indicates a finite sequence that involves all receives and all sends but not those towards participant p .

► **Definition 5.** *The refinement relation \lesssim over SISO trees is coinductively defined with:*

$$\frac{S' \leq: S \quad W \lesssim \mathcal{A}^{(p)}.W' \quad \text{act}(W) = \text{act}(\mathcal{A}^{(p)}.W')}{\frac{p?l(S).W \lesssim \mathcal{A}^{(p)}.p?l(S').W'}{S \leq: S' \quad W \lesssim \mathcal{B}^{(p)}.W' \quad \text{act}(W) = \text{act}(\mathcal{B}^{(p)}.W')}} \text{[REF-A]} \quad \frac{\text{end} \lesssim \text{end}}{\text{end} \leq: \text{end}} \text{[REF-END]}$$

The symbol $\leq:$ denotes the least reflexive relation over payload sorts (i.e., $\text{nat} \leq: \text{int}$) while the function act coinductively accumulates the actions, participant-*dir* pairs where $\text{dir} \in \{!, ?\}$, of a given SISO tree into a stream (or a colist/coseq). Action equality checks serve the purpose of ensuring that rule applications neither introduce nor remove actions.

The rule [REF-B] in general captures the reordering backed by $\mathcal{B}^{(p)}$ prefixes. It allows for the reordering, a finite number of times, of an output directed towards a participant p with any input and output combinations, excluding other outputs directed towards the participant p . While the rule [REF-A] anticipates the reordering of an input from a participant p with any input combination but not those from p .

► **Remark 6.** As opposed to the original definition of refinement relation given in [22, Def. 3.2], we allow prefixes $\mathcal{A}^{(p)}$ and $\mathcal{B}^{(p)}$ to include the empty prefix ε . This deviation indeed introduces an important flexibility in the framework. By permitting this, we are essentially acknowledging the possibility of contexts without any actions, which can be crucial for certain proofs and reasoning processes. It particularly allows for the proof of an inversion lemma, which asserts that *SISO trees with action dis-equality cannot refine each other*. This is one of the key results of our Coq formalisation.

► **Lemma 7.** *$\forall W W', \text{act}(W) \neq \text{act}(W') \implies \neg(W \lesssim W')$.*

This lemma is a significant result, as it establishes a fundamental property regarding the relationship between terms with action mismatch. Note that the action the dis-equality definition here is obtained by negating the statement in Definition 15.

Below is a representation of the refinement relation \lesssim in a Coq implementation where the rule [REF-B] is referred to as `ref_b` while [REF-A] is named `ref_a`.

```

Inductive dir: Type  $\triangleq$  rcv: dir | snd: dir.
CoFixpoint act (t: st): coseq (participant * dir)  $\triangleq$ 
  match t with
  | st_receive p [(l,s,t')]  $\Rightarrow$  cocons (p, rcv) (act t')
  | st_send p [(l,s,t')]  $\Rightarrow$  cocons (p, snd) (act t')
  | _  $\Rightarrow$  conil
  end.
Inductive refinementR (R: st  $\rightarrow$  st  $\rightarrow$  Prop): st  $\rightarrow$  st  $\rightarrow$  Prop  $\triangleq$ 
  | ref_a :  $\forall w w' p l s s' a n, \text{subsort } s s' \rightarrow \text{seq } w (\text{merge\_ap\_contn } p a w' n) \rightarrow$ 
    (  $\exists L1, \exists L2, \text{coseqInLC } (\text{act } w) L1 \wedge \text{coseqInLC } (\text{act } (\text{merge\_ap\_contn } p a w' n)) L2 \wedge$ 
      coseqInR L1 (act w)  $\wedge$  coseqInR L2 (act (merge_ap_contn p a w' n))  $\wedge$ 
      ( $\forall x, \text{List.In } x L1 \leftrightarrow \text{List.In } x L2$ ) )  $\rightarrow$ 
      refinementR R (st_receive p [(l,s,w)]) (merge_ap_contn p a (st_receive p [(l,s',w')]) n)
  | ref_b :  $\forall w w' p l s s' b n, \text{subsort } s s' \rightarrow \text{seq } w (\text{merge\_bp\_contn } p b w' n) \rightarrow$ 
    (  $\exists L1, \exists L2, \text{coseqInLC } (\text{act } w) L1 \wedge \text{coseqInLC } (\text{act } (\text{merge\_bp\_contn } p b w' n)) L2 \wedge$ 
      coseqInR L1 (act w)  $\wedge$  coseqInR L2 (act (merge_bp_contn p b w' n))  $\wedge$ 
      ( $\forall x, \text{List.In } x L1 \leftrightarrow \text{List.In } x L2$ ) )  $\rightarrow$ 
      refinementR R (st_send p [(l,s,w)]) (merge_bp_contn p b (st_send p [(l,s',w')]) n)
  | ref_end: refinementR seq st_end st_end.
Definition refinement: st  $\rightarrow$  st  $\rightarrow$  Prop  $\triangleq$  fun s1 s2  $\Rightarrow$  paco2 refinementR bot2 s1 s2.

```

The function `merge_bp_contn` takes a participant p , an instance b of $\mathcal{B}^{(p)}$, a natural number n and a session tree w . It repeats the action b , n times, and prefixes it to the tree w . There are analogous constructs for $\mathcal{A}^{(p)}$ type of prefixes which we omit elucidating here. The code blocks under the existential quantifiers \exists validate action equalities according to Definition 8.

2.3 Action Equalities

One key point in the context of the refinement relation is to decide the equalities over streams of actions modulo action reordering. There, the focus lies not on assessing structural equality between streams, but rather on discerning the similarity of their constituent elements. A potential strategy to achieve this is having a coinductive definition of stream membership, and checking if a pair of streams have matching members \spadesuit .

```

Inductive coseqInC {A: Type} (R: A → coseq A → Prop): A → coseq A → Prop  $\triangleq$ 
| CoInSplit1A x xs {ys}: xs = cocons x ys → coseqInC R x xs
| CoInSplit2A x xs y ys: xs = cocons y ys → x ≠ y → R x ys → coseqInC R x xs.
Definition coseqCoIn {A}  $\triangleq$  paco2 (@coseqInC A) bot2.

```

This coinductive approach turns out to be unsound as it allows for proving the existence of a ‘b’ within the stream of ‘a’s where $a \neq b$.

```

CoFixpoint W {A: Type} (a: A): coseq A  $\triangleq$  cocons a (W a).
Lemma unsound_coseqCoIn:  $\forall$  A (a b: A), a ≠ b → coseqCoIn b (W a).

```

We proceed by assuming that any stream of actions adheres to a reasonable notion of **finiteness**, meaning it comprises only a finite set of distinct actions. This assumption naturally aligns with the framework of multiparty session types. Even in scenarios where sessions involve an infinite number of interactions, these interactions must occur among a finite number of participants [21, 22], leading to finitely many distinct actions. Consequently, it becomes feasible to state that a pair of streams share identical members if and only if the list of (distinct) actions is contained within the stream of actions. We do not explicitly state this as an axiom in Coq, rather massage it into the action equality check formulated in Definition 8.

► **Definition 8.** For a pair of SISO trees W and W' , we define action equality as follows:

$$\exists l_1 l_2, \quad l_1 \in_I \text{act}(W) \wedge \text{act}(W) \in_C l_1 \wedge l_2 \in_I \text{act}(W') \wedge \text{act}(W') \in_C l_2 \wedge (\forall x, \text{mem } x l_1 \iff \text{mem } x l_2)$$

where the relation \in_I inductively traverses a given action list and checks if every list member is in the stream, while \in_C coinductively folds a provided stream of actions, and checks if every stream member is in the list. These relations are formally defined employing the following constructors:

$$\frac{}{\text{nil} \in_I w} \text{[I-NIL]} \quad \frac{x \in w \quad xs \in_I w}{(x :: xs) \in_I w} \text{[I-CONS]} \quad \frac{}{\text{conil} \in_C l} \text{[C-NIL]} \quad \frac{\text{mem } x l \quad xs \in_C l}{(\text{cocons } x xs) \in_C l} \text{[C-CONS]}$$

The symbol \in represents the inductive stream membership check, associated with the predicate `coseqIn` in the following code snippet, whereas `mem` denotes the typical list membership check.

► **Lemma 9.** \spadesuit Given $W := p!l_1(S_1).p?l_2(S_2).q!l_3(S_3).W$ and $l := p? :: p! :: q! :: \text{nil}$, we have (1) $\text{act}(W) \in_C l$ and (2) $l \in_I \text{act}(W)$.

Proof. To close the first item, we apply the constructor `[C-CONS]` three times making sure that $p!$, $p?$ and $q!$ are in l , and then employ the coinduction hypothesis. The proof of the second item proceeds by applying the constructor `[I-CONS]` three times, ensuring that $p?$, $p!$, and $q!$ are in $\text{act}(W)$. Finally, one more application of `[I-NIL]` suffices to demonstrate the goal. ◀

13:8 Completeness of Asynchronous Session Tree Subtyping in Coq

We formalise the relation \in_C (resp., \in_I) in Coq, denoted as `coseqInLC` (resp., `coseqInR`) \clubsuit .

```

Inductive coseqIn: (participant * dir) → coseq (participant * dir) → Prop ≜
| CoInSplit1 x xs y ys: xs = cocons y ys → x = y → coseqIn x xs
| CoInSplit2 x xs y ys: xs = cocons y ys → x ≠ y → coseqIn x ys → coseqIn x xs.
Inductive coseqInL (R: coseq (participant * dir) → list (participant * dir) → Prop):
coseq (participant * dir) → list (participant * dir) → Prop ≜
| c_nil : ∀ ys, coseqInL R conil ys
| c_incl: ∀ x xs ys, List.In x ys → R xs ys → coseqInL R (cocons x xs) ys.
Definition coseqInLC ≜ fun s1 s2 => paco2 (coseqInL) bot2 s1 s2.
Inductive coseqInR: list (participant * dir) → coseq (participant * dir) → Prop ≜
| i_nil : ∀ ys, coseqInR nil ys
| i_incl: ∀ x xs ys, coseqIn x ys → coseqInR xs ys → coseqInR (x::xs) ys.

```

This strategy remains effective for proving subtyping examples discussed in § 3.1 as it allows us to store actions of specifically given trees into finite lists and perform action equality checks via list membership comparisons. However, it becomes cumbersome when aiming to prove completeness of subtyping. This is because it is not useful for proving general properties about tree shapes with prefixes; see § 4.1, especially Lemmata 12 and 13.

3 Asynchronous Subtyping

► **Definition 10.** \clubsuit *The asynchronous subtyping relation \leq over session trees is defined as:*

$$\frac{\exists \{(W_i, W'_i) \mid i \in I\} \quad W_i \rightsquigarrow T \quad W'_i \rightsquigarrow T' \quad W_i \lesssim W'_i}{T \leq T'}$$

for some finite set of indices I .

We revisit the original subtyping definition in [22, Def. 3.13] such that it relies on the direct decomposition into SISO trees as in Definition 3. The intuitive idea is then to plug in a list of SISO tree pairs (W_i, W'_i) , where each W_i is part of the SISO decomposition of T , similarly each W'_i is part of the decomposition of T' , and check whether W_i refines W'_i .

```

Definition subtype (T T': st): Prop ≜ ∃ (l: list (siso*siso)), decomposeL l T T' ∧ listSisoPRef l.

```

The function `decomposeL` verifies if the first projection of each pair in the list l is a SISO tree taken from decomposing T , and if the second projections are from T' . While, the function `listSisoPRef` is used to conduct refinement checks employing the `refinement` relation.

3.1 Subtyping Example

An instance of optimisation managed by asynchronous subtyping arises in the protocol for distributed batch processing (Example 4.14 in [22]), where a particular segment of the protocol is replaced with another. We have this subtyping proof formalised in Coq, not at the level of session types but their induced session trees \clubsuit . In consideration of space limitations, we omit this proof and instead introduce another optimisation case addressed by subtyping (Example 3.19 in [22]), which is more involved as it contains coinductive but non-cyclic derivations. Consider the following session types:

$$\mathbb{T} = \mu t_1. \& p^? \begin{cases} \ell_1(S).p!\ell_3(S).p!\ell_3(S).p!\ell_3(S).t_1 \\ \ell_2(S).\mu t_2.p!\ell_3(S).t_2 \end{cases} \quad \mathbb{T}' = \mu t_1. \& p^? \begin{cases} \ell_1(S).p!\ell_3(S).t_1 \\ \ell_2(S).\mu t_2.p!\ell_3(S).t_2 \end{cases}$$

■ **Figure 2** Example session types with \mathbb{T}' is a subtype of \mathbb{T} from [22, Example 3.19].

We translate the types \mathbb{T} and \mathbb{T}' into respective session trees \mathbb{TB} and \mathbb{TB}' manually, and then develop them in Coq. The operational aspect of the recursion binder μ is addressed by Coq's `CoFixpoint` vernacular as session trees are coinductively defined.

```

CoFixpoint TS : st  $\triangleq$  "p"!["13",I,TS)].
CoFixpoint TB : st  $\triangleq$  "p"?["11",I,"p"!["13",I,"p"!["13",I,"p"!["13",I,TB]]]); ("12",I,TS)].
CoFixpoint TB' : st  $\triangleq$  "p"?["11",I,"p"!["13",I,TB']]); ("12",I,TS)].

```

Before proceeding with the proof steps, we introduce some tree terms and prefixes that will be used later within the proof.

```

CoFixpoint WB : st  $\triangleq$  "p"?["11",I,"p"!["13",I,"p"!["13",I,"p"!["13",I,WB]]]]].
CoFixpoint WB' : st  $\triangleq$  "p"?["11",I,"p"!["13",I,WB']]].
Definition pi1 : Dp  $\triangleq$  "p"? "11" I ("p!" "13" I).
Definition pi3 : Dp  $\triangleq$  "p"? "11" I ("p!" "13" I ("p!" "13" I ("p!" "13" I))).
CoFixpoint WD : st  $\triangleq$  "p"!["13",I,WD]].
Definition WA : st  $\triangleq$  "p"?["12",I,WD]].

```

The type Dp \mathfrak{P} is inhabiting SISO style (without branching and selection) term prefixes. To prove that $\mathbb{TB} \leq \mathbb{TB}'$ holds, we are supposed to

- (1) show that $(\mathbb{WB}, \mathbb{TB})$, $(\mathbb{WB}', \mathbb{TB}')$, $(\mathbb{WA}, \mathbb{TB})$ and $(\mathbb{WA}, \mathbb{TB}')$ are in `st2sisoC`
- (2) and demonstrate that $\mathbb{WB} \lesssim \mathbb{WB}'$ with $(\text{pi3})^n \cdot \mathbb{WA} \lesssim (\text{pi1})^n \cdot \mathbb{WA}$ for all naturals n . The infix function $\mathfrak{t} \cdot \mathbb{W}$ glues a prefix term \mathfrak{t} (of type Dp) to a SISO tree \mathbb{W} , and the superscript n denotes the repetition of the prefix n times before glueing.

The complication in refinement stated in item (2) above is due to the complex co-recursive structure of terms \mathbb{TB} and \mathbb{TB}' . Intuitively, the former case covers the refinement for the outer recursive structure while the latter is supposed to deal with the inner one.

Proof. \mathfrak{P} We skip the last two cases of the item (1) and the last case of the item (2) due to space constraints, and begin by proving that pairs $(\mathbb{WB}, \mathbb{TB})$ and $(\mathbb{WB}', \mathbb{TB}')$ are in `st2sisoC`. To address the former case, we apply the rule `st2siso_rcv` once and `st2siso_snd` three times. We then employ the coinductive hypothesis that saves the initial proof state. The proof of the second case shares commonalities. It can be effectively handled by first applying the `st2siso_rcv` rule followed by `st2siso_snd`, and then invoking the coinductive hypothesis.

Proving that $\mathbb{WB} \lesssim \mathbb{WB}'$ holds however presents a more intriguing scenario. For that, a pen-and-paper proof is structured in Figure 3 (read: bottom left \rightarrow top left $\xrightarrow{\text{generalised by}}$ bottom right \rightarrow top right). Steps on the left are straightforward refinement rule applications,

$$\begin{array}{c}
\frac{\mathbb{WB} \lesssim \mathfrak{p}^{\ell_1}(\mathbb{S}).\mathfrak{p}^{\ell_1}(\mathbb{S}).\mathbb{WB}'}{\frac{\frac{\frac{\mathfrak{p}^{\ell_3}(\mathbb{S}).\mathbb{WB} \lesssim \mathfrak{p}^{\ell_1}(\mathbb{S}).\mathbb{WB}'}{[\text{REF-}\mathcal{B}]}]{\frac{\mathfrak{p}^{\ell_3}(\mathbb{S})^2.\mathbb{WB} \lesssim \mathbb{WB}'}{[\text{REF-}\mathcal{B}]}]}{[\text{REF-}\mathcal{B}]}]{\frac{\mathfrak{p}^{\ell_3}(\mathbb{S})^3.\mathbb{WB} \lesssim \mathfrak{p}^{\ell_3}(\mathbb{S}).\mathbb{WB}'}{[\text{REF-}\mathcal{A}]}]}{[\text{REF-}\mathcal{A}]}]}{\mathbb{WB} \lesssim \mathbb{WB}'} \\
\end{array}
\qquad
\begin{array}{c}
\frac{\mathbb{WB} \lesssim (\mathfrak{p}^{\ell_1}(\mathbb{S}_1).\mathfrak{p}^{\ell_1}(\mathbb{S}))^{n+1}.\mathbb{WB}'}{\frac{\frac{\frac{\mathfrak{p}^{\ell_3}(\mathbb{S}).\mathbb{WB} \lesssim (\mathfrak{p}^{\ell_1}(\mathbb{S}).\mathfrak{p}^{\ell_1}(\mathbb{S}_1))^n.\mathfrak{p}^{\ell_1}(\mathbb{S}).\mathbb{WB}'}{[\text{REF-}\mathcal{B}]}]{\frac{\mathfrak{p}^{\ell_3}(\mathbb{S})^2.\mathbb{WB} \lesssim (\mathfrak{p}^{\ell_1}(\mathbb{S}).\mathfrak{p}^{\ell_1}(\mathbb{S}))^n.\mathbb{WB}'}{[\text{REF-}\mathcal{B}]}]}{[\text{REF-}\mathcal{B}]}]{\frac{\mathfrak{p}^{\ell_3}(\mathbb{S})^3.\mathbb{WB} \lesssim (\mathfrak{p}^{\ell_1}(\mathbb{S}).\mathfrak{p}^{\ell_1}(\mathbb{S}))^n.\mathfrak{p}^{\ell_3}(\mathbb{S}).\mathbb{WB}'}{[\text{REF-}\mathcal{A}]}]}{[\text{REF-}\mathcal{A}]}]}{\mathbb{WB} \lesssim (\mathfrak{p}^{\ell_1}(\mathbb{S}).\mathfrak{p}^{\ell_1}(\mathbb{S}))^n.\mathbb{WB}'} \\
\end{array}$$

■ **Figure 3** Proof steps of $\mathbb{WB} \lesssim \mathbb{WB}'$. (Source: [22, Example 3.19])

where the topmost derivation is complemented by the helper steps on the right for every natural number n . These auxiliary steps can be proven by conducting a case analysis on n , supported by a “stronger” coinduction hypothesis universally quantifying over n .

13:10 Completeness of Asynchronous Session Tree Subtyping in Coq

In a Coq implementation, however, we take a slightly different approach. We consider merging WB' with a single prefix $p?\ell_1(S)$ and ensure that this happens an even number of times. Below lemma aligns with the bottommost line of the helper steps in Figure 3 \clubsuit .

```
Lemma WBRef:  $\forall n, \text{ev } n \rightarrow \text{refinement } WB \text{ (merge\_bp\_contn "p" (bp\_receivea "p" "11" sint) } WB' \text{ n).$ 
```

The term `bp_receivea` in the lemma statement corresponds to the $r?\ell(S).\mathcal{B}^{(p)}$ constructor of $\mathcal{B}^{(p)}$, enabling the prefixing of a receive action from any participant to a given session tree. Consequently, in the statement, the right-hand side of the refinement represents a SISO tree where the action $p?$ is executed an even number (n) of times before being succeeded by actions from WB' .

To develop this lemma in Coq, we begin by storing the proof state within a coinduction hypothesis `CIH`, universally quantified over n . We then conduct a case analysis based on whether n is even. This results in two subgoals: one where $n = 0$ and another where $n \geq 0$ is an even number. The former case involves demonstrating the validity of $WB \lesssim WB'$ is omitted here due to space constraints. We proceed with the latter case, which is outlined below:

```
CIH :  $\forall n : \text{nat}, \text{ev } n \rightarrow r \text{ WB (merge\_bp\_contn "p" (bp\_receivea "p" "11" (I)) } WB' \text{ n)$ 
H   :  $\text{ev } n$ 
----- (1/1)
paco2 refinementR r WB (merge\_bp\_contn "p" (bp\_receivea "p" "11" (I)) } WB' \text{ n.+2)
```

► **Remark 11.** To center the attention on actions and continuations, we will no longer use list notation, labels, or sorts in the rest of the proof text. This is because both sides of \lesssim are made of streamline of actions (nested singleton lists), where all elements (labels and sorts) align. Just that we employ a dot to delineate prefixes from the infinite terms.

Unfolding WB and applying the rule `ref_a` transforms the goal into $p!p!p!.WB \lesssim (p?')^{n+1}.WB'$. This term corresponds to the one given in second-to-last line on the right-hand side of the proof steps in Figure 3. Note that the rule application permits the discharge of the leftmost receive prefixes on both sides.

After unfolding WB' inside the goal, it takes the form $p!p!p!.WB \lesssim (p?')^{n+2}p!.WB'$. We then apply `ref_b` with $n \triangleq n+2$, resulting in $p!p!.WB \lesssim (p?')^{n+2}.WB'$. Notice that this application effectively shifts the send action on the right to the leftmost position through reordering and cancels the leftmost send prefixes.

We keep unfolding WB' followed by the application of `ref_b` with $n \triangleq n+3$ and $n \triangleq n+4$ respectively and obtain the goal in the following shape: $WB \lesssim (p?')^{n+4}.WB'$ which could easily be closed by instantiating the coinduction hypothesis `CIH` with $n \triangleq n+4$. Note also that we separately prove the action equalities after every single application of rules `ref_a` and `ref_b` employing the idea in Definition 8. ◀

4 Subtyping Negation

The negation of refinement relation $\not\lesssim$ over SISO trees structurally displays the shape of trees which cannot refine each other. It serves as a framework for the complement of subtyping within the context of session types thus session trees. Originally comprised of eighteen inductively stated rules as outlined in [22, Fig. 6], we are able to shrink the set by eliminating ten rules, and present the new set of rules in Figure 4. Completeness with respect to refinement is elaborated in § 5. Before delving into the specifics of the new set of rules, we need to revisit the way action equalities are handled.

4.1 Action Equalities, Refinement and Subtyping Revisited

We revisit the rationale behind membership and action equality checks outlined in § 2.3, and restate refinement and subtyping relations accordingly. We establish an inductive membership relation over streams of actions which is crucial for deriving useful lemmas regarding the structure of terms containing specific actions. For instance, Lemma 12 and 13 cannot be proven unless the membership check \in is inductively defined. This is because it is impossible to infer term shapes from a coinductively defined membership relation. This can only be achieved through the induction schema for an inductively defined membership relation. These lemmata are key in proving completeness of refinement thus subtyping with respect to negations.

► **Lemma 12.** $\forall p \ W, \exists \mathcal{C}^{(p)} \ \ell \ S \ W', \ p? \in \text{act}(W) \implies W = \mathcal{C}^{(p)}.p?\ell(S).W'$.

where $\mathcal{C}^{(p)}$ is a sort of prefixing \forall

$$\mathcal{C}^{(p)} ::= \varepsilon \mid r!\ell(S) \mid q?\ell(S) \mid r!\ell(S).\mathcal{C}^{(p)} \mid q?\ell(S).\mathcal{C}^{(p)} \quad (q \neq p)$$

that allows all sends alongside all receives but not those from a particular participant p .

► **Lemma 13.** $\forall p \ W, \exists \mathcal{B}^{(p)} \ \ell \ S \ W', \ p! \in \text{act}(W) \implies W = \mathcal{B}^{(p)}.p!\ell(S).W'$.

Notice that $\mathcal{C}^{(p)}$ sort of prefixing amounts to the $\mathcal{A}^{(p)}$ sort in the absence of send actions.

► **Lemma 14.** $\forall p \ \mathcal{C}^{(p)} \ W, \ p! \notin \mathcal{C}^{(p)} \implies (\exists \mathcal{A}^{(p)}, \ \mathcal{C}^{(p)}.W = \mathcal{A}^{(p)}.W)$.

► **Definition 15.** \forall For a pair of SISO trees W and W' , we define action equality as follows:

$$\forall a, \ a \in \text{act}(W) \iff a \in \text{act}(W').$$

```
Definition act_eq (t t': st)  $\triangleq$   $\forall$  a, coseqIn a (act t)  $\leftrightarrow$  coseqIn a (act t').
Definition act_neq (t t': st)  $\triangleq$   $\exists$  a, coseqIn a (act t)  $\wedge$  (coseqIn a (act t')  $\rightarrow$  False)  $\vee$ 
coseqIn a (act t')  $\wedge$  (coseqIn a (act t)  $\rightarrow$  False).
```

We redefine the refinement relation by incorporating the action equality check described in Definition 15 \forall . This adjustment enables us to establish its completeness in regard to negations as discussed in § 4.2 and § 5.

```
Inductive refinementR2 (seq: st  $\rightarrow$  st  $\rightarrow$  Prop): st  $\rightarrow$  st  $\rightarrow$  Prop  $\triangleq$ 
| ref2_a:  $\forall$  w w' p l s s' a n,
  subsort s' s  $\rightarrow$  seq w (merge_ap_contn p a w' n)  $\rightarrow$ 
  act_eq w (merge_ap_contn p a w' n)  $\rightarrow$ 
  refinementR2 seq (st_receive p [(l,s,w)]) (merge_ap_contn p a (st_receive p [(l,s',w')]) n)
| ref2_b:  $\forall$  w w' p l s s' b n,
  subsort s s'  $\rightarrow$  seq w (merge_bp_contn p b w' n)  $\rightarrow$ 
  act_eq w (merge_bp_contn p b w' n)  $\rightarrow$ 
  refinementR2 seq (st_send p [(l,s,w)]) (merge_bp_contn p b (st_send p [(l,s',w')]) n)
| ref2_end: refinementR2 seq st_end st_end.
Definition refinement2: st  $\rightarrow$  st  $\rightarrow$  Prop  $\triangleq$  fun s1 s2  $\Rightarrow$  paco2 refinementR2 bot2 s1 s2.
Notation "x' <-< y"  $\triangleq$  (refinement2 x y) (at level 50, left associativity).
```

The subtyping relation therefore takes the following shape \forall :

```
Definition subtype2 (T T': st): Prop  $\triangleq$   $\exists$  (l: list (siso*siso)), decomposeL l T T'  $\wedge$  listSisoPRef2 l.
```

The sole difference between the functions `listSisoPRef` and `listSisoPRef2` is that the former employs the `refinement` relation while the latter makes use of the `refinement2` relation.

13:12 Completeness of Asynchronous Session Tree Subtyping in Coq

► **Remark 16.** In Coq, there is no way to bridge the gap between the action equality checks in Definitions 8 and 15. One potential avenue involves assuming that

$$\forall W \ell, \text{act}(W) \in_C \ell \implies \text{act}(W) \in_C^I \ell \quad (1)$$

holds for the inductively defined version \in_C^I of \in_C ; and deducing that the statement in Definition 8 implies that in Definition 15. However, this approach would lead to inconsistency in Coq. It is because the predicate \in_C^I forces its first argument to be finite whereas \in_C can hold for some infinite stream. Therefore, implication 1 cannot be an instance of the *coinductive extensionality* (`cext`) principle. An example of the affirmative case is presented in [1, Appendix B], where `cext` effectively establishes the Leibniz equality from the bisimulation $=_{\mathbb{N}^{\text{co}}}$ over conats. This is because conats modulo $=_{\mathbb{N}^{\text{co}}}$ is isomorphic to $\mathbb{N} + 1$.

In our current context, such an isomorphism is not available. We are dealing with non-structural equality over streams of actions. Considering this, we decided to employ a pair of refinement (thus subtyping) relations that solely vary in their action equality checks. It is evident that they essentially serve the same purpose provided the `finiteness` assumption that “in a session with a potentially infinite number of interactions, there can only exist a finite number of distinct actions”.

Note also that in the rest of the paper, we overload the symbol \lesssim to denote the refinement relation (`refinement2`) based on the check given in Definition 15.

4.2 Negation of Refinement

The negation of the refinement relation is inductively defined as a counterpart of the coinductively given refinement relation. We revisit the set of rules originally stated in [22, Fig. 6] and list them in Figure 4 ♣. The rule $[\text{N-ACT}]$ states that if a pair of trees do not have

$$\begin{array}{c} \frac{\text{act}(W) \neq \text{act}(W')}{W \not\lesssim W'} \quad [\text{N-ACT}] \quad \frac{q! \in C^{(p)}}{p?l(S).W \not\lesssim C^{(p)}.p?l'(S').W'} \quad [\text{N-I-O-2}] \\ \\ \frac{\ell \neq \ell'}{p?l(S).W \not\lesssim A^{(p)}.p?l'(S').W'} \quad [\text{N-A-}\ell] \quad \frac{S' \not\leq S}{p?l(S).W \not\lesssim A^{(p)}.p?l(S').W'} \quad [\text{N-A-S}] \quad \frac{S' \leq S \quad W \not\lesssim A^{(p)}.W'}{p?l(S).W \not\lesssim A^{(p)}.p?l(S').W'} \quad [\text{N-A-W}] \\ \\ \frac{\ell \neq \ell'}{p!l(S).W \not\lesssim B^{(p)}.p!l'(S').W'} \quad [\text{N-B-}\ell] \quad \frac{S \not\leq S'}{p!l(S).W \not\lesssim B^{(p)}.p!l(S').W'} \quad [\text{N-B-S}] \quad \frac{S \leq S' \quad W \not\lesssim B^{(p)}.W'}{p!l(S).W \not\lesssim B^{(p)}.p!l(S').W'} \quad [\text{N-B-W}] \end{array}$$

■ **Figure 4** The negation of the refinement relation \lesssim over SISO trees.

the same set of actions (in terms of Definition 15) then they cannot refine each other. We managed to prove in Coq that such kind of terms cannot be in the refinement relation thus they must be in the negation of the refinement; see Lemma 7. The rule $[\text{N-ACT}]$ in fact proves four rules given in the original definition.

► **Lemma 17.** ♣ $\forall p \ell S W W'$,

$$\begin{array}{ll} (1) \quad p! \notin \text{act}(W') & \implies \quad p!l(S).W \not\lesssim W' \quad (2) \quad p? \notin \text{act}(W') & \implies \quad p?l(S).W \not\lesssim W' \\ (3) \quad p! \notin \text{act}(W) & \implies \quad W \not\lesssim p!l(S).W' \quad (4) \quad p? \notin \text{act}(W) & \implies \quad W \not\lesssim p?l(S).W' \end{array}$$

The rule $[\text{N-I-O-2}]$ states that within a pair of terms, if the left term starts with a receive action from a fixed participant p , it cannot refine the right term if the latter contains an arbitrary send action occurring before a receive action from the participant p . This restriction arises from the inability to reorder the actions of the right term such that a $p?$ action moves to the beginning and becomes the leftmost action.

Another crucial aspect of this rule is the renovation of its shape, compared to the one in the original definition:

$$\frac{\text{original definition}}{p?l(S).W \not\leq \mathcal{A}^{(p)}.q!l'(S').W'} \quad \Longrightarrow \quad \frac{\text{reformulated shape}}{p?l(S).W \not\leq \mathcal{C}^{(p)}.p?l'(S').W'} \quad q! \in \mathcal{C}^{(p)}$$

This renovation is advantageous because the rule now adopts a similar structural shape to the other rules. It becomes more readily applicable since the right-hand side exhibits the general structural form of terms with receive actions. This connection is shown in Lemma 12. Also, Lemma 14 becomes applicable when the rule premise fails to be satisfied. Moreover, it makes one of the rules in the original definition ($_{[N-I-O-1]}$) provable with the help of $_{[N-ACT]}$.

► **Lemma 18** ($_{[N-I-O-1]}$). $\spadesuit \forall p q \ell \ell' S S' W W', \quad p?l(S).W \not\leq q!l'(S').W'$.

The last six rules ensure subtle cases involving asynchronous reorderings. The rule $_{[N-A-\ell]}$ claims that terms with mismatching labels cannot refine each other even under $\mathcal{A}^{(p)}$ kind of reordering; $_{[N-A-\beta]}$ and $_{[N-A-W]}$ are variants where sorts and continuations mismatch. In the last line, we have similar kind of rules this time for $\mathcal{B}^{(p)}$ style reorderings.

► **Remark 19.** We can prove six more rules from the original definition of negation relation simply by allowing inductive prefixes $\mathcal{A}^{(p)}$ and $\mathcal{B}^{(p)}$ to contain the empty prefix ε . We suffice to state in Lemma 20 only those related with $_{[N-A-\ell]}$ and $_{[N-B-\ell]}$.

► **Lemma 20.** $\spadesuit \forall p \ell \ell' S S' W W',$
 (1) $\ell \neq \ell' \Longrightarrow p?l(S).W \not\leq p?l'(S').W'$ (2) $\ell \neq \ell' \Longrightarrow p!l(S).W \not\leq p!l'(S').W'$

The negation relation is represented by a standard inductive type called `nRefinement`.

```

Inductive nRefinement: siso → siso → Prop ≙
| n_act : ∀ w w', act_neq (@und w) (@und w') → nRefinement w w'
| n_i_o_2: ∀ w w' p l l' s s' c P Q, isInCp p c = true →
  nRefinement (mk_siso (st_receive p [(l,s,(@und w))]) P)
  (mk_siso (merge_cp_cont p c (st_receive p [(l',s',(@und w'))])) Q)
| n_a_1 : ∀ w w' p l l' s s' a n P Q, l ≠ l' →
  nRefinement (mk_siso (p?? [(l,s,(@und w))]) P)
  (mk_siso (merge_ap_contn p a (p?? [(l',s',(@und w'))]) n) Q)
| n_a_s : ∀ w w' p l s s' a n P Q, nsubsort s' s →
  nRefinement (mk_siso (st_receive p [(l,s,(@und w))]) P)
  (mk_siso (merge_ap_contn p a (st_receive p [(l,s',(@und w'))]) n) Q)
| n_a_w : ∀ w w' p l s s' a n P Q R, subsort s' s →
  nRefinement w (mk_siso (merge_ap_contn p a (@und w') n) P) →
  nRefinement (mk_siso (st_receive p [(l,s,(@und w))]) Q)
  (mk_siso (merge_ap_contn p a (st_receive p [(l,s',(@und w'))]) n) R)
| n_b_s : ∀ w w' p l s s' b n P Q, nsubsort s s' →
  nRefinement (mk_siso (st_send p [(l,s,(@und w))]) P)
  (mk_siso (merge_bp_contn p b (st_send p [(l,s',(@und w'))]) n) Q)
| ...

```

The function `merge_cp_cont` \spadesuit takes a participant p , an instance c of $\mathcal{C}^{(p)}$ and a session tree w and prefixes the actions of c to the tree w . The relation involves constructors `n_b_1` and `n_b_w`, omitted in the snippet, serving as alternatives to rules `n_a_1` and `n_a_w` respectively, with $\mathcal{B}^{(p)}$ sort of prefixing. We develop the relation in Coq over SISO trees, therefore the proof obligation `singleton` needs to be satisfied each time a session tree is used in this context. The parameters P , Q and R denote instances of such proofs.

With all the essential components in place, we are now equipped to define the negation of the subtyping relation, for which the refinement negation $\not\leq$ serves as the basis.

► **Definition 21** (negation of subtyping). \spadesuit For any pair of session trees T, T' ,

$$T \not\leq T' \triangleq \forall i \in I \quad \forall (W_i, W'_i) \quad (W_i \rightsquigarrow T) \Longrightarrow (W'_i \rightsquigarrow T') \Longrightarrow W_i \not\leq W'_i.$$

5 Completeness

Completeness serves as the primary meta property of the subtyping relation (with respect to negation) that we successfully formulated and verified in Coq. In essence, it asserts that for any given pair of session trees T and T' , T is either a subtype of T' or it is linked to T' by the negation of the subtyping relation, leaving no room for a third possibility. The subtyping completeness proof relies on the completeness of the revisited refinement relation (§ 4.1), as formally delineated below.

► **Lemma 22** (refinement completeness). \clubsuit *For any pair of SISO trees W, W' , we have*

$$\neg(W \lesssim W') \iff W \not\lesssim W'.$$

Proof. (\Rightarrow) \clubsuit To establish the left-to-right implication, we initially prove $\neg(W \not\lesssim W') \implies W \lesssim W'$, followed by deducing its contrapositive. This choice is motivated by the observation that in Coq proofs, the presence of the negation of a coinductively defined term within the goal context lacks utility, as its inversion fails to produce useful equations \clubsuit .

```
Lemma nRefLH:  $\forall w w', (\text{nRefinement } w w' \rightarrow \text{False}) \rightarrow \text{refinement2 } (@\text{und } w) (@\text{und } w')$ .
```

The proof proceeds by storing the proof state in a coinduction hypothesis CIH following the decomposition of w and w' into pairs of their respective underlying session trees and proofs confirming that they are singletons, namely into (w, Pw) and (w', Pw') .

$\text{CIH}: \forall (w': \text{st}) (Pw' \text{ singleton } w') (w: \text{st}) (Pw: \text{ singleton } w),$
 $(\text{nRefinement } \{\text{und} \triangleq w; \text{sprop} \triangleq Pw\} \{\text{und} \triangleq w'; \text{sprop} \triangleq Pw'\} \rightarrow \text{False}) \rightarrow r w w'$
 CIH is parametrised by the binary relation r over session trees which signifies the accumulated knowledge derived from coinductive foldings of the refinement relation. The rest relies on the inversion lemma `sinv` over SISO trees which discusses the possible shapes they could exhibit: a SISO tree is a streamline of actions that initiates with a send or receive action, or it is simply an end \clubsuit .

```
Lemma sinv:  $\forall w, \text{singleton } w \rightarrow$   

 $(\exists p l s w', w = \text{st\_send } p [(l, s, w')] \wedge \text{singleton } w') \vee$   

 $(\exists p l s w', w = \text{st\_receive } p [(l, s, w')] \wedge \text{singleton } w') \vee (w = \text{st\_end}).$ 
```

Therefore considering the potential shapes of w and w' , the left-to-right proof is made of nine distinct cases. Here we focus on the one where both of the trees start with receive actions such that $w_1 = (p?'[(1, s, w_1)])$ and $w_2 = (q?'[(1', s', w_2)])$ for some p, q, l, l', s, s', w_1 and w_2 such that w_1 and w_2 are indeed singleton trees.

1. We have a case distinction on the fact that $p?' \in \text{act}(w')$. If w' does not contain the $p?'$ action, the goal is a trivial application of the rule `n_act`. Otherwise, we get $w' = \text{merge_cp_cont } p c (p?'[(11, s1, w3)])$ thanks to Lemma 12 for some prefix c , label 11 , sort $s1$ and term $w3$.

We then apply a further case analysis on $p! \in c$. The positive case is a direct implication of the rule `n_i_o_2`. In the negative case, w' takes the shape of `merge_ap_cont p a (p?'[(11, s1, w3)])` for some prefix a due to Lemma 14, transforming the goal into `paco2 refinementR2 r (p?'[(1, s, w1)]) (merge_ap_cont p a (p?'[(11, s1, w3)]))` which is solved by case distinctions described in below items 2 to 4.

2. When $l = l'$, s' is a subsort of s and w_1 and $(\text{merge_ap_cont } p a w_3)$ are of the same actions, we apply the constructor `ref_a` with the prefix $a \triangleq a$ which entails a subgoal `upaco2 refinementR2 r w1 (merge_ap_cont p a w3)`.

To close the subgoal, we do not further fold the coinductive relation `refinementR2`, instead employ the coinduction hypothesis `CIH`. Then, the objective is to show that `nRefinement w1 (merge_ap_cont p a w3) → False` holds under the initial assumption `nRefinement w w' → False`. This is addressed by the rule `n_a_w` with $a \triangleq a$.

3. In case $l = l'$, s' is a subset of s and $w1$ and $(\text{merge_ap_cont } p \ a \ w3)$ are of the different actions, we can deduce that `nRefinement w w'` thanks to the rule `n_act`. This contradicts with the initial assumption of `nRefinement w w' → False` and closes the case.
4. The cases where $l \neq l'$ or s' is not a subset of s are handled by rules `n_a_l` and `n_a_s`. ◀

Proof. (\Leftarrow) \clubsuit . The right-to-left implication reflects into Coq as follows.

```
Lemma nRefR: ∀ w w', nRefinement w w' → (refinement2 (@und w) (@und w') → False).
```

The proof argues by structural induction over the negation relation and is made of eight cases. Here, we present a selected case associated to the rule `n_b_s`. The refinement assumption in this case is of the shape: $H: \text{refinementR2 } (\text{upaco2 refinementR2 bot2}) (p![(l,s,w)]) (\text{merge_bp_contn } p \ b \ (p![(l,s',w')])) \ n$ for some n, w, w', s and s' such that s is not a subset of s' . Inverting H results in proving `False` provided following equations.

1. $p![(l,s'0,w'0)] = \text{merge_bp_contn } p \ b \ (p![(l,s',w')]) \ n$ for some $w'0, s'0$ such that w refines $w'0$ and s is a subset of $s'0$;
2. $\text{merge_bp_contn } p \ b0 \ (p![(l,s'0,w'0)]) \ n0 = \text{merge_bp_contn } p \ b \ (p![(l,s',w')]) \ n$ for some $n0, b0, w'0, s'0$ such that w refines $\text{merge_bp_contn } p \ b0 \ w'0 \ n0$, s is a subset of $s'0$ and w contains the same actions with $\text{merge_bp_contn } p \ b0 \ w'0 \ n0$.

The initial falsity is demonstrated through a case analysis over n (number of times the prefix is repeated) and subsequent case analysis over b (the prefix) when $n \geq 0$. Each of these cases is resolved by establishing contradictions within the context: either an equality between a send and a receive action, a dis-equality between the same actions, or hypotheses asserting both that s is a subset of s' and its negation at the same time.

The proof of the second falsity is somewhat more intricate and relies on the `meqBp` lemma provided below. This lemma establishes the structural equality between *merging a term once with a single sequence of actions captured after n iterations of appending a given prefix with itself* and *merging the term with the given prefix n times*.

The function `BpnA` \clubsuit handles the appending of a given prefix b to itself n times and constructs a sequence of actions from it. And, the function `merge_bp_cont` is a variant of `merge_bp_contn` with n set to 1.

```
Lemma meqBp: ∀ n p b w, merge_bp_cont p (BpnA p b n) w = merge_bp_contn p b w n.
```

We rewrite the lemma `meqBp` \clubsuit in the hypothesis and transform it into `merge_bp_cont p (Bpn p b0 n0) (p![(l,s'0,w'0)]) = merge_bp_cont p (Bpn p b n) (p![(l,s',w')])`. It is now possible to infer that $(p![(l,s'0,w'0)]) = (p![(l,s',w')])$, hence $s'0 = s'$ and $w'0 = w'$, thanks to the lemma `BpBpeqInv2` \clubsuit .

```
Lemma BpBpeqInv2: ∀ p b1 b2 l1 l2 s1 s2 w1 w2,
merge_bp_cont p b1 (p![(l1,s1,w1)]) = merge_bp_cont p b2 (p![(l2,s2,w2)]) →
(p![(l1,s1,w1)]) = (p![(l2,s2,w2)]).
```

13:16 Completeness of Asynchronous Session Tree Subtyping in Coq

We can now close the goal simply by plugging $s0' = s'$ in. This equation leads to inconsistency in the proof context as we then obtain proofs of “ s is not a subsort of s' ” and “ s is a subsort of s' ” at the same time. ◀

► **Corollary 23** (subtyping completeness). *For any pair session trees T, T' , we have*

$$\neg(T \leq T') \iff T \not\leq T'.$$

Proof. Follows from Lemma 22. ◀

```
Lemma subNeqL: ∀ T T', (subtype2 T T' → False) → nsubtype T T'.
Lemma subNeqR: ∀ T T', nsubtype T T' → (subtype2 T T' → False).
Theorem completeness: ∀ T T', (subtype2 T T' → False) ↔ nsubtype T T'.
Proof. split; [ apply (subNeqL T T') | intros. apply (subNeqR T T'); easy ]. Qed.
```

Axiomatic Base and Mechanisation Effort. In the accompanying library, we employ classical reasoning to conduct case analysis primarily over coinductively defined predicates. We also use the proof irrelevance axiom to obtain that different proofs of dis-equality among the same pair of participants are treated the same. The library comprises around 10K lines of code, containing 250 proven lemmata and 166 definitions, with 35 of them being coinductively stated. Initially, integrating inductive and coinductive reasoning seemed challenging, but it scaled remarkably well with the aid of the `paco` library.

6 Related Work and Conclusion

Asynchronous session subtyping was first introduced to achieve message optimization in session-based high-performance computing platforms, i.e., multicore C programming [28, 42] and MPI-C [34, 33]. Then, numerous theoretical and practical advancements have emerged.

In theory, Chen et al. [11] introduced and proved *preciseness* of synchronous [18, 15] and asynchronous subtyping for the binary (2-party) session types. Later, the asynchronous subtyping was found undecidable, independently by Bravetti et al. [6], and by Lange and Yoshida [32]. This provoked active studies on (1) identifying a set of binary session types where asynchronous subtyping is decidable [7, 32]; and (2) proposing *sound* algorithms extending the formalism to *binary* communicating automata [4] in [5, 2] (also to fair refinement in [8]). In the multiparty setting, Ghiezan et al. [20, 21, 22] proposed precise synchronous and asynchronous session subtyping employing coinductive axiomatisation.

In practice, Castro-Perez and Yoshida [10] examined a constrained version of multiparty asynchronous subtyping algorithm where permutations across unrolling recursions are prohibited. This framework has been used for the cost analysis of optimised C code. Cutner et al. [14] proposed a sound multiparty synchronous subtyping algorithm and integrated it into Rust. Neither of the multiparty algorithms in [10] and [14] nor the one for binary sessions types in [5] can validate [22, Example 3.19]. In a recent study [2, Figure 6], Bocchi et al. presented an extended version of the algorithm proposed in [5], incorporating program analysis techniques. They effectively validated the example, albeit the algorithm is limited to the binary setting. We mechanised and proved this example in Coq (Figure 5) within a multiparty setting.

The mechanisation approach we employ is not bounded by the undecidability of asynchronous subtyping, as subtyping is axiomatised as a coinductive relation in Coq. It is non-computational. The key point we make in Figure 5 is to show that our subtyping technique and its implementation in Coq are expressive enough to cover several examples that have been proven using different automated tools.

	[5]	[2]	[10]	[14]	Ours
<code>ring-choice</code> [13]	✗	✗	✓	✓	✓
Example 3.17 [22]	✗	✗	✓	✓	✓
Example 3.19 [22]	✗	✓	✗	✗	✓
Example 4.14 [22]	✗	✗	✗	✓	✓

■ **Figure 5** Examples and related work.

Mechanisation recently emerges as a pivotal facet in the concurrent communication models. Tireore et al. [40] introduced a novel projection function that maps global multiparty session types to local types. This function has been demonstrated to be both sound and complete with respect to its coinductive counterpart. It has been implemented in Coq and its mentioned properties have been formally proven there. Castro-Perez et al. [9] built a domain-specific language named Zooid, and implemented in Coq. Zooid allows for the extraction of certified synchronously interacting programs built upon MPST. Brady [3] devised secure communication protocols for binary sessions in Idris. Thiemann et al. [39] formalised progress and preservation properties for binary session types in Agda. Tassarotti et al. [37] developed a compiler grounded in a simplified version of the GV system [19] for a functional language equipped with binary session types. The correctness of this compiler has been proven in Coq. Jacobs et al. [30] expanded a functional language with multiparty session types (MPGV) and formally verified, in Coq, that the language is deadlock-free. Hinrichsen et al. [25, 23, 24] introduced Actris, a Coq tool that integrates separation logics and asynchronous binary session types with an asynchronous subtyping mechanism. Actris is developed as an extension to the Iris program logic. Jacobs et al. extended Actris by linear logic into LinearActris in their recent work [31] to freely obtain deadlock and leak freedom for binary session types from linearity. Choreographic programming paradigm allows one to implement distributed programs as single programs, ensuring coherence between send and receive operations by consolidating them into a single construct. Deadlock freedom is inherit in the design. Cruz-Filipe et al. [12] formalised the theory of choreographic programming in Coq. In their work [26], Hirsch and Garg introduced Pirouette, a choreographic language designed with formal guarantees, which are rigorously verified in Coq. Similarly, in [35], Pohjola et al. presented Kalas, a compiler for a choreographic language correctness of which has been verified using the HOL4 theorem prover.

Conclusion and Future Work. In this paper, we present the first formalisation of asynchronous subtyping for session trees, establishing a framework for asynchronous subtyping in MPST. The formalisation (1) decomposes arbitrary session trees into SISO trees that are free of choice and selection, and (2) governs the subtyping relation through *refinement* of these trees. It has been used to certify four illustrative protocol optimisation examples presented in the literature. See Figure 5.

In the development, we redefined the negation of the refinement relation, addressing the incompleteness issue spotted in the prior work [22], and proved that subtyping is complete with respect to the renovated negation. To determine the precise configuration for refinement and its negation, we employed a new sort of term prefixing.

Our future plan includes establishing a Coq proof of the soundness of subtyping with respect to *liveness*, a behavioural property of typing environments ensuring that every pending send eventually enqueues messages and every pending reception is eventually executed.

References

- 1 Alexander Bagnall, Gordon Stewart, and Anindya Banerjee. Inductive reasoning for coinductive types. *CoRR*, abs/2301.09802, 2023. doi:10.48550/arXiv.2301.09802.
- 2 Laura Bocchi, Andy King, and Maurizio Murgia. Asynchronous subtyping by trace relaxation. In Bernd Finkbeiner and Laura Kovács, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 30th International Conference, TACAS 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Proceedings, Part I*, volume 14570 of *Lecture Notes in Computer Science*, pages 207–226. Springer, 2024. doi:10.1007/978-3-031-57246-3_12.
- 3 Edwin C. Brady. Type-driven development of concurrent communicating systems. *Comput. Sci.*, 18(3), 2017. doi:10.7494/CSCI.2017.18.3.1413.
- 4 Daniel Brand and Pitro Zafiropulo. On communicating finite-state machines. *J. ACM*, 30(2):323–342, 1983. doi:10.1145/322374.322380.
- 5 Mario Bravetti, Marco Carbone, Julien Lange, Nobuko Yoshida, and Gianluigi Zavattaro. A Sound Algorithm for Asynchronous Session Subtyping and its Implementation. *Logical Methods in Computer Science*, Volume 17, Issue 1, March 2021. doi:10.23638/LMCS-17(1:20)2021.
- 6 Mario Bravetti, Marco Carbone, and Gianluigi Zavattaro. Undecidability of asynchronous session subtyping. *Inf. Comput.*, 256:300–320, 2017.
- 7 Mario Bravetti, Marco Carbone, and Gianluigi Zavattaro. On the boundary between decidability and undecidability of asynchronous session subtyping. *Theor. Comput. Sci.*, 722:19–51, 2018. doi:10.1016/j.tcs.2018.02.010.
- 8 Mario Bravetti, Julien Lange, and Gianluigi Zavattaro. Fair refinement for asynchronous session types. In *FoSSaCS*, Lecture Notes in Computer Science, 2021.
- 9 David Castro-Perez, Francisco Ferreira, Lorenzo Gheri, and Nobuko Yoshida. Zoid: a DSL for certified multiparty computation: from mechanised metatheory to certified multiparty processes. In Stephen N. Freund and Eran Yahav, editors, *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, pages 237–251. ACM, 2021. doi:10.1145/3453483.3454041.
- 10 David Castro-Perez and Nobuko Yoshida. CAMP: cost-aware multiparty session protocols. *Proc. ACM Program. Lang.*, 4(OOPSLA):155:1–155:30, 2020. doi:10.1145/3428223.
- 11 Tzu-Chun Chen, Mariangiola Dezani-Ciancaglini, Alceste Scalas, and Nobuko Yoshida. On the Preciseness of Subtyping in Session Types. *LMCS*, 13:1–62, 2017.
- 12 Luís Cruz-Filipe, Fabrizio Montesi, and Marco Peressotti. A formal theory of choreographic programming. *J. Autom. Reason.*, 67(2):21, 2023. doi:10.1007/S10817-023-09665-3.
- 13 Zak Cutner and Nobuko Yoshida. Safe Session-Based Asynchronous Coordination in Rust. In Ferruccio Damiani and Ornella Dardha, editors, *Coordination Models and Languages - 23rd IFIP WG 6.1 International Conference, COORDINATION 2021, Held as Part of the 16th International Federated Conference on Distributed Computing Techniques, DisCoTec 2021, Valletta, Malta, June 14-18, 2021, Proceedings*, volume 12717 of *Lecture Notes in Computer Science*, pages 80–89. Springer, 2021. doi:10.1007/978-3-030-78142-2_5.
- 14 Zak Cutner, Nobuko Yoshida, and Martin Vassor. Deadlock-free asynchronous message reordering in Rust with multiparty session types. In Jaejin Lee, Kunal Agrawal, and Michael F. Spear, editors, *PPoPP '22: 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Seoul, Republic of Korea, April 2 - 6, 2022*, pages 246–261. ACM, 2022. doi:10.1145/3503221.3508404.
- 15 Romain Demangeon and Kohei Honda. Full Abstraction in a Subtyped pi-Calculus with Linear Types. In *22nd International Conference on Concurrency Theory*, volume 6901 of *LNCS*, pages 280–296. Springer, 2011.
- 16 Romain Demangeon and Kohei Honda. Nested protocols in session types. In Maciej Koutny and Irek Ulidowski, editors, *CONCUR 2012 - Concurrency Theory - 23rd International Conference, CONCUR 2012, Newcastle upon Tyne, UK, September 4-7, 2012. Proceedings*,

- volume 7454 of *Lecture Notes in Computer Science*, pages 272–286. Springer, 2012. doi:10.1007/978-3-642-32940-1_20.
- 17 Burak Ekici and Nobuko Yoshida. <https://github.com/ekiciburak/sessionTreeST/tree/locatype>. Software, swbId: swb:1:dir:33823a0054801bcf4ea95f2dffe733579cbd53c8 (visited on 2024-08-20). URL: <https://github.com/ekiciburak/sessionTreeST/tree/itp2024>.
 - 18 Simon J. Gay and Malcolm Hole. Subtyping for session types in the pi calculus. *Acta Inf.*, 42(2-3):191–225, 2005. doi:10.1007/s00236-005-0177-z.
 - 19 Simon J. Gay and Vasco Thudichum Vasconcelos. Linear type theory for asynchronous session types. *J. Funct. Program.*, 20(1):19–50, 2010. doi:10.1017/S0956796809990268.
 - 20 Silvia Ghilezan, Svetlana Jaksic, Jovanka Pantovic, Alceste Scalas, and Nobuko Yoshida. Precise subtyping for synchronous multiparty sessions. *JLAMP*, 104:127–173, 2019.
 - 21 Silvia Ghilezan, Jovanka Pantovic, Ivan Prokic, Alceste Scalas, and Nobuko Yoshida. Precise Subtyping for Asynchronous Multiparty Sessions. *Proc. ACM Program. Lang.*, 5:16:1–16:28, January 2021.
 - 22 Silvia Ghilezan, Jovanka Pantović, Ivan Prokić, Alceste Scalas, and Nobuko Yoshida. Precise subtyping for asynchronous multiparty sessions. *ACM Trans. Comput. Logic*, 24(2), November 2023. doi:10.1145/3568422.
 - 23 Jonas Kastberg Hinrichsen, Jesper Bengtson, and Robbert Krebbers. Actris: session-type based reasoning in separation logic. *Proc. ACM Program. Lang.*, 4(POPL):6:1–6:30, 2020. doi:10.1145/3371074.
 - 24 Jonas Kastberg Hinrichsen, Jesper Bengtson, and Robbert Krebbers. Actris 2.0: Asynchronous session-type based reasoning in separation logic. *Log. Methods Comput. Sci.*, 18(2), 2022. doi:10.46298/LMCS-18(2:16)2022.
 - 25 Jonas Kastberg Hinrichsen, Daniël Louwriink, Robbert Krebbers, and Jesper Bengtson. Machine-checked semantic session typing. In Catalin Hritcu and Andrei Popescu, editors, *CPP '21: 10th ACM SIGPLAN International Conference on Certified Programs and Proofs, Virtual Event, Denmark, January 17-19, 2021*, pages 178–198. ACM, 2021. doi:10.1145/3437992.3439914.
 - 26 Andrew K. Hirsch and Deepak Garg. Pirouette: higher-order typed functional choreographies. *Proc. ACM Program. Lang.*, 6(POPL):1–27, 2022. doi:10.1145/3498684.
 - 27 Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *ESOP 1998*, pages 122–138, 1998. doi:10.1007/BFb0053567.
 - 28 Kohei Honda, Vasco Thudichum Vasconcelos, and Nobuko Yoshida. Type-Directed Compilation for Multicore Programming. *ENTCS*, 241:101–111, 2009.
 - 29 Chung-Kil Hur, Georg Neis, Derek Dreyer, and Viktor Vafeiadis. The power of parameterization in coinductive proof. 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 193–206. ACM, 2013. doi:10.1145/2429069.2429093.
 - 30 Jules Jacobs, Stephanie Balzer, and Robbert Krebbers. Multiparty GV: functional multiparty session types with certified deadlock freedom. *Proc. ACM Program. Lang.*, 6(ICFP):466–495, 2022. doi:10.1145/3547638.
 - 31 Jules Jacobs, Jonas Kastberg Hinrichsen, and Robbert Krebbers. Deadlock-free separation logic: Linearity yields progress for dependent higher-order message passing. *Proc. ACM Program. Lang.*, 8(POPL):1385–1417, 2024. doi:10.1145/3632889.
 - 32 Julien Lange and Nobuko Yoshida. On the undecidability of asynchronous session subtyping. In *FoSSaCS*, volume 10203 of *Lecture Notes in Computer Science*, pages 441–457, 2017.
 - 33 Nicholas Ng, Jose G.F. Coutinho, and Nobuko Yoshida. Protocols by Default: Safe MPI Code Generation based on Session Types. In *24th International Conference on Compiler Construction*, volume 9031 of *LNCS*, pages 212–232. Springer, 2015.

- 34 Nicholas Ng, Nobuko Yoshida, and Kohei Honda. Multiparty Session C: Safe Parallel Programming with Message Optimisation. In *50th International Conference on Objects, Models, Components, Patterns*, volume 7304 of *LNCs*, pages 202–218. Springer, 2012.
- 35 Johannes Åman Pohjola, Alejandro Gómez-Londoño, James Shaker, and Michael Norrish. Kalas: A verified, end-to-end compiler for a choreographic language. In June Andronick and Leonardo de Moura, editors, *13th International Conference on Interactive Theorem Proving, ITP 2022, August 7-10, 2022, Haifa, Israel*, volume 237 of *LIPICs*, pages 27:1–27:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPICs.ITP.2022.27.
- 36 Kaku Takeuchi, Kohei Honda, and Makoto Kubo. An interaction-based language and its typing system. In *PARLE 1994*, pages 398–413, 1994. doi:10.1007/3-540-58184-7_118.
- 37 Joseph Tassarotti, Ralf Jung, and Robert Harper. A higher-order logic for concurrent termination-preserving refinement. In Hongseok Yang, editor, *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*, volume 10201 of *Lecture Notes in Computer Science*, pages 909–936. Springer, 2017. doi:10.1007/978-3-662-54434-1_34.
- 38 The Coq Development Team. The Coq reference manual – release 8.18.0. <https://coq.inria.fr/doc/V8.18.0/refman>, 2023.
- 39 Peter Thiemann. Intrinsically-typed mechanized semantics for session types. In Ekaterina Komendantskaya, editor, *Proceedings of the 21st International Symposium on Principles and Practice of Programming Languages, PPDP 2019, Porto, Portugal, October 7-9, 2019*, pages 19:1–19:15. ACM, 2019. doi:10.1145/3354166.3354184.
- 40 Dawit Legesse Tirore, Jesper Bengtson, and Marco Carbone. A sound and complete projection for global types. In Adam Naumowicz and René Thiemann, editors, *14th International Conference on Interactive Theorem Proving, ITP 2023, July 31 to August 4, 2023, Białystok, Poland*, volume 268 of *LIPICs*, pages 28:1–28:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPICs.ITP.2023.28.
- 41 Nobuko Yoshida and Lorenzo Gheri. A very gentle introduction to multiparty session types. In Dang Van Hung and Meenakshi D’Souza, editors, *Distributed Computing and Internet Technology - 16th International Conference, ICDCIT 2020, Bhubaneswar, India, January 9-12, 2020, Proceedings*, volume 11969 of *Lecture Notes in Computer Science*, pages 73–93. Springer, 2020. doi:10.1007/978-3-030-36987-3_5.
- 42 Nobuko Yoshida, Vasco Thudichum Vasconcelos, Hervé Paulino, and Kohei Honda. Session-based compilation framework for multicore programming. In *FMCO 2008*, pages 226–246, 2008. doi:10.1007/978-3-642-04167-9_12.
- 43 Yannick Zakowski, Paul He, Chung-Kil Hur, and Steve Zdancewic. An equational theory for weak bisimulation via generalized parameterized coinduction. In Jasmin Blanchette and Catalin Hritcu, editors, *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020*, pages 71–84. ACM, 2020. doi:10.1145/3372885.3373813.

End-To-End Formal Verification of a Fast and Accurate Floating-Point Approximation

Florian Faissole ✉ 

Mitsubishi Electric R&D Centre Europe, 35700 Rennes, France

Paul Geneau de Lamarlière ✉ 

Mitsubishi Electric R&D Centre Europe, 35700 Rennes, France

Université Paris-Saclay, CNRS, ENS Paris-Saclay, Inria, LMF, 91190 Gif-sur-Yvette, France

Guillaume Melquiond ✉ 

Université Paris-Saclay, CNRS, ENS Paris-Saclay, Inria, LMF, 91190 Gif-sur-Yvette, France

Abstract

Designing an efficient yet accurate floating-point approximation of a mathematical function is an intricate and error-prone process. This warrants the use of formal methods, especially formal proof, to achieve some degree of confidence in the implementation. Unfortunately, the lack of automation or its poor interplay with the more manual parts of the proof makes it way too costly in practice. This article revisits the issue by proposing a methodology and some dedicated automation, and applies them to the use case of a faithful *binary64* approximation of exponential. The peculiarity of this use case is that the target of the formal verification is not a simple modeling of an external code; it is an actual floating-point function defined in the logic of the Coq proof assistant, which is thus usable inside proofs once its correctness has been fully verified. This function presents all the attributes of a state-of-the-art implementation: bit-level manipulations, large tables of constants, obscure floating-point transformations, exceptional values, etc. This function has been integrated into the proof strategies of the CoqInterval library, bringing a 20× speedup with respect to the previous implementation.

2012 ACM Subject Classification Software and its engineering → Formal software verification; Theory of computation → Interactive proof systems; Theory of computation → Automated reasoning; Mathematics of computing → Mathematical software performance; Mathematics of computing → Interval arithmetic

Keywords and phrases Program verification, floating-point arithmetic, formal proof, automated reasoning, mathematical library

Digital Object Identifier 10.4230/LIPIcs.ITP.2024.14

Supplementary Material *Software*: <https://gitlab.inria.fr/coqinterval/interval.git> [6]
archived at [swh:1:dir:78da3e6e98b7ef018180119255ce1e10a048cc88](https://swh.1:dir:78da3e6e98b7ef018180119255ce1e10a048cc88)

Funding *Guillaume Melquiond*: This work was partly supported by the NuSCAP project (ANR-20-CE48-0014) of the French national research agency (ANR).

1 Introduction

A mathematical library is a set of floating-point functions that are designed to approximate mathematical functions. They are used in various domains, ranging from engineering to scientific computing and experimental mathematics. For such applications, these functions are required to be both accurate and fast to compute. To meet those requirements, the code of such a floating-point function is usually intricate and its correctness is far from trivial [14]. This warrants verifying the latter formally, which can be long and tedious [8, 9].



© Florian Faissole, Paul Geneau de Lamarlière, and Guillaume Melquiond;
licensed under Creative Commons License CC-BY 4.0

15th International Conference on Interactive Theorem Proving (ITP 2024).

Editors: Yves Bertot, Temur Kutsia, and Michael Norrish; Article No. 14; pp. 14:1–14:18

Leibniz International Proceedings in Informatics



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

Formally verified mathematical libraries can even be used for formal proofs. An example of such usage is the `CoqInterval` library,¹ which provides a set of strategies for the Coq proof assistant that automate the verification of enclosures of real-valued expressions. It is based on a formalization of rigorous polynomial approximations that are computed using an interval arithmetic with floating-point bounds [11]. Originally, the floating-point computations were performed one bit at a time in the logic of the Coq system. Later, support for 63-bit integers was added to Coq to speed up computation [5]. Even then, a formal verification of the following approximation of Siegfried Rump’s integral – an example known to cause computer algebra systems to struggle due to the large number of oscillations of the integrand – would still take minutes to complete:

$$\int_0^8 \sin(x + \exp x) dx = 0.3474 \pm 10^{-6}.$$

Indeed, proving this approximation requires computing polynomial approximations of the integrand on numerous subintervals of $[0; 8]$, which itself requires computing enclosures of the sine and exponential functions on many inputs.

To improve performance, recent work has added support for performing hardware floating-point computations inside Coq proofs [12]. These built-in operations are axiomatized with a purely computational specification, which has been formally proved to comply with the IEEE 754 standard thanks to the `Flocq` library [3]. This makes it possible to both trust the specification and use it for formal reasoning. Thanks to this new feature, the time needed to verify the above approximation is reduced to just a few seconds.

This remains much slower than what could be achieved by state-of-the-art libraries [17, §12]. Part of the reason is that we are performing computations inside the logic system, but also that the code itself uses hardware floating-point numbers in a very naive way. To make `CoqInterval` even more suitable for this kind of numerically intensive proofs, we would like to improve on this last point by providing it with a mathematical library that is specialized for hardware floating-point numbers and that is also formally verified. The work presented in this article focuses on the implementation of the exponential function.

1.1 The original `CoqInterval` implementation

Prior to this work, `CoqInterval` would use a single algorithm for the exponential function, but instantiated twice: once for floating-point numbers computed in hardware and once for floating-point numbers slowly emulated in the logic of Coq.² This was made possible thanks to a suitable abstraction of floating-point arithmetic [12]. Having a single algorithm, and thus a single proof of correctness, made the large formalization effort that went into adding hardware computations to `CoqInterval` much less tedious.

`CoqInterval`’s original algorithm for computing an enclosure of $\exp x$ goes as follows. First, using the following mathematical identities, an argument reduction brings the input x into the interval $[-2^{-8}; 0]$:

$$\begin{aligned} \exp x &= (\exp(-x))^{-1} \\ \exp x &= (\exp(x/2))^2 \end{aligned} \tag{1}$$

¹ <https://coqinterval.gitlabpages.inria.fr/>

² While slow, this emulation of floating-point arithmetic is still useful for proofs that require more than the 53 bits of precision provided by the *binary64* format.

■ **Listing 1** Guaranteed approximation of exponential in OCaml. The output is a pair of floating-point numbers that enclose $\exp x$. Symbols `invLog2_64`, `log2_64h`, `log2_64l`, `cst`, `p1`, `p2`, etc, are predefined floating-point literals.

```
let iexp x =
  if x < -0x1.74385446d71c4p9 then (0., 0x1.p-1074) else
  if x > 0x1.62e42fefa39efp9 then (0x1.fffffffffff2ap1023, infinity) else
  let k' = x *. invLog2_64 +. 0x1.8p52 in
  let k = k' -. 0x1.8p52 in
  let t = (x -. k *. log2_64h) -. k *. log2_64l in
  let y = t *. (p1 +. t *. (p2 +. t *. (... + t *. p5))) in
  let ki = int_of_float k' - 0x18000000000000 in
  let p0 = cst.(ki land 63) in
  let d = 0x1.25p-57 in
  let lb = p0 +. (p0 *. y -. d) in
  let ub = p0 +. (p0 *. y +. d) in
  next_down (ldexp lb (ki asr 6)), next_up (ldexp ub (ki asr 6))
```

Second, the alternating series $\exp(-x) = \sum(-x)^n/n!$ is computed using interval arithmetic to a high enough order. Thanks to the use of interval arithmetic and an alternating series, the algorithm is guaranteed to compute an enclosure of the real number $\exp x$. Third, the argument reduction is reversed to reconstruct the final interval result.

By suitably choosing the order of truncation of the series, one can obtain arbitrarily tight enclosures of $\exp x$, assuming that the precision of the floating-point arithmetic used to compute the interval bounds can be made accordingly large. This property is invaluable when used in conjunction with the original multi-precision floating-point arithmetic of CoqInterval. But for hardware floating-point numbers and their fixed precision of 53 bits, the property is pointless. The inadequacies of the implementation of `exp` then become prominent. First, Equation (1) means that computing an enclosure of $\exp x$ is not constant time, but proportional to the magnitude of x . Second, an alternating series is the worst way of approximating a value, as part of the computations performed at order i are immediately canceled by those at order $i + 1$ and thus have been performed in vain. Third, while interval arithmetic is correct by construction, hence very proof-friendly, it performs twice as many floating-point operations as needed.

1.2 The whole new algorithm

An approximation of the exponential function, as found in usual mathematical libraries, does not suffer from these defects, as it is generally implemented along the following guidelines [14, §6.2]. It would first perform a constant-time argument reduction using the following mathematical identity:

$$\exp x = \exp(x - k \cdot \ln 2) \cdot 2^k \quad \text{with } k = \lceil x / \ln 2 \rceil \in \mathbb{Z}.$$

where $\lceil \cdot \rceil$ denotes the nearest integer. Then, a low-degree polynomial approximation of \exp around 0 would be evaluated. Finally, the result reconstruction is trivial, as it is a multiplication by a power of two. The whole algorithm amounts to just a few tens of operations; it is thus extremely fast.

Listing 1 shows the implementation we have devised, represented as an OCaml function for readability. (Its translation to Coq's λ -calculus is straightforward.) Given a finite floating-point number x , the code computes a pair of floating-point numbers enclosing $\exp x$. In particular, it uses the functions `next_down` and `next_up` to compute the predecessor and successor of a floating-point number.

While the code seems to contain useless, if not adverse, floating-point computations, this is not the case. For example, `k` looks like it could be directly computed as `x *. invLog2_64` by canceling `0x1.8p52 -. 0x1.8p52`. This optimization would completely break the function, causing it to no longer approximate the exponential, not even roughly. Similarly, `t` should not be rewritten as `x -. k *. (log2_64h +. log2_64l)`, and `d` should not be moved outside of the parentheses in the computations of `lb` and `ub`. So, not only does floating-point arithmetic ignore the usual algebraic laws of associativity and distributivity, but floating-point experts actively rely on the lack of these laws to compute more accurate approximations.

1.3 Challenges of the formal verification

Contrarily to the previous algorithm, the adequacy of this new implementation of the exponential no longer derives from the use of interval arithmetic, so the proof is no longer straightforward. But at the same time, the proof effort needs to be sufficiently light so that it is worth replacing a feature that is already good enough for most use cases.

There have been several attempts at formally verifying this kind of state-of-the-art implementation using the Coq proof assistant, but they all have suffered from various shortcomings. It might have been that the floating-point arithmetic was modeled without any exceptional value [3, §6.2.3]. Indeed, when a computer-assisted proof is meant to complement a pen-and-paper proof, it is acceptable that it only focuses on the most intricate parts of the proof, which the absence of exceptional behavior is hardly ever. But, since this idealized arithmetic does not match the behavior of hardware floating-point numbers, it cannot be used here. Some later attempt solved the issue of the exceptional values [7], but it was still targeting the verification of some code meant to run outside of Coq and thus did not need to cover all of its facets. On the contrary, the algorithm shown in Listing 1 will effectively be run when checking subsequent Coq proofs, so absolutely no shortcuts can be taken.

1.4 Contributions

This article proposes a fully proven, fast, and accurate implementation of the exponential function for `CoqInterval`. The intricacy of this implementation corresponds to what is typically found in the state of the art: tables of precomputed values, mixed floating-point integer operations, etc. The proof covers all aspects of the correctness: the argument reduction, the polynomial approximation, and the reconstruction. In addition, this article describes our methodology for formally verifying floating-point approximations of mathematical functions. In particular, we will present the automated strategies that were added to make this verification as painless as possible.

Section 2 reminds both the arithmetic language and the notion of well-behaved expression that were introduced in a previous work [7]. Section 3 explains how some strategies of `CoqInterval` have been improved to automatically verify properties involving tight bounds on rounding errors. Section 4 details the new features added to the arithmetic language and associated tools to tackle the algorithm of Listing 1: hardware floating-point numbers, conversions, macro-operations, array accesses, etc. Section 5 describes the methodology used to formally verify the correctness of exponential, as well as some unusual properties of floating-point arithmetic we ended up with. Section 6 explains how this work relates to some other works. Finally, Section 7 concludes with some benchmarks and some perspectives.

2 Preliminaries

This work is partly built on top of a framework for modeling floating-point expressions [7]. In particular, that framework provides some facilities to automate the proof of the absence of exceptional behaviors, thus making it possible for the user to focus on a modeling of floating-point expressions as real numbers. This section reminds the features of that framework that are the most relevant to the presented work. Section 2.1 focuses on the expressions and their various interpretations, while Section 2.2 shows how one can jump between interpretations to ease the proof process.

In the following, the unary operator $\circ(\cdot)$ designates a rounding operator from \mathbb{R} to \mathbb{R} ; it returns the real number the nearest to the input that fits in the target floating-point format (with unlimited range) [3, §3.2.2]. This theoretical operator is at the core of the IEEE-754 standard for floating-point arithmetic.

2.1 Arithmetic expressions

An arithmetic expression e is represented as the value of an inductive type corresponding to a typed abstract syntax tree, namely an expression tree [7]. The nodes of an expression tree correspond to arithmetic expressions, including floating-point operations, integer operations, and some functions such as `nearbyint`.

The expression e can then be interpreted in several ways, two of which are relevant here. First, it can be interpreted as the floating-point number $\llbracket e \rrbracket_{\text{flt}}$ that would be obtained according to the IEEE-754 standard. Second, e can be interpreted as the value $\llbracket e \rrbracket_{\text{rnd}}$ obtained by performing all the operations on real numbers and rounding their results. For example, in the case of the floating-point addition, we have $\llbracket u + v \rrbracket_{\text{rnd}} = \circ(\llbracket u \rrbracket_{\text{rnd}} + \llbracket v \rrbracket_{\text{rnd}})$. In the case of integer operations, $\llbracket e \rrbracket_{\text{flt}}$ performs computations modulo a power of two, while $\llbracket e \rrbracket_{\text{rnd}}$ performs operations on unbounded integers, *e.g.*, $\llbracket u + v \rrbracket_{\text{rnd}} = \llbracket u \rrbracket_{\text{rnd}} + \llbracket v \rrbracket_{\text{rnd}}$. The first interpretation corresponds to the value actually computed by an implementation, and therefore the value on which we need to prove a correctness theorem. The second interpretation, however, is the one that is the more amenable to formal reasoning, as it is not susceptible to exceptional behaviors such as overflows.

There are two features of expression trees that are of interest to us. The first one is the support for let-binding operators, with binders represented by their de Bruijn indices, to express sharing between sub-expressions and to guide proofs. The second one is the availability of exact arithmetic operations, as they are commonly encountered in implementation of mathematical functions. As far as $\llbracket \cdot \rrbracket_{\text{flt}}$ is concerned, there is no difference in interpretation between exact and inexact operations over floating-point numbers; they are performed as mandated by the IEEE-754 standard. For $\llbracket \cdot \rrbracket_{\text{rnd}}$, exact arithmetic operations, however, are not rounded, which makes formal proofs, both manual and automatic, much simpler. This raises the concern of whether such a proof about $\llbracket e \rrbracket_{\text{rnd}}$ is meaningful, which Theorem 1 below will tackle.

Let us illustrate these two features on the example of the argument reduction of the Cody-Waite exponential [4], variants of which are still widely used in modern implementations, including in the code shown in Listing 1:

$$\begin{aligned} k &\leftarrow \text{nearbyint}(x \cdot C), \\ t &\leftarrow x - k \cdot c_1 - k \cdot c_2, \end{aligned}$$

with $c_1 + c_2 \simeq 1/C$ and $c_2 \ll c_1$. Below is the formulation of this argument reduction as an expression tree.


```

Let (NearbyInt (Op MUL (Var 0) (BinFl C)))
(Op SUB
  (OpExact SUB (Var 1) (OpExact MUL (Var 0) (BinFl c1)))
  (Op MUL (Var 0) (BinFl c2)))

```

Notice that both floating-point operations in $x - k \cdot c_1$ are annotated as exact operations by using the `OpExact` constructor. All the other operations are marked as potentially inexact (`Op` constructor). This gives the following value for $\llbracket t \rrbracket_{\text{rnd}}$:

$$\circ(x - k \cdot c_1 - \circ(k \cdot c_2)) \quad \text{with } k = \lceil \circ(x \cdot C) \rceil.$$

2.2 Relation between interpretations

As mentioned earlier, a correctness statement is about $\llbracket e \rrbracket_{\text{flt}}$, while a user only wants to have to deal with $\llbracket e \rrbracket_{\text{rnd}}$, as it is free of exceptional behaviors and contains fewer rounding operations. In order to bridge the gap between both interpretations, a predicate `WB` (for *well-behaved*) is defined recursively over expressions. For example, the proposition `WB(Op DIV u v)` is defined as

$$\text{WB}(u) \wedge \text{WB}(v) \wedge \llbracket v \rrbracket_{\text{rnd}} \neq 0 \wedge |\circ(\llbracket u \rrbracket_{\text{rnd}} / \llbracket v \rrbracket_{\text{rnd}})| \leq \Omega$$

with Ω the value of the largest finite floating-point number. In other words, for the floating-point division u/v to be well-behaved, it is sufficient that u and v are well-behaved, that the interpretation of v as a real number is non-zero, and that the division over real numbers, once rounded, does not overflow the floating-point format. The predicate `WB` is defined in a similar way for the other inexact operations over floating-point numbers. For exact operations, the formula contains an additional conjunct that states that the result is exactly representable. For example, the proposition `WB(OpExact ADD u v)` is defined as

$$\text{WB}(u) \wedge \text{WB}(v) \wedge \circ(\llbracket u \rrbracket_{\text{rnd}} + \llbracket v \rrbracket_{\text{rnd}}) = \llbracket u \rrbracket_{\text{rnd}} + \llbracket v \rrbracket_{\text{rnd}} \wedge |\llbracket u \rrbracket_{\text{rnd}} + \llbracket v \rrbracket_{\text{rnd}}| \leq \Omega.$$

The key result is that, if an expression e is well-behaved, then $\llbracket e \rrbracket_{\text{flt}}$ is a finite floating-point number and it represents the real number $\llbracket e \rrbracket_{\text{rnd}}$. This is expressed by the following theorem:

► **Theorem 1.** *Given an expression e , $\text{WB}(e) \Rightarrow \llbracket e \rrbracket_{\text{flt}} \text{ finite} \wedge \llbracket e \rrbracket_{\text{flt}} = \llbracket e \rrbracket_{\text{rnd}}$.*

When applying Theorem 1, the user is left with a subgoal `WB(e)`, which is painful to prove by hand. So, to ease the proof process, the framework proposes a proof strategy called `simplify_wb`, which tackles this subgoal by applying a procedure similar to `CoqInterval`'s `interval` strategy to every conjunct of `WB(e)` individually. In practice, one can expect all the conjuncts related to the absence of exceptional behaviors to be automatically proved. Conjuncts related to exact operations, *e.g.*, $\circ(\llbracket u \rrbracket_{\text{rnd}} + \llbracket v \rrbracket_{\text{rnd}}) = \llbracket u \rrbracket_{\text{rnd}} + \llbracket v \rrbracket_{\text{rnd}}$, are however out of the scope of `CoqInterval`. So, the user will have to prove them either manually or using a dedicated tool like `Gappa` [3, §4.3].

Note that, in order for `simplify_wb` to make use of the `interval` strategy of `CoqInterval`, the latter had to be enhanced with some support for rounding operators, as they appear in almost all the conjuncts of `WB(e)`. This support was based on the so-called *standard model* of floating-point arithmetic. (Section 3.2 will propose a better approach.) For instance, given an enclosure $u \in [\underline{u}; \bar{u}]$, `CoqInterval` would compute an enclosure of $\circ(u)$ as follows, assuming a *binary64* format:

$$\circ(u) \in [\underline{u} - \varepsilon; \bar{u} + \varepsilon] \quad \text{with } \varepsilon = \max(2^{-1075}, -2^{-53} \cdot \underline{u}, +2^{-53} \cdot \bar{u}). \quad (2)$$

3 Automated tools and rounding errors

Consider the code of Listing 1. To verify its correctness, we need to prove the following bound on the absolute error between the exponential function and its floating-point degree-5 polynomial approximation:

$$\forall t \in \mathbb{R}, |t| \leq 355 \cdot 2^{-16} \Rightarrow |1 + y - \exp t| \leq 11 \cdot 2^{-62} \quad \text{with } y = \circ(t \cdot \circ(p_1 + \circ(t \cdot \circ(p_2 + \dots))))). \quad (3)$$

The traditional methodology to prove such a bound is as follows [3, §6.2.3]. One would split the expression $1 + y - \exp t$ into two parts $e_1 + e_2$, with $e_1 = y - t \cdot (p_1 + t \cdot (p_2 + \dots))$ and $e_2 = (1 + t \cdot (p_1 + t \cdot (p_2 + \dots))) - \exp t$. On one hand, expression e_1 , which contains only arithmetic operations and rounding operators, can be bounded using the dedicated Gappa tool [3, §4.3]. On the other hand, expression e_2 , which contains no rounding operator, can be bounded using the rigorous polynomial approximations of CoqInterval [11]. Combining the proofs of both bounds gives the final result.

Since support for rounding operators has been added to CoqInterval so that `simplify_wb` could automatically prove conjuncts of WB [7], it should now be possible to perform this kind of proof directly, without any need for such algebraic manipulations nor the use of an external tool. Unfortunately, several issues arise when using the `interval` strategy on Equation (3). Indeed, it is slightly more involved than the usual conjuncts of WB.

First of all, both sides of the subtraction are strongly correlated, since the left-hand side $1 + y$ was chosen among the best possible floating-point approximations of the right-hand side $\exp t$. This means that naive interval arithmetic, as used in Equation (2) to define the enclosure of a rounding operator, will cause an overestimation of the final enclosure that is so large that it becomes useless for proving anything interesting. On this example, the strategy would only be able to prove that the error is bounded by 10^{-2} , very far from the expected bound of $11 \cdot 2^{-62}$. So, the first step is to define rigorous polynomial approximations for rounding operators (Section 3.1).

This is not sufficient though, as the strategy would only succeed in proving a bound of $24 \cdot 2^{-62}$, which is already quite good, but not sufficient to prove the correctness of the code of Listing 1. This overestimation is a consequence of using the standard model of floating-point arithmetic to derive Equation (2), as it is a bit too naive. So, the second step is to prove tighter bounds on rounding errors (Section 3.2).

3.1 Rigorous polynomial approximations

The correlation issue of naive interval arithmetic is well-known, and it is independent of rounding errors. In fact, even the interval evaluation of $(a + x) \cdot (b - x)$ would suffer from it, as $a + x$ and $b - x$ vary in opposite directions with respect to x . A first solution to this issue is to split the domain of x into smaller sub-intervals and to take the union of the enclosures of the whole expression on all these sub-intervals. This approach is very simple proof-wise, but it scales poorly computation-wise, so it should only be used as a last resort. A second solution is to compute enclosures whose bounds symbolically depend on x rather than being just numerical values. This approach scales better, but it requires a much larger formalization effort.

CoqInterval provides both approaches [11]. In particular, the second approach is implemented using rigorous polynomial approximations. Instead of just computing a single interval $[\underline{e}; \bar{e}]$ that encloses an expression $e(x)$ for any $x \in X$, it computes a polynomial P and an interval Δ such that, for any $x \in X$, we have $e(x) - P(x) \in \Delta$, which we denote by $e \in (P, \Delta)_X$. Those polynomial enclosures can then be composed. For example, if we have

$f \in (P_f, \Delta_f)_X$ and $g \in (P_g, \Delta_g)_X$, we also have $f + g \in (P_f + P_g, \Delta_f + \Delta_g)_X$. This makes it possible to compute the polynomial enclosure of an arbitrary expression, by induction on its structure.

Therefore, to benefit from the rigorous polynomial approximations of CoqInterval, we need to be able to compute a polynomial enclosure of $\circ(u)$, given an enclosure $u \in (P, \Delta)_X$. To do so, we rewrite $\circ(u(x))$ into the sum $[\circ(u(x)) - u(x)] + u(x)$. For the left-hand side, we use the degree-0 enclosure $\circ(u) - u \in (0, [-\varepsilon; \varepsilon])_X$ with ε computed as in Equation (2). Then, by adding the original enclosure $(P, \Delta)_X$, we get a polynomial enclosure of $\circ(u)$.

This change to CoqInterval was straightforward, but it has shifted the perspective on rounding operators in the library. Indeed, the original implementation, which was designed for `simplify_wb`, computed an enclosure of $\circ(u)$ given an enclosure of u . Then, the user could ask for an enclosure of $\circ(u) - u$, which would be correct but overestimated. The new implementation computes a tight enclosure of $\circ(u) - u$ from an enclosure of u , from which it derives an enclosure of $\circ(u)$. This change has been propagated up to the surface language, that is, CoqInterval now recognizes the expression $\circ(u) - u$ as an atomic error for an expression u rather than a subtraction between two sub-expressions involving u .

3.2 Tighter error bounds

By adding support for rounding operators, CoqInterval is now able to automatically prove Equation (3), but only if the rightmost bound is changed to $24 \cdot 2^{-62}$. It fails for any tighter bound, especially for $11 \cdot 2^{-62}$, which we need to prove the correctness of the implementation of Listing 1. As mentioned earlier, the issue comes from the simplicity of the standard model of floating-point arithmetic, which states that the absolute error between $\circ(u)$ and u is bounded by $2^{-53} \cdot |u|$, assuming that u is in the normal range. While this is sensibly true for values of u slightly larger than a power of two, this is off by a factor two for values of u that are slightly smaller than a power of two. A better model of floating-point errors is to bound the absolute error between $\circ(u)$ and u by $\frac{1}{2}\text{ulp}(u)$, where ulp denotes the *unit in the last place*, which is the distance between $|u|$ and its successor. In other words, given an enclosure $u \in [\underline{u}; \bar{u}]$, we have the following enclosure of the absolute error:

$$\circ(u) - u \in \left[-\frac{\varepsilon}{2}; \frac{\varepsilon}{2}\right] \quad \text{with } \varepsilon = \text{ulp}(\max(-\underline{u}, \bar{u})).$$

Not only is this new enclosure tighter, but it also makes the implementation and its proof more generic, as it separates the concerns about the target format and the rounding direction. Regarding the target format, one just has to choose the corresponding definition for ulp . As a consequence, CoqInterval now supports not only the floating-point formats of Flocq, but also its fixed-point formats. As for the rounding direction, it is a matter of choosing the enclosing interval: $[-\frac{\varepsilon}{2}; \frac{\varepsilon}{2}]$ for rounding to nearest, $[0; \varepsilon]$ for rounding toward $+\infty$, and so on.

Thanks to these improvements, the `interval` strategy can now directly prove Equation (3). This proof only takes a tenth of a second using degree-10 polynomials (default degree for `interval`). Note that the use of polynomial approximations, rather than the use of more naive variants of interval arithmetic, is critical for this proof, as can be experienced by reducing the degree. With degree 3, it takes about one second; with degree 2, it takes about one minute; and with degree 1, it does not seem to terminate.

4 Supported formats and expressions

Since the goal of our work is to formally prove the function shown in Listing 1, we need several new features that were missing from the earlier work on the Cody-Waite algorithm [7]. First of all, since our implementation relies on hardware support for both integer and floating-point

numbers, we need an interpretation of the expressions from Section 2 into the corresponding types (Section 4.1). Since the implementation also uses an array of pre-calculated values to reduce the degree of the polynomial approximation, the grammar of expressions has been extended with array accesses (Section 4.2). Finally, as mentioned earlier, the algorithm takes advantage of the inaccuracies and “flaws” of floating-point arithmetic to implement optimized versions of `nearbyint` and `int_of_float`. Hence, to ease the proof of this algorithm, we have added support for these optimized operations (Section 4.3).

4.1 Hardware operations

Similarly to $\llbracket e \rrbracket_{\text{flt}}$, we would like to define another interpretation $\llbracket e \rrbracket_{\text{prim}}$ which represents the computation of e using the hardware types provided by Coq’s standard library. The `PrimFloat` module offers support for hardware floating-point numbers [12], while the `PrimInt63` module offers support for OCaml’s 63-bit integers [5]. Both modules provide constants, basic operations ($+$, $-$, \times , $/$, etc.), comparisons ($=$, $<$, \leq), conversions, and some miscellaneous functions (e.g., floating-point predecessor and successor functions). They also provide axiomatized specifications for these hardware operations.

Since hardware floating-point numbers are just an instance of Flocq’s generic floating-point numbers, we have derived the following variant of Theorem 1:

► **Theorem 2.** *Given an expression e , $\text{WB}(e) \Rightarrow \llbracket e \rrbracket_{\text{prim}} \text{ finite} \wedge \llbracket e \rrbracket_{\text{prim}} = \llbracket e \rrbracket_{\text{rnd}}$.*

There are two things to note about the definition of $\llbracket e \rrbracket_{\text{prim}}$. First, not all operations can be performed directly on hardware types. Fused multiply-add (FMA), for example, is not yet provided by the `PrimFloat` module. Therefore, to complete the definition of $\llbracket e \rrbracket_{\text{prim}}$, we emulate these missing operations using the Flocq library (i.e., convert the operands to the formalized Flocq types, compute the result using Flocq’s operations, and convert it back to the hardware type).

Second, we have made our integers 32-bit wide, so that we can use Coq’s 63-bit hardware integers to compute $\llbracket e \rrbracket_{\text{prim}}$ while maintaining our ability to export verified algorithms as C programs. For 32-bit integer expressions, $\text{WB}(e)$ hence requires $\llbracket e \rrbracket_{\text{rnd}}$ to remain inside the $[-2^{31}; 2^{31} - 1]$ range.

4.2 Array accesses

The implementation of Listing 1 starts with an argument reduction that is very similar to Cody & Waite’s, but based on a slightly different identity:

$$\exp(x) = \exp\left(x - k \cdot \frac{\ln 2}{64}\right) \cdot 2^{k/64} \quad \text{with} \quad k = \left\lceil x \cdot \frac{64}{\ln 2} \right\rceil. \quad (4)$$

This makes the reduced argument much smaller, but it also means that the reconstruction is not a simple multiplication by an integer power of 2 anymore. To multiply by $2^{k/64}$ for some integer k , we first compute the Euclidean division of k by 64, in other words find k_q and k_r such that $k = k_q \cdot 64 + k_r$ and $0 \leq k_r \leq 63$. Since there are only finitely many different values of k_r , we pre-compute the floating-point number closest to $2^{k_r/64}$ for each value of k_r and store the results in a table `cst`. Therefore, to multiply by $2^{k/64}$, we first multiply by `cst.[k_r]` and then by 2^{k_q} (see Listing 1).

We have defined `cst` using the Coq standard library `PArray`, which provides persistent arrays [5]. To ease proofs, we have added a constructor `ArrayAcc` to the type of expressions to represent accesses to tables of constants. The constructor takes as argument an array a of hardware floating-point numbers and an integer expression i and is interpreted as follows:

$$\llbracket \text{ArrayAcc } a \ i \rrbracket_{\text{prim}} := a.\llbracket i \rrbracket_{\text{prim}}.$$

For an access to be well-behaved, we need the index to be well-behaved and smaller than the length of the array, and all the entries of the array to be finite floating-point numbers.

4.3 Macro-operations

As we can see in Equation (4), the argument reduction also requires the `nearbyint` function. This poses a problem as the latter is not provided by the `PrimFloat` module. Since we only need to compute the exponential on inputs in the $[-746; 710]$ range, we can use the following trick³ to compute the integer part:

$$\lceil f \rceil = \circ(\circ(f + 1.5 \cdot 2^{52}) - 1.5 \cdot 2^{52}). \quad (5)$$

Using the language of abstract expressions, we could simply represent this sequence of operations as `Op SUB (Op ADD f (BinF1 0x1.8p52)) (BinF1 0x1.8p52)`. However, that would not be very helpful in proofs because it leaves us the tedious work of showing that those operations behave as `nearbyint`. Instead, we want to treat those operations as if they were one single `nearbyint` operation. For this, we define a new constructor `FastNearbyint` in the language whose interpretation as a floating-point expression is the sequence of operations above, but whose interpretation as a rounded expression is the integer part:

$$\begin{aligned} \llbracket \text{FastNearbyint } e \rrbracket_{\text{flt/prim}} &:= \llbracket e \rrbracket_{\text{flt/prim}} \oplus 0\text{x}1.8\text{p}52 \ominus 0\text{x}1.8\text{p}52, \\ \llbracket \text{FastNearbyint } e \rrbracket_{\text{rnd}} &:= \lceil \llbracket e \rrbracket_{\text{rnd}} \rceil. \end{aligned}$$

Since these interpretations are no longer in one-to-one correspondence, proving Theorems 1 and 2 for these constructors required significantly more work on our part. This, however, saves the user from having to do the work themselves. Note that Equation (5) is only meaningful for inputs $|f| \leq 2^{51}$, so $\text{WB}(\text{FastNearbyint } e)$ contains a conjunct $|\llbracket e \rrbracket_{\text{rnd}}| \leq 2^{51}$.

The macro-operation we have just defined computes the integer part as a floating-point number, but the algorithm in Listing 1 also needs it as an integer. Hence, we define another constructor `FastNearbyintToInt` which extracts the mantissa⁴ after adding $1.5 \cdot 2^{52}$:

$$\lceil f \rceil_{\mathbb{Z}} = \text{mantissa}(\circ(f + 0\text{x}1.8\text{p}52)) - 3 \cdot 2^{51}.$$

5 Application: a state-of-the-art exponential

We now have all the ingredients to state and prove the correctness of the algorithm shown in Listing 1. It is stated as follows, with x the floating-point input, and with flb and fub the components of the pair computed by the algorithm:

► **Theorem 3.** *If x is finite, then $flb \leq \exp x \leq fub$.*

Proof of this theorem for large positive and negative values of x is straightforward, as those are the cases where the exponential either overflows or degenerates to 0. This section presents the methodology we have followed for proving the correctness theorem for $x \in [-745.13; 709.78]$.

For a given floating-point input x , to find an enclosure of $\exp x$, our algorithm performs only one approximation y , but then subtracts (resp. adds) an error term d to find the lower (resp. upper) bound of the enclosure. Correctness of the algorithm therefore relies on

³ If $|f| \leq 2^{51}$ then $f + 1.5 \cdot 2^{52}$ is between 2^{52} and 2^{53} with $\text{ulp}(2^{52}) = 1$, which means the result of the addition is rounded to the nearest integer.

⁴ For hardware numbers, we have implemented it as `normfr_mantissa (fst (frshiftexp f))`.

whether d is big enough to cancel out the inaccuracy of the approximation. For this reason, an essential step in our proof is to find some ε such that any choice of d with $|d| > \varepsilon$ makes $\circ(\circ(p_0 \cdot y) + d)$ an upper bound of $\exp(x - k \ln 2/64) - 2^{k_r/64}$ (and similarly for the lower bound). The value we have experimentally found for ε is characterized by the following lemma:

► **Lemma 4.** *For any finite floating-point number d such that $|d| \leq 2^{-52}$, we have*

$$\left| 2^{k_r/64} + \circ(\circ(p_0 \cdot y) + d) - (\exp(x - k_q \ln 2) + d) \right| < \varepsilon \simeq 1.14 \cdot 2^{-57}.$$

The proof of this lemma is too intricate to be comprehensively explained. Instead, we will illustrate our methodology on the parts that verify the argument reduction and the polynomial evaluation (Section 5.1). We will also show part of the proof for the reconstruction as it involves some unusual facts about floating-point arithmetic (Section 5.2).

5.1 Illustration of the methodology

Among other facts about the argument reduction, we need to prove that the computation of \mathbf{ki} causes no exceptional behaviors and that it is indeed an integer part equal to k despite its convoluted code. To do so in the Coq proof, we have defined an abstract expression \mathbf{ki}' whose interpretation in the hardware numbers – namely, $\llbracket \mathbf{ki}' \rrbracket_{\text{prim}}$ – is the value stored in \mathbf{ki} , and whose interpretation in the rounded real numbers – namely, $\llbracket \mathbf{ki}' \rrbracket_{\text{rnd}}$ – is k . Then we can use Theorem 2 to transform a goal about \mathbf{ki} into a goal about k .

In practice, we not only want to transform the goal, but we also want to assert some property on the transformed subexpression. Hence, we have implemented a strategy `assert_float` which takes as argument a predicate Q , looks for an expression of the form $\llbracket e \rrbracket_{\text{prim}}$, and applies the following corollary of Theorem 2:

$$\text{WB}(e) \implies Q(\llbracket e \rrbracket_{\text{rnd}}) \implies (\forall x, x = \llbracket e \rrbracket_{\text{rnd}} \wedge Q(x) \implies G(x)) \implies G(\llbracket e \rrbracket_{\text{prim}}).$$

The strategy also invokes `simplify_wb` to discharge as many conjuncts of $\text{WB}(e)$ as possible. When using this strategy, the Coq proof usually looks as follows:

```
set (ki' := FastNearbyIntToInt (Op MUL (Var 0) InvLog2_64)).
change (normfr_mantissa _ - _)
  with (evalPrim ki' [:x:]). (* Var 0 is mapped to x *)
assert_float (fun ki => -68736 <= ki <= 65536).
{ ... proof of the assertion ... }
```

We have used the `assert_float` strategy 8 times in the proof. Here is another example of its usage to state the main property about the reduced argument t , which contains exact operations just like in the original Cody-Waite algorithm (see Section 2.1):

```
set (t' := Op SUB (OpExact SUB (Var 1) ...) ...).
change (x - _ - _) with (evalPrim t' [:k, x:]).
assert_float (fun t => abs t <= 355 / 65536
  /\ abs (t - (x - k * ln 2)) <= 65537 * pow2 (-77)).
```

Contrary to the other uses of `assert_float` in the proof, `simplify_wb` is not able to completely discharge the subgoal $\text{WB}(t)$. So we have to manually prove the remaining conjuncts, which are the proof obligations of exact operations:

$$\circ(k \cdot c_1) = k \cdot c_1 \quad \wedge \quad \circ(x - k \cdot c_1) = x - k \cdot c_1.$$

The proof of both equalities relies on bit-counting reasoning, which Gappa is specifically designed for [3, §4.3.4]. But to avoid introducing a dependency over Gappa just to prove these two conjuncts, we have performed this reasoning by hand.

As a last illustration, let us consider Equation (3), which bounds the error caused by both the polynomial approximation and its floating-point evaluation. The corresponding proof script looks as follows. For the sake of readability, we have removed a few administrative steps (*e.g.*, unfolding of definitions) from the script.

```
change (Papprox t') with (evalPrim g0 [:t':]).
assert_float (fun y => abs y <= 0.0055
              /\ abs (1 + y - exp t) <= 11 * pow2 (-62)).
{ split.
- interval.
- interval with (i_taylor t, i_bisect t, i_prec 80). }
```

Thanks to the automation provided by `assert_float` and `interval`, in just a few lines, we have proved that none of the floating-point operations had any exceptional behavior, that the image of the floating-point function was bounded, and more importantly, that its error was bounded too. More generally, if a user needed to prove the correctness of a simple floating-point implementation with no intricate argument reduction (*e.g.*, a piece-wise polynomial approximation), that would be the whole of the script.

5.2 Correctness of reconstruction

At this point in the proof, thanks to Lemma 4, we know $2^{k_r/64} + y_\ell \leq \exp x \cdot 2^{-k_q} \leq 2^{k_r/64} + y_u$, with y_ℓ and y_u some intermediate floating-point results. To complete the proof, we need to deduce the following enclosure:

$$flb = \text{pred}(\circ(\circ(p_0 + y_\ell) \cdot 2^{k_q})) \leq \exp x \leq \text{succ}(\circ(\circ(p_0 + y_u) \cdot 2^{k_q})) = fub.$$

To do so, we want to factor out the multiplication by 2^{k_q} , but the possibility that the result might fall into the subnormal range makes this factorization impossible. So we have proved the following lemma:

► **Lemma 5.** *Let y be a binary64 floating-point number greater than 2^{-1021} . Then, for any integer k , $\text{pred}(\circ(y \cdot 2^k)) \leq \text{pred}(y) \cdot 2^k$ and $\text{succ}(\circ(y \cdot 2^k)) \geq \text{succ}(y) \cdot 2^k$.*

By transitivity, we are thus left to prove the following inequalities:

$$\text{pred}(\circ(p_0 + y_\ell)) \leq 2^{k_r/64} + y_\ell \quad \wedge \quad 2^{k_r/64} + y_u \leq \text{succ}(\circ(p_0 + y_u)).$$

These inequalities hold because the predecessor and successor functions are enough to compensate both the error between p_0 and $2^{k_r/64}$ and the rounding error of the final addition. Indeed, there are two cases, depending on the value of k_r :

- If $k_r = 0$, then $p_0 = 1$. So, the first inequality reduces to $\text{pred}(\circ(1 + y_\ell)) \leq 1 + y_\ell$, which is a general property of the predecessor function. Proof of the upper bound is similar.
- If $k_r \neq 0$, then we have $1.01 < p_0 < 1.99$. Therefore, $1.001 < \circ(p_0 + y_\ell) < 1.999$ and hence $\text{pred}(\circ(p_0 + y_\ell)) = \circ(p_0 + y_\ell) - 2^{-52}$. Moreover, we have $|p_0 - 2^{k_r/64}| \leq 2^{-53}$. Similarly, $|\circ(p_0 + y_\ell) - (p_0 + y_\ell)| \leq 2^{-53}$. As a consequence,

$$\begin{aligned} \text{pred}(\circ(p_0 + y_\ell)) &= 2^{k_r/64} + y_\ell + (p_0 - 2^{k_r/64}) + (\circ(p_0 + y_\ell) - (p_0 + y_\ell)) - 2^{-52} \\ &\leq 2^{k_r/64} + y_\ell + 2^{-53} + 2^{-53} - 2^{-52} = 2^{k_r/64} + y_\ell \end{aligned}$$

which completes the proof for the lower bound. Proof of the upper bound is similar.

6 Related work

While there had been some earlier works to formalize hardware arithmetic operators [18], formal verification of mathematical libraries really started with the impressive work by John Harrison. Among other things, he used the HOL Light system to prove the correctness of a *binary32* approximation of the exponential function which presents many similarities with our own algorithm [8]. But being a *binary32* function, it could nowadays be validated by sheer exhaustive testing. So, perhaps more interesting is Harrison’s subsequent work on the formal verification of the implementation of sin and cos for IA-64 architectures, as it sets the bar even higher [9]. Indeed, these approximations perform 80-bit floating-point computations and are accurate to 0.574 ulp for inputs smaller than 2^{63} . They use an intricate argument reduction: first a pre-reduction, followed by a 3-term Cody-Waite reduction, resulting in a double-*binary80* reduced argument. Then a degree-17 polynomial is evaluated, followed by a simple reconstruction of the result so as to take the lower part of the reduced argument into account. During both the argument reduction and the reconstruction, several floating-point operations are actually exact and need to be considered as such, in order to be able to prove anything interesting about the result. Our methodology could be used to automate various parts of this proof, but the representation of the reduced argument as a non-evaluated sum of two floating-point numbers would presumably warrant adding a few more macro-operations to our expression language, *e.g.*, a FastTwoSum operator [15, §1.3].

A more recent work is the large verification using the Coq proof assistant of the power function of the CORE-MATH library by Laurence Rideau and Laurent Théry [10]. This includes the correctness of an exponential function whose implementation shares some similarities with ours, but it is a lot more subtle, since both input and output are double-*binary64* numbers. Their formalization, however, ignores the issue of exceptional behaviors and just assumes that numbers can be arbitrarily large, as is traditionally the case in pen-and-paper proofs. Again, our methodology could help transition to a complete proof, especially since they are already making heavy use of CoqInterval.

Regarding the use of hardware floating-point numbers in the Coq proof assistant, Érik Martin-Dorel and Pierre Roux have implemented and verified a checker for semi-definite positive matrices [12]. The algorithm performs a Choleski decomposition using floating-point arithmetic on a slightly perturbed input matrix. The correctness theorem states that, if this decomposition succeeds, then a Choleski decomposition using exact arithmetic would have succeeded on the original input matrix, which guarantees that it was indeed semi-definite positive. The perturbation, and hence the correctness proof, depends on the ability to compute a bound on the rounding error of the floating-point decomposition [16]. Our approach would have been of little help for that use case, as the algorithm and the error bound highly depend on the dimension of the matrix.

Regarding the automation of proofs of bounds on rounding errors in a proof assistant, one can cite the FPTaylor tool, which can generate proofs for HOL Light [19]. Given a floating-point expression, it computes an affine form that encloses it, using elementary rounding errors as variables of the affine form. The strength of that tool is that the coefficients of the affine form are kept as symbolic expressions rather than intervals. This approach separates the concerns between the global optimizer used for computing enclosures and the formalization of affine forms for floating-point arithmetic. Indeed, enclosures of the symbolic expressions are only needed when they occur in terms of order 2 or more, as these terms cannot be represented as part of the affine form. Therefore, the verification of these enclosures can be done in a rather naive way, since they are only used for higher-order error term and thus do

not have to be tight. It should be noted that FPTaylor supports both the standard model of floating-point arithmetic and a tighter model (see Section 3.2), but it can only generate proofs for the former. Moreover, the global optimizer used to compute the enclosures in FPTaylor is not the same as the one used to verify them in HOL Light, which might cause difficulties if the latter procedure is not strong enough or too slow.

A similar tool is PRECiSA, which targets the PVS proof assistant [13]. As with FPTaylor, errors are kept as symbolic expressions, and a global optimizer is used to compute their enclosures. Higher-order error terms, however, are not eliminated as computations progress, which might cause some performance issues compared to FPTaylor. PRECiSA, however, uses a tight formal model of floating-point errors, and the tool can detect exact subtractions (Sterbenz' lemma). Moreover, it supports conditional expressions, including the cases where rounding causes a different branch to be taken.

Finally, one should mention the VCFLOAT2 tool, which targets the Coq proof assistant [1]. As with the previous two tools, the error is kept as a symbolic expression. Before being fed to a global optimizer (namely CoqInterval), this expression is first simplified by expanding it through distributivity and discarding the sub-expressions that cancel. This expansion might cause some performance issues, due to combinatorial explosion. A user-provided threshold is used to further discard negligible terms, at the expense of a potentially worse error bound. It can also use a technique similar to Gappa to reduce the correlation between sub-expressions, and thus improve the tightness of the computed enclosures. The tool uses the standard model of floating-point errors, but the user can annotate operations that are supposed to be exact and the tool will verify that the conditions hold (Sterbenz' lemma). Moreover, the tool supports user-defined operations, which means that it can easily be extended with double-word arithmetic, as long as the user has formalized it beforehand.

7 Conclusion

In this article, we have presented a floating-point approximation of the exponential function, its mechanized proof of correctness, and the tools we have developed to ease the verification work. One peculiarity of this work is that the verified code is not just modeled using the Coq proof assistant, it can actually run in the logic of the system and therefore be used to perform proofs by computations. Indeed, the correctness theorem tells how the result of the approximation can be used as a lower/upper bound of the mathematical exponential. The specification of the code of Listing 1 and its correctness proof take about 600 lines of Coq script;⁵ Lemma 5 is about 130 lines; extending the proof of Theorem 1 to support macro-operations and arrays takes about 500 lines; the tighter bounds on rounding errors take about 200 lines. This work was integrated in release 4.10.0 of CoqInterval.

7.1 Integration to CoqInterval and performances

As explained in the introduction, the CoqInterval library provides an interval extension of the exponential function that can use the floating-point unit of the processor to speed up proof checking [12]. Its implementation, however, is based on a truncated power series, which is effective but rather naive, compared with the implementations that can be found in mathematical libraries targeting hardware floating-point formats. We have thus plugged our verified implementation in place of the original one. Consider the following Coq script.

⁵ https://gitlab.inria.fr/coqinterval/interval/-/blob/interval-4.11.0/src/Interval/Float_full_primfloat.v

```
Goal forall x, 10 <= x <= 11 -> Rabs (exp x - exp x) <= 0x1p-6.
Proof. intros x Hx. interval with (i_bisect x, i_depth 30). Qed.
```

It states that, for any real number x between 10 and 11, the difference between $\exp x$ (mathematical exponential) and itself is less than 2^{-6} . From a mathematical point of view, this statement is useless, since it could be trivially proved by rewriting $\exp x - \exp x$ to zero, but it is a good way to exercise the computations performed by CoqInterval. Indeed, the way the `interval` strategy is invoked, it will not try to use anything fancier than naive interval arithmetic. As a consequence, because $\exp x$ is strongly correlated with itself, the formal proof generated by the tactic ends up considering around 6.7 million sub-intervals of the input enclosure $x \in [10; 11]$ (and as many interval evaluations of the exponential function).

Using the original implementation, this computationally intensive proof takes about 160 seconds to be checked by the Coq proof assistant on an Intel 13th-generation 4GHz processor. With the implementation verified in this work, the proof is checked in less than 8 seconds. Taking the average of three runs, the speedup is $20.5\times$. Since the argument reduction of the original implementation is more costly the further away from zero the input is, the speedup can grow even larger, up to $24\times$.

As for the accuracy of the new implementation, one can get an intuitive feel of it by considering the distance between the bounds of the output interval. Ideally, it should be one ulp (except for the input 0), meaning that the bounds of the interval should be consecutive floating-point numbers. This property, called *correct rounding* [14, §12.3], is still an open research question for floating-point formats larger than *binary32* and completely out of reach of a formal proof, as of today. So, the best we can hope to achieve is a distance of up to two ulps, that is, one component is optimal, while the other is off-by-one. In the code shown in Listing 1, if the constant d was zero, this would be the case. As it is not quite zero here, when $\exp x$ is close to the midpoint between two consecutive floating-point numbers, the distance might end up being three ulps. The proportion of inputs that cause a 3-ulp interval output is roughly $d \cdot 2^{51} \simeq 1/60$.

7.2 Real-life performances

Being able to perform about one million faithful interval evaluations of exponential per second inside the logic of Coq is impressive, but it is nowhere near the actual throughput of the floating-point unit of the processor. Indeed, disregarding any concern about the guaranteed accuracy of a mathematical library, one should expect a state-of-the-art implementation to take 25–50 cycles to compute two floating-point approximations of exponential⁶ (and thus one interval enclosure), so about $100\times$ faster than what we currently achieve in the logic of Coq. There are several reasons for the remaining gap. First of all, the code of our implementation is not directly run by the processor, but interpreted by a virtual machine. Second, this bytecode interpreter boxes floating-point numbers, and thus performs a large amount of memory allocations. Third, while our code only performs computations on values, the interpreter still needs to account for the possibility of open terms (*e.g.*, free variables) appearing as operands to the floating-point computations.

The first issue can be worked around by using the `native_compute` machinery of the Coq system, which compiles the code using the OCaml compiler and then executes it directly [2]. This machinery also partially avoids the second issue, since the compiler can optimize away

⁶ <https://core-math.gitlabpages.inria.fr/>

■ **Listing 2** Floating-point exponential in OCaml.

```

let fexp x =
  if x < -0x1.74385446d71c4p9 then 0. else
  if x > 0x1.62e42fefa39efp9 then infinity else
  if x <> x then nan else
  let k' = x *. invLog2_64 +. 0x1.8p52 in
  ...
  let p0 = cst.(ki land 63) in
  ldexp (p0 +. p0 *. y) (ki asr 6)

```

the boxing of some intermediate floating-point results. But the third issue is still present and makes it hard to avoid pessimization in the generated code. As a consequence, this only improves proof checking by a factor $3\times$ to $4\times$ for the longer proofs.

To get a better feel of the actual performances of our implementation, we can instead implement the function directly in OCaml, as shown in Listing 2. This is roughly the same code as Listing 1, except that the original last three lines, which were computing an enclosure of $\exp x$, have been replaced by a single floating-point value: `ldexp (p0 +. p0 *. y) (ki asr 6)`. Accordingly, the first few lines return a single value for the exceptional cases. The code is run on about $1.5 \cdot 10^9$ inputs uniformly distributed among those that lead to a finite output. Compiling the code with OCaml 5.1.1, we get that the floating-point exponential from the GNU C Library is about $1.45\times$ faster than our implementation.

Even if the GNU C Library has been heavily tuned, this is still a rather large gap. Part of the reason is its use of the FMA operation. This ternary operation computes $o(x \cdot y + z)$ at once, which halves the number of operations performed during the argument reduction and the polynomial evaluation. Modifying our code accordingly, this reduces its slowdown to $1.32\times$. When translating the code to C and compiling it with GCC, the slowdown is brought down to $1.24\times$. Obviously, using FMAs in place of multiplications and additions invalidates the correctness proof, since they do not compute the same values (notice the lack of rounding operator around the product). Fortunately, the proof can be easily adapted. Indeed, exact operations during the argument reduction are still exact when performed with an FMA, and having a more accurate polynomial evaluation only makes the proof simpler. Note that, while our framework supports reasoning about the FMA operation, it is not one of the native floating-point operations provided by the Coq system, so it cannot be used to speed up the implementation of CoqInterval. One would instead have to use larger tables, as does the GNU C Library, so as to reduce the degree of the polynomial approximation.

7.3 Future works

First, it should be noted that, while the GNU C Library does not implement correct rounding either, it is nonetheless slightly more accurate than our implementation. In about 20% of cases, the code of Listing 2 returns a floating-point result that is off by one, while for the GNU C Library, the probability is 10^{-5} . In the context of CoqInterval, this hardly matters, since we want to compute an enclosure of the mathematical result rather than the nearest floating-point number. But for a mathematical library, people might prefer a code that is experimentally a bit more accurate to a code whose correctness has been formally verified. Most of the inaccuracy comes from the factor `p0`. There are two ways to improve it, both of which require adding a new table along `cst`. In the first approach, the new table contains the error on `p0`, which can then be reintroduced in the computation. In the second approach, the new table tells how to shift the input, such that the error on `p0` becomes negligible.

A natural extension of this work is to convert all the other mathematical functions of CoqInterval to use some state-of-the-art implementation when hardware floating-point numbers are used as interval bounds. For functions such as log and arctan, our approach should work without difficulty, as they are quite similar to exp. For trigonometric functions such as sin and cos, the situation is slightly different. First of all, they are not monotone, so considering the lower and upper bounds of the input interval separately might be counter-productive; it might be better to perform a simultaneous argument reduction on both bounds. Second, the Cody-Waite approach to argument reduction does not scale well to extremely large inputs, while some other algorithms for argument reduction take advantage of the periodicity of the trigonometric functions [14, §11.4].

References

- 1 Andrew Appel and Ariel Kellison. VCFloat2: Floating-point error analysis in Coq. In *13th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 14–29, London, United Kingdom, 2024. doi:10.1145/3636501.3636953.
- 2 Mathieu Boespflug, Maxime Dénès, and Benjamin Grégoire. Full reduction at full throttle. In *1st International Conference on Certified Programs and Proofs*, pages 362–377, Kenting, Taiwan, December 2011. doi:10.1007/978-3-642-25379-9_26.
- 3 Sylvie Boldo and Guillaume Melquiond. *Computer Arithmetic and Formal Proofs*. ISTE Press – Elsevier, 2017.
- 4 William J. Cody, Jr. and William Waite. *Software Manual for the Elementary Functions*. Prentice-Hall, Englewood Cliffs, NJ, 1980.
- 5 Maxime Dénès. Towards primitive data types for Coq 63-bits integers and persistent arrays. In *5th Coq Workshop*, Rennes, France, July 2013. URL: https://coq.inria.fr/files/coq5_submission_2.pdf.
- 6 Paul Geneau de Lamarlière and Guillaume Melquiond. CoqInterval. Software, version 4.10.0., swhId: swh:1:dir:78da3e6e98b7ef018180119255ce1e10a048cc88 (visited on 2024-08-22). URL: <https://gitlab.inria.fr/coqinterval/interval.git>.
- 7 Paul Geneau de Lamarlière, Guillaume Melquiond, and Florian Faissole. Slimmer formal proofs for mathematical libraries. In Theo Drane and Anastasia Volkova, editors, *30th IEEE International Symposium on Computer Arithmetic*, Portland, OR, USA, September 2023. doi:10.1109/ARITH58626.2023.00026.
- 8 John Harrison. Floating-point verification in HOL Light: The exponential function. Technical Report 428, University of Cambridge Computer Laboratory, 1997.
- 9 John Harrison. Formal verification of floating point trigonometric functions. In Warren A. Hunt and Steven D. Johnson, editors, *3rd International Conference on Formal Methods in Computer-Aided Design*, volume 1954 of *Lecture Notes in Computer Science*, pages 217–233, 2000.
- 10 Tom Hubrecht, Claude-Pierre Jeannerod, Paul Zimmermann, Laurence Rideau, and Laurent Théry. Towards a correctly-rounded and fast power function in binary64 arithmetic, 2024. URL: <https://inria.hal.science/hal-04159652>.
- 11 Érik Martin-Dorel and Guillaume Melquiond. Proving tight bounds on univariate expressions with elementary functions in Coq. *Journal of Automated Reasoning*, 57(3):187–217, 2016. doi:10.1007/s10817-015-9350-4.
- 12 Érik Martin-Dorel, Guillaume Melquiond, and Pierre Roux. Enabling floating-point arithmetic in the Coq proof assistant. *Journal of Automated Reasoning*, 67, 2023. doi:10.1007/s10817-023-09679-x.
- 13 Mariano Moscato, Laura Titolo, Aaron Dutle, and César A. Muñoz. Automatic estimation of verified floating-point round-off errors via static analysis. In Stefano Tonetta, Erwin Schoitsch, and Friedemann Bitsch, editors, *36th International Conference on Computer Safety, Reliability*,

- and Security*, volume 10488 of *Lecture Notes in Computer Science*, pages 213–229, Trento, Italy, 2017. doi:10.1007/978-3-319-66266-4_14.
- 14 Jean-Michel Muller. *Elementary Functions, Algorithms and Implementation*. Birkhäuser, Boston, MA, 3rd edition, 2016. doi:10.1007/978-1-4899-7983-4.
 - 15 Jean-Michel Muller, Nicolas Brunie, Florent de Dinechin, Claude-Pierre Jeannerod, Mioara Joldes, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, and Serge Torres. *Handbook of Floating-Point Arithmetic*. Birkhäuser Basel, 2nd edition, 2018. doi:10.1007/978-3-319-76526-6.
 - 16 Pierre Roux. Formal proofs of rounding error bounds – with application to an automatic positive definiteness check. *Journal of Automated Reasoning*, 57(2):135–156, 2016. doi:10.1007/s10817-015-9339-z.
 - 17 Siegfried M. Rump. Verification methods: Rigorous results using floating-point arithmetic. *Acta Numerica*, 19:287–449, May 2010. doi:10.1017/S096249291000005X.
 - 18 David M. Russinoff. *Formal Verification of Floating-Point Hardware Design*. Springer Cham, 2nd edition, 2022. doi:10.1007/978-3-030-87181-9.
 - 19 Alexey Solovyev, Marek S. Baranowski, Ian Briggs, Charles Jacobsen, Zvonimir Rakamarić, and Ganesh Gopalakrishnan. Rigorous estimation of floating-point round-off errors with symbolic Taylor expansions. *ACM Transactions on Programming Languages and Systems*, 41(1), December 2018. doi:10.1145/3230733.

Typed Compositional Quantum Computation with Lenses

Jacques Garrigue ✉ 🏠 

Nagoya University, Japan

Takafumi Saikawa ✉ 

Nagoya University, Japan

Abstract

We propose a type-theoretic framework for describing and proving properties of quantum computations, in particular those presented as quantum circuits. Our proposal is based on an observation that, in the polymorphic type system of COQ, currying on quantum states allows one to apply quantum gates directly inside a complex circuit. By introducing a discrete notion of lens to control this currying, we are further able to separate the combinatorics of the circuit structure from the computational content of gates. We apply our development to define quantum circuits recursively from the bottom up, and prove their correctness compositionally.

2012 ACM Subject Classification Theory of computation → Program verification; Theory of computation → Quantum computation theory

Keywords and phrases quantum programming, semantics, lens, currying, Coq, MathComp

Digital Object Identifier 10.4230/LIPIcs.ITP.2024.15

Supplementary Material *Software:* <https://github.com/t6s/qecc> [9]
archived at `swh:1:dir:d4d158675180ee276e730bd7f67a9122a6472eb3`

Funding This research was partially supported by JSPS KAKENHI grant No. JP22H00520 and MEXT Q-LEAP grant No. JPMXS0120319794.

1 Introduction

Quantum computation is a theory of computation whose unit of information is the states of a quantum particle, called a quantum bit. A quantum bit is unlike a classical bit in that the former may retain many values at the same time, albeit they ultimately can only be observed as probabilities, while the latter has a single value. This possibility of a multitude of values is preserved by pure quantum computation, and destroyed by a measurement of the probability.

These properties of quantum bits and computation are commonly modelled in terms of unitary transformations in a Hilbert space [19]. Such a transformation is constructed by composing both sequentially and parallelly various simple transformations called quantum gates.

Many works have been built to allow proving quantum algorithms in such settings [15, 18, 20], or more abstractly using string diagrams representing computations in a symmetric monoidal category [5]. We investigate whether some type-theoretic insights could help in describing and proving properties of quantum computations, in particular those denoted by so-called quantum circuits.

Our main goal is to reach *compositionality* inside a semantical representation of computations. We wish it both at the level of definitions and proofs, with as little overhead as possible.

Definitional compositionality means that it should be possible to turn any (pure) quantum circuit into an abstract component, which can be instantiated repeatedly in various larger circuits.



© Jacques Garrigue and Takafumi Saikawa;
licensed under Creative Commons License CC-BY 4.0

15th International Conference on Interactive Theorem Proving (ITP 2024).

Editors: Yves Bertot, Temur Kutsia, and Michael Norrish; Article No. 15; pp. 15:1–15:18

Leibniz International Proceedings in Informatics



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

Proof compositionality means that the proof of functional properties about (pure) quantum circuits should be statable as a generic lemma about the corresponding abstract component, so that one can build proofs of a large circuit by applying this lemma to instances of the component, without having to unfold the concrete definition of the component during the proof.

Abstraction overhead refers to the extra steps required for abstraction and instantiation, both in definitions and proofs.

The approach we have designed represents circuits as linear transformations, and reaches the above goals by cleanly separating the complex linear algebra in computation from the combinatorics of the wiring, using a combinatorial notion of lens. Compared to more abstract approaches, such as the ZX-calculus [4], we are directly working on an explicit representation of states, but we are still able to prove properties in a scalable way that does not rely on automation, as one can compose circuits without adding complexity to the proof.

Our proposal combines several components, which are all represented using dependent and polymorphic types in COQ. *Finite functions* over n -tuples of bits can encode a n -qubit quantum state. *Lenses* are injections between sets of indices, which can be used to describe the wiring of quantum circuits in a compositional way. They are related to the lenses used for view-update in programming languages and databases [7]. *Currying* of functions representing states, along a lens, provides a direct representation of tensor products. *Polymorphism* suffices to correctly apply transformations to curried states. We need this polymorphism to behave uniformly, which is equivalent to morphisms being natural transformations.

Using these components, we were able to provide a full account of pure quantum circuits in COQ, on top of the MATHCOMP library, proving properties from the ground up. We were also able to prove a number of examples, such as the correctness of Shor coding [17] (formalized for the first time, albeit only for an error-free channel at this point), the Greenberger-Horne-Zeilinger (GHZ) state preparation [10], and the reversed list circuit [20].

Our development is available online [9].

The plan of this paper is as follows. In Section 2, we provide a short introduction to quantum states and circuits. In Section 3, we define lenses. In Section 4, we provide the mathematical definition of focusing of a circuit through a lens. In Sections 5 and 6, we explain the COQ definitions of gates and their composition. In Section 7, we introduce some lemmas used in proof idioms that we apply to examples in Section 8. In Section 9, we define noncommutative and commutative monoids of sequential and parallel compositions of gates. We present related works in Section 10 before concluding.

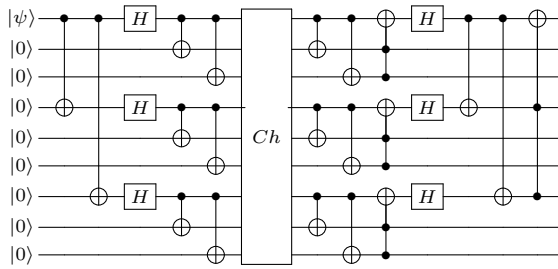
2 Quantum circuits and unitary semantics

In this section, we present basic notions from linear algebra to describe the unitary model of quantum computation, and how they appear in a quantum circuit diagram.

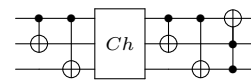
2.1 Quantum states

Let us first recall that pure classical computation can be seen as a sequence of boolean functions acting on an array of bits of type 2^n for some n . Similarly, pure quantum computation is modeled, in terms of linear algebra, as a sequence of unitary transformations that act on a quantum state of type \mathbb{C}^{2^n} .

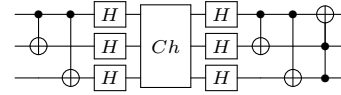
A quantum bit (or *qubit*) is the most basic unit of data in quantum computation. We regard it as a variable of type \mathbb{C}^2 and each vector of norm 1 is considered to be a state of the qubit. \mathbb{C}^2 has a standard basis $(1, 0), (0, 1)$, which we denote in the context of quantum



■ **Figure 1** Shor's 9-qubit code.



■ **Figure 2** Bit-flip code.



■ **Figure 3** Sign-flip code.

programming $|0\rangle, |1\rangle$, indicating that the state of the qubit is 0 and 1 respectively. Regarding \mathbb{C}^2 as the function space $[2] \rightarrow \mathbb{C}$, where $[n]$ stands for $\{0, \dots, n - 1\}$, we can express the standard basis in the form of functions

$$|0\rangle := x \mapsto \begin{cases} 1 & \text{if } x = 0 \\ 0 & \text{otherwise} \end{cases} \quad |1\rangle := x \mapsto \begin{cases} 1 & \text{if } x = 1 \\ 0 & \text{otherwise} \end{cases}$$

States other than basis states are linear combinations, which we call *superpositions*. The state of a qubit is mapped to a classical bit by an operation called *measurement*, which probabilistically results in values 0 or 1. The measurement of a state in superposition $a|0\rangle + b|1\rangle$ results in 0 with probability $|a|^2$ and 1 with probability $|b|^2$.

Those definitions naturally extend to n -ary quantum states. The basis states for n qubits are functions

$$|i_1 i_2 \dots i_n\rangle := (x : [2]^n) \mapsto \begin{cases} 1 & \text{if } x = (i_1, i_2, \dots, i_n) \\ 0 & \text{otherwise} \end{cases}$$

States other than basis states are again superpositions, which are linear combinations of norm 1. In other words, a state is represented by a function of type \mathbb{C}^{2^n} , besides the condition on its norm. We hereafter regard this type as the space of states. This type can also be identified with the n -ary tensor power $(\mathbb{C}^2)^{\otimes n}$ of \mathbb{C}^2 , a usual presentation of states in textbooks.

Similarly to the unary case, a measurement of an n -ary quantum state $\sum_{i \in [2]^n} c_i |i_1 i_2 \dots i_n\rangle$ results in an array of classical bits $i = (i_1, i_2, \dots, i_n)$ with probability $|c_i|^2$.

2.2 Unitary transformations

We adopt the traditional view that pure quantum computation amounts to applying unitary transformations to a quantum state. A unitary transformation is a linear function from a vector space to itself that preserves the inner product of any two vectors, that is, $\langle U(a) | U(b) \rangle$ is equal to $\langle a | b \rangle$ for any unitary U and vectors a and b , if we denote the inner product by $\langle a | b \rangle$. Since the norm of a is defined to be $\sqrt{\langle a | a \rangle}$, a unitary also preserves the norm condition of quantum states.

2.3 Quantum circuits

In the same way that classical computation can be expressed by an electronic circuit comprised of boolean gates (AND, OR, etc.), quantum computation is also conveniently presented as a circuit with quantum gates that represent primitive unitary transformations. More generally, a quantum circuit may contain nonunitary operations such as measurement, but we restrict ourselves to pure quantum circuits that contain none of them.

15:4 Typed Compositional Quantum Computation with Lenses

A quantum circuit is a concrete representation of quantum computation, drawn as n parallel wires with quantum gates and other larger subcircuits being placed over those wires. A quantum state is input from the left end of a circuit, transformed by gates and subcircuits on the corresponding wires, and output from the right end. As an example, we show the Shor's 9-qubit error correction code (Figure 1) and its subcomponents (Figures 2 and 3).

The primitive operations in a quantum circuit are quantum gates. In the Shor's code, three kinds of gates appear, namely Hadamard $\text{---}\boxed{H}\text{---}$, Controlled Not (CNOT) $\text{---}\overset{\bullet}{\oplus}\text{---}$, and Toffoli $\text{---}\overset{\bullet}{\oplus}\text{---}$. The large box \boxed{Ch} denotes an arbitrary unitary transformation modelling a possibly erroneous channel. The gates placed to the left of \boxed{Ch} implement the encoder algorithm of the code, and those to the right the decoder. The unitary operations denoted by these gates can be expressed as matrices with respect to the lexicographically ordered standard basis (e.g. $|00\rangle, |01\rangle, |10\rangle, |11\rangle$ for two qubits):

$$\text{---}\boxed{H}\text{---} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \quad \text{---}\overset{\bullet}{\oplus}\text{---} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \quad \text{---}\overset{\bullet}{\oplus}\text{---} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

A gate composed in a circuit is represented by a matrix by, first taking the Kronecker product with identity matrices corresponding to irrelevant wires, and second sandwiching it with the matrices that represent the action of a permutation on the index of tensors to reorder the input and output wires. For example, to describe the leftmost CNOT gate in the Shor's code, we first *pad* (append) seven wires to CNOT by taking the Kronecker product with $I_{2^7} = I_{128}$ and apply the permutation (24) to move \oplus from the second wire to the fourth wire. The resulting matrix is:

$$U_{2^9}((42)) \begin{bmatrix} I_{128} & 0 & 0 & 0 \\ 0 & I_{128} & 0 & 0 \\ 0 & 0 & 0 & I_{128} \\ 0 & 0 & I_{128} & 0 \end{bmatrix} U_{2^9}((24))$$

where $U_{2^9}((24))$ denotes the matrix representation of (24) that maps the basis vectors $|i_1 i_2 i_3 i_4 i_5 i_6 i_7 i_8 i_9\rangle$ to $|i_1 i_4 i_3 i_2 i_5 i_6 i_7 i_8 i_9\rangle$, and its inverse $U_{2^9}((42))$ is the same since $(42) = (24)$.

The above method realizes the padding and permutation as linear transformations, resulting in multiplications of huge matrices. Taken literally, this method is compositional in that the embedding of a smaller circuit into a larger one can be iterated, but impractical because of the exponential growth of the dimension of the matrices. A way to avoid this problem is to stick to a symbolic representation based on sums of matrix units, that can ignore zero components, but it is less compositional, in that the representation of the gate is modified to fit an application site, leading to different representations and reasoning at different sites. We aim at solving this problem by separating the wiring part, which is a combinatorics that does not essentially touch quantum states, from the actions of a quantum gate, which is an intrinsic property of the gate itself.

3 Lenses

The first element of our approach is to provide a data structure, which we call a *lens*, that describes the composition of a subcircuit into a circuit. It forms the basis for a combinatorics of composition.

The concept of lens [7] was introduced in the programming language community as a way to solve the *view-update* problem [1], which itself comes from the database community. Lenses are often described as a pair of functions $\mathbf{get} : S \rightarrow V$ and $\mathbf{put} : V \times S \rightarrow S$, which satisfy the laws GETPUT : $\mathbf{put}(\mathbf{get}(s), s) = s$ and PUTGET : $\mathbf{get}(\mathbf{put}(v, s)) = v$. A more versatile approach adds the concept of complementary view [1, 2], which adds another type C and a function $\mathbf{get}^c : S \rightarrow C$, changing the type of \mathbf{put} to $V \times C \rightarrow S$, so that the first law becomes $\mathbf{put}(\mathbf{get}(s), \mathbf{get}^c(s)) = s$.

Our representation of lenses is an instance of the second approach. We want to map the m wires of a subcircuit to the n wires of the external one. This amounts to defining an injection from $[m]$ to $[n]$, which can be represented canonically as a list of m indices in $[n]$, without repetition.

Record $\mathbf{lens}_{n,m} := \{\ell : [n]^m \mid \mathbf{uniq} \ell\}$.

Throughout this paper, we use mathematical notations to make our COQ code easier to read. For instance $[n]$ in the above record definition denotes the ordinal type 'I_n of MATHCOMP, and $[n]^m$ denotes the type of tuples of arity m of this type (i.e. the type `m.-tuple 'I_n`). We also write type parameters as indices, and allow for omitting them.

We call *focusing* the operation using a lens to update a system according to changes in a subsystem. The following operations on lenses are basic and required to define focusing.

Definition $\mathbf{extract}_{T,n,m} : \mathbf{lens}_{n,m} \rightarrow T^n \rightarrow T^m$.

Definition $\mathbf{lensC}_{n,m} : \mathbf{lens}_{n,m} \rightarrow \mathbf{lens}_{n,n-m}$.

Definition $\mathbf{merge}_{T,n,m} : \mathbf{lens}_{n,m} \rightarrow T^m \rightarrow T^{n-m} \rightarrow T^n$.

The \mathbf{get} operation of lens ℓ is $\mathbf{extract} \ell$, which is the projection of T^n onto T^m along ℓ . Each lens ℓ has its complementary lens $\mathbf{lensC} \ell$, which is the unique monotone bijection from $[n-m]$ to $[n] \setminus \text{Im}(\ell)$. We will write ℓ^c for $\mathbf{lensC} \ell$. Their composition $\mathbf{extract} \ell^c$ returns the complementary view. The corresponding \mathbf{put} operation is $\mathbf{merge} \ell v c$. In the following, the lens ℓ will be available from the context, so that we omit it in $\mathbf{extract}$ and \mathbf{merge} , and $\mathbf{extract}^c$ denotes $\mathbf{extract} \ell^c$. The GETPUT and PUTGET laws become:

Lemma $\mathbf{merge_extract} : \mathbf{merge} (\mathbf{extract} v) (\mathbf{extract}^c v) = v$.

Lemma $\mathbf{extract_merge} : \mathbf{extract} (\mathbf{merge} v_1 v_2) = v_1$.

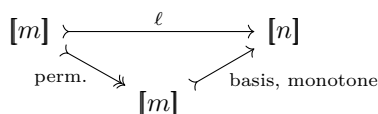
Lemma $\mathbf{extractC_merge} : \mathbf{extract}^c (\mathbf{merge} v_1 v_2) = v_2$.

We show the classical case of focusing (`focus1`) as an example (Figure 4). In this case, data is represented by direct products, whose elements are tuples, readily manipulated by $\mathbf{extract}$ and \mathbf{merge} . A change on the subsystem of type T^m is thus propagated to the global state of type T^n .

Definition $\mathbf{focus1}_{T,n,m} (\ell : \mathbf{lens}_{n,m}) (f : T^m \rightarrow T^m) : T^n \rightarrow T^n :=$
 $s \mapsto \mathbf{merge} (f (\mathbf{extract} s)) (\mathbf{extract}^c s)$.

Lemma $\mathbf{focus1_in} : \mathbf{extract} \circ (\mathbf{focus1}_\ell f) = f \circ \mathbf{extract}$.

It is also often useful to compose lenses, or to factorize a lens into its basis (the monotone part) and permutation part.



Namely, we have the following functions and laws:

15:6 Typed Compositional Quantum Computation with Lenses

$$\begin{array}{ccc}
 S = T^n & \xrightarrow{\text{extract}_\ell} & T^m = V \\
 \downarrow \text{focus}_{1_\ell} & \curvearrowright & \downarrow f \\
 T^n & \xleftarrow{\text{merge}_\ell} & T^m
 \end{array}$$

■ **Figure 4** Classical focusing.

$$\begin{array}{ccccc}
 S = T^{2^n} & \xrightarrow{\text{curry}_\ell} & (T^{2^{n-m}})^{2^m} = V & & T^{2^m} \\
 \downarrow \text{focus}_\ell & & \downarrow G_{T^{2^{n-m}}} & & \downarrow G_T \\
 T^{2^n} & \xleftarrow{\text{uncurry}_\ell} & (T^{2^{n-m}})^{2^m} & & T^{2^m}
 \end{array}$$

■ **Figure 5** Quantum focusing.

Definition $\text{lens_comp}_{n,m,p} : \text{lens}_{n,m} \rightarrow \text{lens}_{m,p} \rightarrow \text{lens}_{n,p}$.

Definition $\text{lens_basis}_{n,m} : \text{lens}_{n,m} \rightarrow \text{lens}_{n,m}$.

Definition $\text{lens_perm}_{n,m} : \text{lens}_{n,m} \rightarrow \text{lens}_{m,m}$.

Lemma $\text{lens_basis_perm} : \text{lens_comp} (\text{lens_basis } \ell) (\text{lens_perm } \ell) = \ell$.

Lemma $\text{mem_lens_basis} : \text{lens_basis } \ell =_i \ell$.

where $\ell_1 =_i \ell_2$ means that ℓ_1 and ℓ_2 are equal as sets.

4 Quantum focusing

We are going to define actions of lenses on quantum states and operators. The classical operators `merge` and `extract` introduced in the previous section play an important role in the definition.

In the quantum case, the `get` operation must not discard the irrelevant part of an input state, unlike the classical one that was defined as a projection. Such a quantum `get` and the corresponding `put` operations can be defined in a form of currying and uncurrying:

Definition $\text{curry}_{T,n,m} : \text{lens}_{n,m} \rightarrow T^{2^n} \rightarrow (T^{2^{n-m}})^{2^m}$.

Definition $\text{uncurry}_{T,n,m} : \text{lens}_{n,m} \rightarrow (T^{2^{n-m}})^{2^m} \rightarrow T^{2^n}$.

The type parameter T is intended to vary over \mathbb{C} -modules, whose archetypical example is \mathbb{C} itself. The result of applying curry_ℓ to an input state $\sigma \in T^{2^n}$ is a function that takes two indexing tuples $v \in 2^m$ and $w \in 2^{n-m}$ and returns $\sigma(\text{merge } v \ w)$, the evaluation of σ at the combined index of v and w along ℓ . Its inverse uncurry_ℓ is defined similarly as $\sigma(\text{extract } v)(\text{extract}^0 v)$ for $\sigma \in (T^{2^{n-m}})^{2^m}$ and $v \in 2^n$.

We verify that `curry` and `uncurry` form an isomorphism by cancellation lemmas:

Lemma $\text{curryK} : \text{uncurry}_\ell \circ \text{curry}_\ell = \text{id}_{T^{2^n}}$.

Lemma $\text{uncurryK} : \text{curry}_\ell \circ \text{uncurry}_\ell = \text{id}_{(T^{2^{n-m}})^{2^m}}$.

When specialized to $T := \mathbb{C}$, we can further follow another isomorphism derived from the adjunction between the category **Set** of sets and **Vect** of vector spaces, showing that our `curry` is actually equivalent to the currying for tensor products in **Vect**:

$$\mathbb{C}^{2^n} \cong (\mathbb{C}^{2^{n-m}})^{2^m} = \mathbf{Set} (2^m, \mathbb{C}^{2^{n-m}}) \cong \mathbf{Vect} (\mathbb{C}^{2^m}, \mathbb{C}^{2^{n-m}}).$$

An m -qubit quantum gate G is a linear transformation on \mathbb{C}^{2^m} , and it can be represented by a matrix. The action of this matrix on a 2^m -dimensional vector is computed only by scalar multiplications and additions. Therefore, the action can be extended to T^{2^m} for any \mathbb{C} -module T . We are thus led to endow such G with a polymorphic type of linear transformations indexed by T .

$$G : \forall T : \mathbb{C}\text{-module}, T^{2^m} \xrightarrow{\text{linear}} T^{2^m}$$

Along the curry-uncurry isomorphism above, a gate G can be applied to a larger number of qubits, to become composable in a circuit. This realizes quantum focusing (Figure 5).

$$\text{focus}_\ell G := \Lambda T.(\text{uncurry}_\ell \circ G_{T^{2^n-m}} \circ \text{curry}_\ell)$$

So far, the type of G has told that each instance G_T is linear and can be represented by a matrix, but not that they are the same matrix for any T . We impose the uniqueness of the matrix as an additional property as follows.

$$\exists M : \mathcal{M}_{2^m}(\mathbb{C}), \forall T : \mathbb{C}\text{-module}, \forall s : T^{2^m}, G_T(s) = Ms.$$

Here the multiplication Ms is defined for $s = (s_1, \dots, s_{2^m})^t$ and $M = (M_{(i,j)})_{i,j}$ as

$$Ms := \sum_{1 \leq j \leq 2^m} (M_{(1,j)} s_j, \dots, M_{(2^m,j)} s_j)^t.$$

This existence of a unique matrix representation implies the uniformity of the actions of G , which amounts to naturality with respect to the functor $(-)^{2^m}$:

$$\begin{array}{ccccc} T & T^{2^m} & \xrightarrow{G_T} & T^{2^m} & \\ \forall \varphi \downarrow & \varphi^{2^m} \downarrow & & \downarrow \varphi^{2^m} & \\ T' & T'^{2^m} & \xrightarrow{G_{T'}} & T'^{2^m} & \end{array}$$

We proved conversely that this naturality implies the uniqueness of the matrix. We shall incorporate naturality, instead of a matrix, in our definition of quantum gates.

5 Defining quantum gates

Using MATHCOMP, we can easily present the concepts described in the previous sections. From here on, we fix K to be a field, and denote by K^1 the one-dimensional vector space over K to distinguish them as different types.

We first define quantum states as the double power T^{2^n} discussed in Section 2. It is encoded as a function type $\widehat{T^n}$ from n -tuples of some finite type I to a type T . For qubits, we shall have $I = [2] = \{0, 1\}$, but we can also naturally represent qutrits (quantum information units with three states) by choosing $I = [3]$.

Variables (I : finite type) ($dI : I$) (K : field) ($T : K$ -module).

Definition $\widehat{T^n} := I^n \xrightarrow{\text{finite}} T$.

Definition $\text{dpm}_{m,T_1,T_2} (\varphi : T_1 \rightarrow T_2) (s : T_1^{\widehat{m}}) : T_2^{\widehat{m}} := \varphi \circ s$.

This construction, $(-)^{\widehat{n}}$, can be regarded as a functor with its action on functions provided by dpm_{m,T_1,T_2} , that is, any function $\varphi : T_1 \rightarrow T_2$ can be extended to $\text{dpm}_{m,T_1,T_2} \varphi : T_1^{\widehat{m}} \rightarrow T_2^{\widehat{m}}$, which are drawn as the vertical arrows in the naturality square in the previous section.

We next define quantum gates as natural transformations (or *morphisms*).

Definition $\text{morlin}_{m,n} := \forall T : K\text{-module}, T^{\widehat{m}} \xrightarrow{\text{linear}} T^{\widehat{n}}$.

Definition $\text{naturality}_{m,n} (G : \text{morlin}_{m,n}) :=$

$$\forall (T_1 T_2 : K\text{-module}), \forall (\varphi : T_1 \xrightarrow{\text{linear}} T_2), (\text{dpm}_{m,T_1,T_2} \varphi) \circ (G T_1) = (G T_2) \circ (\text{dpm}_{m,T_1,T_2} \varphi).$$

Record $\text{mor}_{m,n} := \{G : \text{morlin}_{m,n} \mid \text{naturality } G\}$.

Notation $\text{endo}_n := (\text{mor}_{n,n})$.

Definition $\text{unitary_mor}_{m,n} (G : \text{mor}_{m,n}) := \forall s, t, \langle G_{K^1} s \mid G_{K^1} t \rangle = \langle s \mid t \rangle$.

15:8 Typed Compositional Quantum Computation with Lenses

A crucial fact we rely on here is that, for any K -module T , MATHCOMP defines the K -module of the finite functions valued into it, so that $T^{\widehat{n}}$ is a K -module. This allows us to define the type `morLin` of polymorphic linear functions between $T^{\widehat{m}}$ and $T^{\widehat{n}}$, and further combine it with naturality into the types `morm,n` of morphisms from $(-)^{\widehat{m}}$ to $(-)^{\widehat{n}}$ and `endon` of endo-morphisms.

We leave unitarity as an independent property, called `unitary_mor`, since it makes sense to have non-unitary morphisms in some situations.

Concrete quantum states can be expressed directly as functions in $(K^1)^{\widehat{n}}$, or as a linear combination of computational basis vectors $|v\rangle$, where $v : I^n$ is the index of the only 1 in the vector.

Definition $|v\rangle : (K^1)^{\widehat{n}} := (v' : I^n) \mapsto \begin{cases} 1 & \text{if } v = v' \\ 0 & \text{otherwise} \end{cases}$

For a concrete tuple, we also write $|i_1, \dots, i_n\rangle$ for $[[\text{tuple } i_1; \dots; i_n]]$. This representation of states allows us to go back and forth between computational basis states and indices, and is amenable to proofs.

Using this basis, one can also define a morphism from its matrix representation (expressed as a nested double power, in column-major order). We define the CNOT gate as mapping from computational basis indices to column vectors, using $v[i]$ as a notation for the i th element of the tuple v , aka `tnth v i`. The expression `ket_bra k b` stands for the product of a column vector and a row vector, resulting in an $m \times n$ matrix (written $|k\rangle\langle b|$ in the Dirac notation). We use it to define the Hadamard gate as a sum of matrix units. Both matrices are then fed to `dpmor` to obtain morphisms.

Definition `dpmorm,n` : $((K^1)^{\widehat{n}})^{\widehat{m}} \rightarrow \text{mor}_{m,n}$.

Definition `ket_bram,n` ($k : (K^1)^{\widehat{m}}$) ($b : (K^1)^{\widehat{n}}$) : $((K^1)^{\widehat{n}})^{\widehat{m}} := v \mapsto (k \ v) \cdot b$.

Definition `cnot` : `endo2` := `dpmor` ($v : [2]^2 \mapsto |v[0], v[0] \oplus v[1]\rangle$).

Definition `hadamard` : `endo1` :=

$$\text{dpmor} \left(\frac{1}{\sqrt{2}} (\text{ket_bra } |0\rangle |0\rangle + \text{ket_bra } |0\rangle |1\rangle + \text{ket_bra } |1\rangle |0\rangle - \text{ket_bra } |1\rangle |1\rangle) \right).$$

As explained in Section 4, naturality for a morphism is equivalent to the existence of a uniform matrix representation.

Lemma `naturalityP` : `naturality G` $\longleftrightarrow \exists M, \forall T, s, G_T s = (\text{dpmor } M)_T s$.

On the right hand side of the equivalence we use the extensional equality of morphisms, which quantifies on T and s . By default, it is not equivalent to COQ's propositional equality; however the two coincide if we assume functional extensionality and proof irrelevance, two relatively standard axioms inside COQ.

Lemma `morP` : $\forall (F, G : \text{mor}_{m,n}), (\forall T, s, F_T s = G_T s) \longleftrightarrow F = G$.

While our development distinguishes between the two equalities, in this paper we will not insist on the distinction, and just abusively write $F = G$ for extensional equality too. Only in Section 9 will we use those axioms to prove and use the above lemma.

6 Building circuits

The currying defined in Section 4 allows us to compose circuits without referring to a global set of qubits. This is obtained through two operations: (sequential) composition of morphisms, which just extends function composition, and focusing through a lens, which allows us to connect the wires of a gate into a larger circuit.

Definition $\bullet_{n,m,p} : \text{mor}_{m,p} \rightarrow \text{mor}_{n,m} \rightarrow \text{mor}_{n,p}$.

Definition $\text{focus}_{n,m} : \text{lens}_{n,m} \rightarrow \text{endo}_m \rightarrow \text{endo}_n$.

To define `focus`, we combine currying and polymorphism into `focuslin` as we did in Section 4, and add a proof of naturality.

Definition $\text{focuslin}_{n,m} (\ell : \text{lens}_{n,m}) (G : \text{endo}_m) : \text{morlin}_{n,n} :=$

$$\Lambda T. (\text{uncurry } \ell)_T \circ G_{T \widehat{\text{curry}} \ell} \circ (\text{curry } \ell)_T.$$

Lemma $\text{focusN } \ell G : \text{naturality } (\text{focuslin } \ell G)$.

Definition $\text{focus}_{n,m} \ell G := (\text{a morphism packing } \text{focuslin } \ell G \text{ and } \text{focusN } \ell G)$.

In particular, `focus` and sequential composition satisfy the following laws, derived from naturality and lens combinatorics.

Lemma $\text{focus_comp} : \text{focus}_\ell (F \bullet G) = (\text{focus}_\ell F) \bullet (\text{focus}_\ell G)$.

Lemma $\text{focusM} : \text{focus}_{(\text{lens_comp } \ell \ell')} G = \text{focus}_\ell (\text{focus}_{\ell'} G)$.

Lemma $\text{focusC} : \ell \text{ and } \ell' \text{ disj.} \rightarrow (\text{focus}_\ell F) \bullet (\text{focus}_{\ell'} G) = (\text{focus}_{\ell'} G) \bullet (\text{focus}_\ell F)$.

Lemma $\text{unitary_comp} : \text{unitary_mor } F \rightarrow \text{unitary_mor } G \rightarrow \text{unitary_mor } (F \bullet G)$.

Lemma $\text{unitary_focus} : \text{unitary_mor } G \rightarrow \text{unitary_mor } (\text{focus}_\ell G)$.

The law `focus_comp` states that the sequential composition of morphism commutes with focusing. Similarly, `focusM` states that the composition of lenses commutes with focusing. The law `focusC` states that the sequential composition of two morphisms focused through disjoint lenses (i.e. lenses whose codomains are disjoint) commutes. The last two lemmas are about unitarity. Since all circuits can be built from unitary basic gates using sequential composition and `focus`, they are sufficient to guarantee unitarity for all of them.

7 Proving correctness of circuits

Once we have defined a circuit by combining gates through the above functions, we want to prove its correctness. Usually this involves proving a relation between the input and the output of the transformation, which can be expressed as a behavior on computational basis vectors. In such situations, the following lemmas allow the proof to progress.

Variables $(n m : \mathbb{N}) (\ell : \text{lens}_{n,m})$.

Definition $\text{dpmerge} : I^n \rightarrow (K^1)^{\widehat{m}} \xrightarrow{\text{linear}} (K^1)^{\widehat{n}}$.

Lemma $\text{focus_dpbasis} : (\text{focus}_\ell G)_{K^1} |v\rangle = \text{dpmerge } v (G_{K^1} |\text{extract } v\rangle)$.

Lemma $\text{dpmerge_dpbasis} : \text{dpmerge } v |v'\rangle = |\text{merge } v' (\text{extract}^{\text{g}} v)\rangle$.

Lemma $\text{decompose_scaler} : \forall (\sigma : (K^1)^{\widehat{n}}), \sigma = \sum_{v:I^k} \sigma(v) \cdot |v\rangle$.

The function `dpmerge` embeds the result of a quantum gate applied to a part of the system into the whole system, using the input computational basis vector for complement; this can be seen as an asymmetric variant of the `put` operation. It is defined using `uncurryℓ` and `dpmap`. It is only introduced and eliminated through the two lemmas following. The helper law `focus_dpbasis` allows one to apply the morphism `G` to the local part of the basis vector `v`. The result of this application must then be decomposed into a linear combination of (local) basis vectors, either by using the definition of the gate, or by using `decompose_scaler`. One can then use linearity to obtain terms of the form `dpmerge v |v'⟩` and merge the local result into the global quantum state. Linear algebra computations have good support in `MATHCOMP`, so we do not need to extend it much.

15:10 Typed Compositional Quantum Computation with Lenses

Extraction and merging only rely on lens-related lemmas, orthogonal to the linear algebra part. We have not yet developed a complete theory of lenses, but we have many such lemmas. The following ones are of particular interest:

Section `lens_index`.

Variables $(n\ m : \mathbb{N})\ (i : [n])\ (\ell : \text{lens}_{n,m})$.

Definition `lens_index` $(H : i \in \ell) : [m]$.

Lemma `tnth_lens_index` $: \forall (H : i \in \ell), \ell[\text{lens_index } H] = i$.

Lemma `tnth_merge` $: \forall (H : i \in \ell), (\text{merge } v\ v')[i] = v[\text{lens_index } H]$.

Lemma `tnth_extract` $: (\text{extract } v)[j] = v[\ell[j]]$.

Lemma `mem_lensC` $: (i \in \ell^c) = (i \notin \ell)$.

Lemma `mem_lens_comp` $: \forall (H : i \in \ell), (i \in \text{lens_comp } \ell\ \ell') = (\text{lens_index } H \in \ell')$.

End `lens_index`.

Lemma `tnth_mergeC` $: \forall (H : i \in \ell^c), (\text{merge } v\ v')[i] = v'[\text{lens_index } H]$.

The expression `lens_index H`, where H is a proof that i is in ℓ , denotes the ordinal position of i in ℓ , hence the statement of `tnth_lens_index`. It is particularly useful in `tnth_merge` and `tnth_mergeC`, where it allows one to prove equalities of tuples and lenses through case analysis on the boolean expression $i \in \ell$ (using `mem_lensC` for conversion).

Using these two techniques we have been able to prove the correctness of a number of pure quantum circuits, such as Shor's 9-qubit code or the GHZ preparation.

8 Concrete examples

When working on practical examples we move to more concrete settings. Namely, we use \mathbb{C} as the coefficient field, which can also be seen as the vector space $\text{Co} = \mathbb{C}^1$. The indices are now in $\mathbb{I} = [2] = \{0, 1\}$. In this section we use Coq notations rather than the mathematical ones of the previous sections, so as to keep close to the actual code.

As an example, let us recall the circuit diagram of Shor's code (Figure 1). It consists of two smaller components, bit-flip and sign-flip codes (Figures 2 and 3), in such a way that three bit-flip codes are placed in parallel and surrounded by one sign-flip code. This construction can be expressed straightforwardly as the following Coq code.

Definition `bit_flip_enc` $: \text{endo}_3 := \text{focus } [\text{lens } 0; 2] \text{ cnot} \bullet \text{focus } [\text{lens } 0; 1] \text{ cnot}$.

Definition `bit_flip_dec` $: \text{endo}_3 := \text{focus } [\text{lens } 1; 2; 0] \text{ toffoli} \bullet \text{bit_flip_enc}$.

Definition `hadamard3` $: \text{endo}_3 :=$

`focus` $[\text{lens } 2] \text{ hadamard} \bullet \text{focus } [\text{lens } 1] \text{ hadamard} \bullet \text{focus } [\text{lens } 0] \text{ hadamard}$.

Definition `sign_flip_dec` $:= \text{bit_flip_dec} \bullet \text{hadamard3}$.

Definition `sign_flip_enc` $:= \text{hadamard3} \bullet \text{bit_flip_enc}$.

Definition `shor_enc` $: \text{endo}_9 :=$

`focus` $[\text{lens } 0; 1; 2] \text{ bit_flip_enc} \bullet \text{focus } [\text{lens } 3; 4; 5] \text{ bit_flip_enc} \bullet$

`focus` $[\text{lens } 6; 7; 8] \text{ bit_flip_enc} \bullet \text{focus } [\text{lens } 0; 3; 6] \text{ sign_flip_enc}$.

Definition `shor_dec` $: \text{endo}_9 := \dots$

We proved that Shor's code is the identity on an error-free channel:

Theorem `shor_code_id` $: (\text{shor_dec} \bullet \text{shor_enc}) \ |i, 0, 0, 0, 0, 0, 0, 0, 0\rangle = \ |i, 0, 0, 0, 0, 0, 0, 0, 0\rangle$.

The proof is compositional, relying on lemmas for each subcircuit.

Lemma `cnotE` $: \text{cnot} \ |i, j\rangle = \ |i, i + j\rangle$.

Lemma `toffoliE00` $: \text{toffoli} \ |0, 0, i\rangle = \ |0, 0, i\rangle$.

Lemma `hadamardK` $: \forall T, \text{involutive } \text{hadamard}_T$.

```

1   bit_flip_enc | i, j, k >
2   rewrite /=.
3   = focus [lens 0; 2] cnot (focus [lens 0; 1] cnot | i, j, k >)
4   rewrite focus_dpbasis.
5   = focus [lens 0; 2] cnot (dpmerge [lens 0; 1] [tuple i; j; k]
6                               (cnot | extract [lens 0; 1] [tuple i; j; k]))
7   simpl_extract.
8   = focus [lens 0; 2] cnot (dpmerge [lens 0; 1] [tuple i; j; k] (cnot | i, j >))
9   rewrite cnotE.
10  = focus [lens 0; 2] cnot (dpmerge [lens 0; 1] [tuple i; j; k] | i, i + j >)
11  rewrite dpmerge_dpbasis.
12  = focus [lens 0; 2] cnot | merge [lens 0; 1] [tuple i; i + j]
13                                (extract (lensC [lens 0; 1]) [tuple i; j; k]) >
14  simpl_merge.
15  = focus [lens 0; 2] cnot | i, i + j, k >

```

■ **Figure 6** Excerpt of interactive proof of `bit_flip_enc_ok`.

Lemma `bit_flip_enc_ok` : `bit_flip_enc |i, j, k> = |i, i + j, i + k>`.

Lemma `bit_flip_toffoli` : `bit_flip_dec • bit_flip_enc = focus [lens 1;2;0] toffoli`.

Lemma `sign_flip_toffoli`: `sign_flip_dec • sign_flip_enc = focus [lens 1;2;0] toffoli`.

The notation $(i : [m])$ in expressions (here in `flip`) denotes that we have a proof that $i \in [m]$; in the actual code one uses specific function to build such dependently-typed values. The first 3 lemmas describe properties of the matrix representation of gates, and involve linear algebra computations. The proof of `HadamardK` also involves some real computations about $\sqrt{2}$. The remaining 3 lemmas and the theorem do mostly computations on lenses. In total, there were about 100 lines of proof.

To give a better idea of how the proofs proceed, we show a few steps of the beginning of `bit_flip_enc_ok`, in Figure 6, interspersing tactics on a gray background between quantum state expressions and equations. Lines beginning with an “=” symbol state that the expression is equal to the previous one.

Simplifying on line 2 reveals the focused application of the two `cnot` gates. Rewriting with `focus_dpbasis`, on line 4, applies the first gate directly to a basis vector. The helper tactic `simpl_extract`, on line 7, computes the tuple obtained by `extract` (`MATHCOMP` is not good at computing in presence of dependent types). It results here in the vector $|i, j\rangle$, which we can rewrite with `cnotE`. As a result, on line 10, `dpmerge` is applied to a basis vector, so that we can rewrite it with `dpmerge_dpbasis`. Again, on line 14, we use a helper tactic `simpl_merge`, which uses the same code as `simpl_extract` to simplify the value of the merge expression. We obtain $|i, i + j, k\rangle$ as result after the first gate, and can proceed similarly with the second gate to reach $|i, i + j, i + k\rangle$.

As we explained above, our approach cleanly separates computation on lenses from linear algebra parts. Namely, in the above proof we have three logical levels: `focus_dpbasis` and `dpmerge_dpbasis` let one get in and out of a `focus` application; `simpl_extract` and `simpl_merge` are doing lens computations; and finally `cnotE` uses a property of the specific gate.

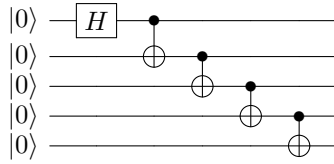
The proof of `shor_code_id` is more involved as the Hadamard gates introduce superpositions. The code can be found in the file `qexamples_shor.v` of the accompanying development, and is about 30 lines long. We will just explain here the main steps of the proof. The basic idea is to pair the encoders and decoders, and to turn them into Toffoli gates, which happen to be identities when the extra inputs are zeros. The first goal is to prove that

```

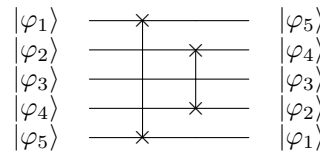
(shor_dec • shor_enc) |i, 0, 0, 0, 0, 0, 0, 0> =
focus [lens 0;3;6] (sign_flip_dec • sign_flip_enc) |i, 0, 0, 0, 0, 0, 0, 0>

```

15:12 Typed Compositional Quantum Computation with Lenses



■ **Figure 7** 5-qubit GHZ state preparation.



■ **Figure 8** 5-qubit reversed state circuit.

If we expand the compositions on both sides, we see that they both start by applying `focus [lens 0;3;6] sign_flip_enc` to the input. We can use `focus_dpasis` and `simpl_extract` to progress, but due to the Hadamard gates in `sign_flip_enc`, the state of the corresponding 3 qubits becomes non-trivial. However, we can use `decompose_scaler` to see this state as a sum of unknown computational basis vectors, and progress using linear algebra lemmas, to reach the bit-flip part of the circuit. Once we do that, the remainder of the proof consists in using `focusC` to reorder the bit-flip encoders and decoders, so that the corresponding ones are sequentially paired. We can then use `focus_comp` to produce applications of `bit_flip_dec • bit_flip_enc`, which can be converted to Toffoli gates by `bit_flip_toffoli`. Then we observe that in the input the ancillaries are all zeros, so that the result of each gate is the identity, which concludes the first part of the proof. Then we can proceed similarly to prove that the remaining composition of the sign-flip encoder and decoder is the identity, which concludes the proof.

Another interesting example is the Greenberger-Horne-Zeilinger (GHZ) state preparation. It is a generalization of the Bell state, resulting in a superposition of $|0\rangle^{\otimes n}$ and $|1\rangle^{\otimes n}$, which denote states composed of n zeroes and ones, respectively. As a circuit, it can be expressed by the composition of one Hadamard gate followed by n CNOT gates, each one translated by 1 qubit, starting from the state $|0\rangle^{\otimes n}$. The 5-qubit case is shown in Figure 7.

We can write the transformation part as follows in our framework (for an arbitrary n):

```

Lemma succ_neq n (i : [n]) : (i : [n + 1]) ≠ (i + 1 : [n + 1]).
Fixpoint ghz n :=
  match n as n return endo_{n+1} with
  | 0 => hadamard
  | m.+1 => focus (lens_pair (succ_neq (m : [m.+1]))) cnot •
             focus (lensC (lens_single (m.+1 : [m.+2]))) (ghz m)
  end.

```

The definition works by composing `ghz(m)`, which has type `endon` (since $n = m + 1$), with an extra CNOT gate. Note that we use dependent types, and the recursion is at a different type. The lemma `succ_neq` is a proof that $i \neq i + 1$ in $[n + 1]$. It is used by `lens_pair` to build the lens `[lens m; m + 1]` from `[2]` to `[m + 2]`. `lens_single` builds a singleton lens, so that `lensC (lens_single (m.+1 : [m.+2]))` is the lens from `[m + 1]` to `[m + 2]` connecting the inner circuit to the first $m + 1$ wires. We can express the target state and correctness property as follows:

Definition `ghz_state n : $(\mathbb{C}^1)^{\widehat{n+1}}$:= $\frac{1}{\sqrt{2}} \cdot (|0\rangle^{\otimes(n+1)} + |1\rangle^{\otimes(n+1)})$.`

Lemma `ghz_ok : ghz n $|0\rangle^{\otimes(n+1)}$ = ghz_state n.`

Due to the nesting of lenses, the proof includes a lot of lens combinatorics, and is about 50 lines long. We only show the last few lines of the proof in Figure 9, as they include typical steps. They prove the action of the last CNOT gate of the circuit when it propagates a 1 to the last qubit of the state. The notation `[tuple F i | i < n]` denotes the n -tuple whose i th

```

1   lp := lens_pair (succ_neq (n : [n.+1]))
2   =====
3   merge lp [tuple 1; 1]
4     (extract (lensC lp) [tuple if i != n.+1 then 1 else 0 | i < n.+2])
5   = [tuple 1 | _ < n.+2]
6   apply eq_from_tnth => i; rewrite [RHS]tnth_mktuple.
7   case/boolP: (i \in lp) => Hi.
8     Hi : i \in lp
9     =====
10    tnth (merge lp [tuple 1; 1]
11      (extract (lensC lp) [tuple if i0 != n.+1 then 1 else 0 | i0 < n.+2])) i = 1
12  rewrite tnth_merge -[RHS](tnth_mktuple (fun=>1) (lens_index Hi)).
13    tnth [tuple 1; 1] (lens_index Hi) = tnth [tuple 1 | _ < 2] (lens_index Hi)
14  by congr tnth; eq_lens.
15    Hi : i \notin lp
16    =====
17    tnth (merge lp [tuple 1; 1]
18      (extract (lensC lp) [tuple if i0 != n.+1 then 1 else 0 | i0 < n.+2])) i = 1
19  rewrite -mem_lensC in Hi.
20  rewrite tnth_mergeC tnth_extract tnth_mktuple.
21    Hi : i \in lensC lp
22    =====
23    (if tnth (lensC lp) (lens_index Hi) < n.+1 then 1 else 0) = 1
24  rewrite tnth_lens_index ifT //.
25    i != n.+1
26  move: Hi; rewrite mem_lensC !inE; apply contra.
27    i == n.+1 -> (i == (n : [n.+2])) || (i == (n.+1 : [n.+2]))
28  by move/eqP => Hi; apply/orP/or_intror/eqP/val_inj.

```

■ **Figure 9** Excerpt of interactive proof of `ghz_ok`.

element is $F\ i$. Lemma `eq_from_tnth` on line 6 allows index-wise reasoning. The `tnth_mktuple` on the same line extracts the i th element of the tuple comprehension on the right-hand side. We immediately do a case analysis on whether i is involved in the last gate. In the first case, we have $i \in \text{lens_pair}(\text{succ_neq}(n : [n + 1]))$, so we can use `tnth_merge` on the left-hand side. On the right-hand side we use `tnth_mktuple` backwards, to introduce a 2-tuple. As a result, we obtain on line 13 a goal on which we can use congruence, and conclude with `eq_lens` as both tuples are equal. The second case, when $i \notin \text{lens_pair}(\text{succ_neq}(n : [n + 1]))$, is more involved. By using `mem_lensC` in `Hi`, we can use `tnth_mergeC`, followed by `tnth_extract` and `tnth_mktuple` to reach the goal at line 21. But then the argument to `tnth` is precisely that of `Hi`, so this expression can be rewritten to i by `tnth_lens_index`. From line 25 on it just remains to prove that i cannot be $n + 1$, which is true since it is in the complement of `lens_pair (succ_neq (n : [n + 1]))`.

9 Parallel composition

In this section, we extend our theory with noncommutative and commutative monoids of the sequential and parallel compositions of morphisms. Thanks to quantum state currying, we have been able to define focusing and composition of circuits without relying on the Kronecker product. This also means that parallel composition is not primitive in this system. Thanks to `focusC`, morphisms applied through disjoint lenses do commute, but it is harder

15:14 Typed Compositional Quantum Computation with Lenses

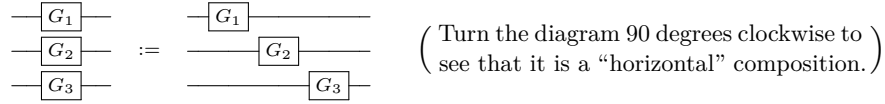
to extend this to an n -ary construct, as done in CoqQ [20]. Yet it is possible to define parallel composition using MATHCOMP *big operators* by defining a new notion of commuting composition of morphisms. Note that big operators on monoids require axioms based on propositional equality, rather than the extensional equality of morphisms, so in this section we assume functional extensionality and proof irrelevance, which allows us to use lemma `morP` of Section 5.

As a first step, we define the noncommutative monoid of morphisms, using the sequential (*vertical* in category-theoretic terminology) composition as monoid operation and the identity morphism as unit element. Registering the associativity and unitality laws with Hierarchy Builder [6], allows one to use the corresponding m -ary big operator.

```
HB.instance Definition _ := Monoid.isLaw.Build on •n,n,n and idmorn.
Definition compn_mor m (F : [m] → endon) (P : pred [n]) :=
  \big[•n,n,n/idmorn](i < n, P i) F i.
```

By itself, it just allows us to define some circuits in a more compact way. It will also allow us to connect with the commutative version.

The parallel (*horizontal*) composition of morphisms is derived from vertical composition, in the case where the morphisms focused in a circuit have disjoint supports.



We construct a commutative monoid whose operation is the horizontal composition, by reifying the notion of focused morphism (inside an n -qubit circuit), using the corresponding lens to express the support.

```
Record foc_endon := {(m, ℓ, e) : ℕ × lensn,m × endom | ℓ is monotone}.
```

The monotonicity of ℓ in focused morphisms is demanded for the canonicity and strictness of their compositions. The arity m of the morphism is existentially quantified.

The actual COQ definition of `foc_endo` has four fields `foc_m`, `foc_l`, `foc_e`, and `foc_s`, the first three corresponding to m, ℓ, e above, and the last one being the proof that ℓ is monotone. We define `mkFendo`, a “smart constructor” that factorizes a given lens (`lens_basis` and `lens_perm` in Section 3) into its basis (whose monotonicity proof being `lens_sorted_basis`) and permutation to build a focused morphism.

```
Definition mkFendon,m (ℓ : lensn,m) (G : endom) :=
  {| foc_s := lens_sorted_basis ℓ; foc_e := focus (lens_perm ℓ) G |}.
```

Focused morphisms come with both a unit element and an annihilating (zero) element.

```
Definition id_fendo := mkFendo (lens_empty n) (idmor I K 0).
```

```
Definition err_fendo := mkFendo (lens_id n) (nullmor n n).
```

The unit element `id_fendo` has an empty support, and the zero element `err_fendo` has a full support.

A focused morphism can be used as an ordinary morphism at arity n by actually focusing the morphism field e along the lens field ℓ (field projections `foc_l` and `foc_e` are denoted by $\cdot\ell$ and $\cdot e$).

```
Definition fendo_mor (Φ : foc_endo) : endon := focus Φ.ℓ Φ.e.
```

We can then define commutative composition `comp_fendo`.

Definition `par_compp,q` ($F : \text{endo}_p$) ($G : \text{endo}_q$) : $\text{endo}_{p+q} :=$
 $(\text{focus lens_left } F) \bullet (\text{focus lens_right } G)$

Definition `comp_fendo` ($\Phi \Psi : \text{foc_endo}$) :=

$$\begin{cases} \text{mkFendo } (\Phi.\ell ++ \Psi.\ell : \text{lens}_{n, \Phi.m + \Psi.m}) (\text{par_comp } \Phi.e \Psi.e) & \text{if } \Phi.\ell \text{ and } \Psi.\ell \text{ are disjoint} \\ \text{err_fendo} & \text{otherwise} \end{cases}$$

To make composition commutative, we return the zero element whenever the lenses of the two morphisms are not disjoint. If they are disjoint, we return their composition, using the union of the two lenses. We require lenses to be monotone to guarantee associativity.

Using this definition of commutative composition, we can declare the commutative monoid structure on focused morphisms and define their m -ary parallel composition. When the lenses are pairwise disjoint, it coincides with `compn_mor`.

`HB.instance Definition _ := Monoid.isComLaw.Build on comp_fendo and id_fendo.`

Variables ($m : \mathbb{N}$) ($F : [m] \rightarrow \text{foc_endo}$) ($P : \text{pred } [m]$).

Definition `compn_fendo` := $\bigwedge_{(i < m, P \ i)} F \ i$.

Hypothesis `Hdisj` : $\forall i, j, i \neq j \rightarrow (F \ i).\ell \text{ and } (F \ j).\ell \text{ are disjoint}$.

Theorem `compn_mor_disjoint` : `compn_mor (fendo_mor o F) P = fendo_mor compn_fendo`.

To exemplify the use of this commutative monoid, we proved that the circuit that consists of $\lfloor n/2 \rfloor$ swap gates that swap the i th and $(n - i - 1)$ th of n qubits returns a reversed state (Figure 8).

Lemma `rev_ord_neqn` ($i : \lfloor n/2 \rfloor$) : $(i : [n]) \neq (n - i - 1 : [n])$.

Definition `rev_circuit` $n : \text{endo}_n :=$

`compn_mor (i ↦ focus (lens_pair (rev_ord_neq i)) swap) xpredT.`

Lemma `rev_circuit_ok` : $\forall (i : [n]),$

`proj (lens_single (n - i - 1 : [n])) (rev_circuit n σ) = proj (lens_single i) σ.`

Here `rev_ord_neq` produces an inequality in $[n]$, which we can use to build the required pair lens to apply `swap`.

10 Related works

There are many works that aim at the mechanized verification of quantum programs [14]. Here we only compare with a number of like-minded approaches, built from first principles, i.e. where the formalization includes a model of computation based on unitary transformations, which justifies the proof steps.

Qiskit [16] is a framework for writing quantum programs in Python. While it does not let one write proofs, it has the ability to turn a circuit into a gate, allowing one to reuse it in other circuits, so that it has definitional compositionality.

QWIRE [15] and SQIR [11] define a quantum programming language and its Hoare logic in Coq, modeling internally computation with matrices and Kronecker products. QWIRE and SQIR differ in their handling of variables: in QWIRE they are abstract, handled through higher-order abstract syntax, but in SQIR, which was originally intended as an intermediate language for the compilation of QWIRE, they are concrete natural numbers, denoting indices of qubits. The authors note in their introduction [11] that “[abstract variables] necessitate a map from variables to indices, which we find confounds proof automation”. They go on remarking that having a distinct semantics for pure quantum computation, rather than relying only on the density matrices needed for hybrid computations, considerably simplifies proofs; this justifies our choice of treating specifically the pure case. While QWIRE satisfies definitional compositionality, this is not the case for SQIR, as circuits using fixed indices cannot be directly reused. We have not proved enough programs to provide a meaningful

comparison, yet it is noteworthy that our proof of GHZ, which uses virtually no automation, is about half the size of the proof in SQIR [11]. The main difference is that we are able to solve combinatorics at the level of lenses, while they have to work all along with a symbolic representation of matrices, that is a linear combination of matrix units (Dirac’s notation), to avoid working directly on huge matrices.

VyZX [12, 13] formalizes the ZX-calculus in Coq, on top of SQIR. Its goal is to prove graph-rewriting rules, and ultimately to build a verified optimizer for the ZX-calculus. However, as they state themselves, the graphical nature of the calculus appears to be a major difficulty, and only restricted forms of the rules are proved at this point. Since the ZX-calculus itself enjoys compositionality, albeit at the graph level, this is a promising line of work. It would be interesting to see if our approach can make proving such graph-rewriting rules easier. As preliminary experiment, we have proved the triangular identity involving a cup and a cap, by defining an asymmetric version of focusing. More generally, finding a nice way to compose graphs is essential, and concepts such as lenses could have a role there.

CoqQ [20] builds a formalized theory of Hilbert spaces and n-ary tensor products on top of MATHCOMP, adding support for the so-called *labelled Dirac notation*. Again they define a Hoare logic for quantum programs, and are able to handle both pure and hybrid computations. While the labelled Dirac notation allows handling commutation comfortably, it does not qualify as compositional, since it is based on a fixed set of labels, i.e. one cannot mix programs if they do not use the same set of labels.

Unruh developed a quantum Hoare logic and formalized it in Isabelle, using a concept of *register* [18] for which he defines a theory, including operations such as taking the complement of a register. His registers in some meaning generalize our `focus` function, as they allow focusing between arbitrary types rather than just sets of qubits. Since one can compose registers, his approach is compositional, for both definitions and proofs, and the abstraction overhead is avoided through automation. However, while each application of `focus` to a lens can be seen as a register, he has not separated out a concrete combinatorics based on finite objects similar to our notion of lens.

In a slightly different direction, Qbricks [3] uses the framework of *path-sums* to allow the automatic proof of pure quantum computations. The notion of path is more expressive than that of computational basis state, and allows one to represent many unitary transformations as maps from path to path, making calculations easier. It would be interesting to see whether it is possible to use them in our framework.

Most approaches above support not only pure quantum computation but also hybrid quantum-classical computation. While we have concentrated here on pure computation, we have already extended our approach to the density-matrix interpretation required to support hybrid computations, and verified that it commutes with focusing. Practical applications are left to future work.

Note also that, while some of the above works use dependent types to represent matrix sizes for instance, they all rely on ways to hide or forget this information as a workaround. On the other hand, our use of dependent types is strict, only relying on statically proved cast operators to adjust types where needed, yet it is lightweight enough for practical use.

Some other aspects of our approach can be related to programming language theory. For instance, the way we shift indices during currying is reminiscent of De Bruijn indices, and our `merge` operation shifts indices in the precise same way as the record concatenation defined in the label-selective λ -calculus [8]. This suggests that our currying of quantum states is actually similar to the currying occurring in that calculus.

11 Conclusion

We have been able to build a compositional model of pure quantum computation in COQ, on top of the MATHCOMP library, by using finite functions, lenses, and focusing. We have applied the development to prove the correctness of several quantum circuits. An interesting remark is that, while we started from the traditional view of seeing quantum states as tensor products, our implementation does not rely on the Kronecker product for composing transformations. Since the Kronecker product of matrices can be cumbersome to work with, this is a potential advantage of this approach.

Many avenues are open for future work. First we need to finish the proof of Shor's code, this time for erroneous channels; paper proofs are simple enough but the devil is in the details. Next, building on our experience, we would like to formalize and abstract the algebraic theory of lenses. Currently we rely on a large set of lemmas developed over more than a year, without knowing their interdependencies; such a theory would have both theoretical and practical implications. Third, we are interested in the category-theoretic aspects of this approach, and would like to give an account of `focus`, explaining both the relation between a lens and its action, and the structural properties of focusing.

References

- 1 F. Bancilhon and N. Spyratos. Update semantics of relational views. *ACM Trans. Database Syst.*, 6(4):557–575, December 1981. doi:10.1145/319628.319634.
- 2 Davi M.J. Barbosa, Julien Cretin, Nate Foster, Michael Greenberg, and Benjamin C. Pierce. Matching lenses: alignment and view update. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming, ICFP '10*, pages 193–204, New York, NY, USA, 2010. Association for Computing Machinery. doi:10.1145/1863543.1863572.
- 3 Christophe Chareton, Sébastien Bardin, François Bobot, Valentin Perrelle, and Benoît Valiron. An automated deductive verification framework for circuit-building quantum programs. In Nobuko Yoshida, editor, *Programming Languages and Systems, ESOP 2021*, volume 12648 of *Lecture Notes in Computer Science*, pages 148–177, Cham, March 2021. Springer International Publishing. doi:10.1007/978-3-030-72019-3_6.
- 4 Bob Coecke and Ross Duncan. Interacting quantum observables. In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfssdóttir, and Igor Walukiewicz, editors, *Automata, Languages and Programming*, pages 298–310, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. doi:10.1007/978-3-540-70583-3_25.
- 5 Bob Coecke and Aleks Kissinger. *Picturing Quantum Processes: A First Course in Quantum Theory and Diagrammatic Reasoning*. Cambridge University Press, 2017. doi:10.1017/9781316219317.
- 6 Cyril Cohen, Kazuhiko Sakaguchi, and Enrico Tassi. Hierarchy builder: Algebraic hierarchies made easy in Coq with Elpi (system description). In *5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020), June 29–July 6, 2020, Paris, France (Virtual Conference)*, volume 167 of *LIPICs*, pages 34:1–34:21. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPICs.FSCD.2020.34.
- 7 J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Trans. Program. Lang. Syst.*, 29(3):17, 2007. doi:10.1145/1232420.1232424.
- 8 Jacques Garrigue and Hassan Ait-Kaci. The typed polymorphic label-selective λ -calculus. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 35–47, 1994. doi:10.1145/174675.174434.

- 9 Jacques Garrigue and Takafumi Saikawa. QECC: Quantum Computation and Error-Correcting Codes. Software, swhId: [swh:1:dir:d4d158675180ee276e730bd7f67a9122a6472eb3](https://github.com/t6s/qecc) (visited on 2024-08-21). URL: <https://github.com/t6s/qecc>.
- 10 Daniel M. Greenberger, Michael A. Horne, and Anton Zeilinger. Going beyond bell's theorem. In Menas Kafatos, editor, *Bell's Theorem, Quantum Theory and Conceptions of the Universe*, pages 69–72. Springer Netherlands, Dordrecht, 1989. doi:10.1007/978-94-017-0849-4_10.
- 11 Kesha Hietala, Robert Rand, Shih-Han Hung, Xiaodi Wu, and Michael Hicks. A verified optimizer for quantum circuits. *Proc. ACM Program. Lang.*, 5(POPL), January 2021. doi:10.1145/3434318.
- 12 Adrian Lehmann, Ben Caldwell, and Robert Rand. VyZX : A vision for verifying the ZX calculus, 2022. doi:10.48550/arXiv.2205.05781.
- 13 Adrian Lehmann, Ben Caldwell, Bhakti Shah, and Robert Rand. VyZX: Formal verification of a graphical quantum language, 2023. doi:10.48550/arXiv.2311.11571.
- 14 Marco Lewis, Sadegh Soudjani, and Paolo Zuliani. Formal verification of quantum programs: Theory, tools and challenges, 2022. doi:10.1145/3624483.
- 15 Jennifer Paykin, Robert Rand, and Steve Zdancewic. QWIRE: A core language for quantum circuits. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL '17, pages 846–858, 2017. doi:10.1145/3009837.3009894.
- 16 Qiskit contributors. Qiskit: An open-source framework for quantum computing, 2023. doi:10.5281/zenodo.2573505.
- 17 Peter W. Shor. Scheme for reducing decoherence in quantum computer memory. *Phys. Rev. A*, 52:R2493–R2496, October 1995. doi:10.1103/PhysRevA.52.R2493.
- 18 Dominique Unruh. Quantum and classical registers. *CoRR*, abs/2105.10914, 2021. doi:10.48550/arXiv.2105.10914.
- 19 Mingsheng Ying. *Foundations of Quantum Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2016. doi:10.1016/C2014-0-02660-3.
- 20 Li Zhou, Gilles Barthe, Pierre-Yves Strub, Junyi Liu, and Mingsheng Ying. CoqQ: Foundational verification of quantum programs. *Proc. ACM Program. Lang.*, 7(POPL), January 2023. doi:10.1145/3571222.

A Formal Proof of $R(4,5)=25$

Thibault Gauthier  

Czech Technical University in Prague, Czech Republic

Chad E. Brown

Czech Technical University in Prague, Czech Republic

Abstract

In 1995, McKay and Radziszowski proved that the Ramsey number $R(4,5)$ is equal to 25. Their proof relies on a combination of high-level arguments and computational steps. The authors have performed the computational parts of the proof with different implementations in order to reduce the possibility of an error in their programs. In this work, we prove this theorem in the interactive theorem prover HOL4 limiting the uncertainty to the small HOL4 kernel. Instead of verifying their algorithms directly, we rely on the HOL4 interface to MiniSat to prove gluing lemmas. To reduce the number of such lemmas and thus make the computational part of the proof feasible, we implement a generalization algorithm. We verify that its output covers all the possible cases by implementing a custom SAT-solver extended with a graph isomorphism checker.

2012 ACM Subject Classification Theory of computation → Automated reasoning

Keywords and phrases Ramsey numbers, SAT solvers, symmetry breaking, generalization, HOL4

Digital Object Identifier 10.4230/LIPIcs.ITP.2024.16

Related Version *Full Version:* <https://arxiv.org/abs/2404.01761>

Funding Both authors were supported by the Ministry of Education, Youth and Sports within the dedicated program ERC CZ under the project POSTMAN no. LL1902. The second author has also received funding from the European Union’s Horizon Europe research and innovation programme under grant agreement no. 101070254 CORESENSE.

1 Introduction

Formalizations are useful to verify that there are no bugs in some software and also that there are no errors in a mathematical proof. Researchers write their formalizations in an interactive theorem prover also called a proof assistant. An interactive theorem prover transforms high-level proof steps, written by its user in the language of the interactive theorem prover, into low-level proof steps at the level of logical rules and axioms. These low-level steps are then verified by the kernel of the proof assistant. Formalizations are thus doubly appropriate when a proof combines advanced human-written arguments and computer-generated lemmas. This is the case for the four-color theorem [3] which was proved by Appel and Haken in 1976 and the Kepler conjecture [13] which was proved by Hales and Ferguson in 1998.

In those two cases, a human argument is used to reduce a potentially infinite number of cases to a finite number. Then, a computer algorithm is used to generate a proof for each of these cases. The generated proofs are too numerous to be verified manually and so the generating code, which is in those cases quite complicated, had to be trusted. To avoid trusting that the generating code fits together with the human argument, a formalization of the four-color theorem [11] was completed in the Coq proof assistant [4] by Gonthier in 2005 and a formalization of the Kepler conjecture [12] was completed in the HOL Light proof assistant [14] by a team led by Hales in 2014.

The case of $R(4,5) \leq 25$ is different. Since it is a finite problem, one could prove it by considering a finite number of cases. Since there are $\frac{25 \times 24}{2} = 300$ edges in a graph with 25 vertices, a naive proof would consist of checking the presence of a 4-clique or a 5-independent set in all graphs of size 25 which would amount to $2^{300} \approx 10^{90}$ graphs. Another approach



© Thibault Gauthier and Chad E. Brown;

licensed under Creative Commons License CC-BY 4.0

15th International Conference on Interactive Theorem Proving (ITP 2024).

Editors: Yves Bertot, Temur Kutsia, and Michael Norrish; Article No. 16; pp. 16:1–16:18

Leibniz International Proceedings in Informatics



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

would be to encode the clique constraints into a SAT solver. This reduces the search space dramatically but so far no proof of $R(4, 5) \leq 25$ relying only on calls to SAT solvers has been found.

The proof of McKay and Radziszowski [18] first uses a high-level argument and then relies on a pre-processing algorithm to reduce the number of cases to a manageable number. Each of those cases requires proving that a pair of graph cannot occur together in an $\mathcal{R}(4, 5, 25)$ -graph. These kinds of problems are called gluing problems. Our formalization of $R(4, 5) = 25$ in the HOL4 theorem prover [19] will mostly follow the initial splitting argument. We construct a slightly different pre-processing algorithm that uses gray edges instead of removing vertices. We also make use of the HOL4 interface [20] to the SAT solver MiniSat [9], instead of re-using the custom-built solver of McKay and Radziszowski, to prove that the gluing problems are unsatisfiable. This greatly simplifies our proof as we do not need to trace the proof steps of their optimized solver and we do not have to replay those proof steps in HOL4. Additional differences between our formal proof and the original proof are discussed in Section 8.

We now explain in more detail the different components of our formal proof. To conclude that $R(4, 5) = 25$, we prove that $R(4, 5) \leq 25$ and that $R(4, 5) > 24$. The statement $R(4, 5) > 24$ can simply be proved by exhibiting an $\mathcal{R}(4, 5, 24)$ -graph. The existence of such graph has been known since 1965 thanks to a construction by Kalbfleisch [16]. The formal proof of the existence of an $\mathcal{R}(4, 5, 24)$ -graph is given in Section 6.2. The other parts of this paper describe how to formally prove the more challenging statement $R(4, 5) \leq 25$. In Section 2, we first give three important definitions. In particular, we define the Ramsey number $R(4, 5)$ which is necessary to state the final theorem. In Section 3, we prove that in an $\mathcal{R}(4, 5, 25)$ -graph there exists a vertex of degree $d \in \{8, 10, 12\}$ and that the neighbors of that vertex form an $\mathcal{R}(3, 5, d)$ -graph and the antineighbors form an $\mathcal{R}(4, 4, 24 - d)$ -graph as illustrated in Figure 1. This vertex is referred to in other parts of the proofs as the splitting vertex. In Section 4, we enumerate all possible $\mathcal{R}(3, 5, d)$ -graphs and $\mathcal{R}(4, 4, 24 - d)$ -graphs modulo isomorphism. We then regroup similar graphs together in what we call generalizations. In Section 5, we prove that there is no way to glue an $\mathcal{R}(3, 5, d)$ -generalization and an $\mathcal{R}(4, 4, 24 - d)$ -generalization while respecting the clique constraints. This is achieved by encoding the gluing into SAT and calling the HOL4 interface to MiniSat. In Section 5.2, we improve the construction of generalizations by preferring ones resulting in easier gluing problems. This selection is guided by a simplicity heuristic, which estimates how hard the resulting SAT solving problems would be, as described in Section 5.1. In Section 6.1, we connect the different parts of the proofs proving that formulas stated at different logical levels (propositional, first-order and higher-order) imply each other in the desired way.

► **Remark.** Not every algorithm needs to have its computation steps verified in a formal manner. Sometimes, it is enough to verify that the mathematical object produced by the algorithm satisfies the desired properties. For example, we did not verify every step of the nauty algorithm [17] which we rely on to normalize graphs in Section 4. Indeed, it is sufficient to save the witness permutations used during graph normalization to show that two graphs are isomorphic.

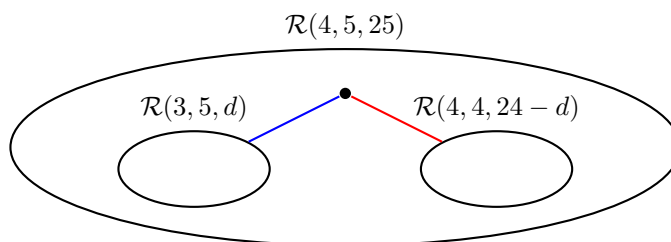
2 Preliminaries

Throughout our proof, we rely on the following definitions.

► **Definition 1** (neighbors, antineighbors).

Given a graph (V, E) :

- the set of neighbors (blue-neighbors) of a vertex $x \in V$ is $\{y \in V \mid y \neq x \wedge (x, y) \in E\}$,
- the set of antineighbors (red-neighbors) of a vertex $x \in V$ is $\{y \in V \mid y \neq x \wedge (x, y) \notin E\}$.



■ **Figure 1** Neighbors (blue-neighbors) and antineighbors (red-neighbors) of a vertex of degree d in an $\mathcal{R}(4, 5, 25)$ -graph.

► **Definition 2** (Ramsey property).

The Ramsey property $\mathcal{R}(n, m, k)$ holds for a graph (V, E) if:

- V has size k ,
- (V, E) does not contain a clique (blue-clique) of size n ,
- (V, E) does not contain an independent set (anticlique, red-clique) of size m .

We also use $\mathcal{R}(n, m, k)$ to refer to the set of graphs with property $\mathcal{R}(n, m, k)$.

A graph for which the property $\mathcal{R}(n, m, k)$ holds is called a $\mathcal{R}(n, m, k)$ -graph.

► **Definition 3** (Ramsey number).

The Ramsey number $R(n, m)$ is the least $k \in \mathbb{N}$ such that $\mathcal{R}(n, m, k)$ is empty.

In our formalization, a set of vertices V will be represented by a subset of nonnegative integers. Moreover, we often use an equivalent formulation of graphs when discussing algorithms on graphs. In the equivalent formulation, all graphs are complete graphs but their edges are colored either blue or red. The correspondence between the two formulations is straightforward. There is a blue edge in the second formulation if and only if there is an edge in the first.

3 Degree Constraints

As an intermediate concept, we define $R^o(r, s, n)$ to hold if $\mathcal{R}(r, s, n)$ is empty. It is easy to see $R(r, s) \leq n$ iff $R^o(r, s, n)$. In our formalization, we are primarily interested in proving $R^o(r, s, n)$ for values of r, s and n . In this section our focus is on reducing the goal of proving $R^o(4, 5, 25)$ to ruling out vertices of degrees 8, 10 or 12. All the lemmas presented in this section are formalized in the file `basicRamsey` of our repository [10]. These lemmas are reformulations of basic results in graph theory [5].

Given a graph (V, E) ¹ and a vertex $v \in V$, we write $\mathcal{N}^{(V, E)}(v)$ for the set of neighbors of v and $\mathcal{A}^{(V, E)}(v)$ for the set of antineighbors of v . We will almost always omit the superscript and write $\mathcal{N}(v)$ and $\mathcal{A}(v)$. The degree of v is defined to be the cardinality of $\mathcal{N}(v)$. Likewise, the antidegree of v is the cardinality of $\mathcal{A}(v)$.

Several relevant smaller Ramsey numbers are well-known: $R(2, s) = s$, $R(3, 3) = 6$, $R(3, 4) = 9$, $R(3, 5) = 14$ and $R(4, 4) = 18$. In our formalization we prove the R^o variant, only proving the known values are upper bounds. We begin by sketching a description of these results as well as some of the preliminary results used to obtain them.

¹ We always implicitly assume the set V is finite.

16:4 A Formal Proof of $R(4,5)=25$

By considering complements of graphs we know that $R^o(r, s, n)$ implies $R^o(s, r, n)$. If $(V, E) \in \mathcal{R}(r+1, s, n)$ and $v \in V$ is a vertex of degree d , then $(\mathcal{N}(v), E) \in \mathcal{R}(r, s, d)$. Likewise, if $(V, E) \in \mathcal{R}(r, s+1, n)$ and $v \in V$ is a vertex with antidegree d , then $(\mathcal{A}(v), E) \in \mathcal{R}(r, s, d)$.

Every graph in $\mathcal{R}(2, s, m)$ has no edges (since an edge would be a 2-clique). Thus every graph in $\mathcal{R}(2, s, m)$ is an independent set of size m . This is impossible if $s \leq m$, and so we conclude $R^o(2, m, m)$. Likewise, $R^o(m, 2, m)$.

We next prove a well-known result that provides upper bounds for values of $R(r, s)$.

► **Lemma 4.** *If $R^o(r+1, s, m+1)$ and $R^o(r, s+1, n+1)$, then $R^o(r+1, s+1, m+n+2)$.*

Proof. Assume we have a graph (V, E) in $\mathcal{R}(r+1, s+1, m+n+2)$. We choose a vertex $v \in V$ with degree d and antidegree d' . We know $(\mathcal{N}(v), E) \in \mathcal{R}(r, s+1, d)$ and $(\mathcal{A}(v), E) \in \mathcal{R}(r+1, s, d')$. We obtain a contradiction using $d + d' = m + n + 1$, $d < m + 1$ (since $R^o(r+1, s, m+1)$) and $d' < n + 1$ (since $R^o(r, s+1, n+1)$). ◀

Applying the previous results, we immediately obtain $R^o(3, 3, 6)$. We also obtain $R^o(3, 4, 10)$, but need the stronger result $R^o(3, 4, 9)$.

There is an easy informal argument for why $\mathcal{R}(3, 4, 9)$ is empty. Assume (V, E) is a graph in $\mathcal{R}(3, 4, 9)$. The results above ensure every vertex $v \in V$ must have degree $d < 4$ (since $R^o(2, 4, 4)$) and antidegree $d' < 6$ (since $R^o(3, 3, 6)$). Since $d + d' = 8$, we must have $d = 3$ and $d' = 5$. We now consider the sum of the degrees of each vertex. Since the relation is symmetric, the sum must be even, as each edge is counted as part of the degree of each of the vertices of the edge. However, the sum is also $9 \cdot 3 = 27$, which is odd. Hence no such graph exists. Below we describe our formalization of general results allowing us to prove $R^o(3, 4, 9)$. The results will also allow us to later prove every graph in $\mathcal{R}(4, 5, 25)$ must have a vertex with even degree.

► **Lemma 5.** *Let (V, E) be a graph in which every vertex has odd degree. For each $U \subseteq V$, U has odd cardinality if and only if $\sum_{u \in U} |\mathcal{N}(u)|$ is odd.*

Proof. The proof follows by an induction over the finite set U . ◀

Applying Lemma 5 with $U = V$, we obtain that if V has odd cardinality and every vertex has odd degree, then $\sum_{v \in V} |\mathcal{N}(v)|$ is odd. In particular for a hypothetical graph $(V, E) \in \mathcal{R}(3, 4, 9)$, $\sum_{v \in V} |\mathcal{N}(v)|$ is odd since 9 and 3 are odd.

On the other hand we can prove $\sum_{v \in V} |\mathcal{N}(v)|$ is always even, though this requires two inductions on finite sets. We first prove that if we extend a graph with a new vertex, the neighbors of the new vertex in the larger graph contribute twice to the sum.

► **Lemma 6.** *Let V be a finite set, $u \notin V$ and E be a symmetric relation (on $V \cup \{u\}$).² For every finite set U , if $\mathcal{N}^{(V \cup \{u\}, E)}(u) = U$, then*

$$\sum_{w \in V \cup \{u\}} |\mathcal{N}^{(V \cup \{u\}, E)}(w)| = \sum_{v \in V} |\mathcal{N}^{(V, E)}(v)| + 2|U|.$$

Proof. This is proved by induction on the finite set U . ◀

We can now prove the sum is even by induction on the finite set of vertices V .

► **Lemma 7.** *For every finite set V and symmetric relation E , $\sum_{v \in V} |\mathcal{N}^{(V, E)}(v)|$ is even.*

² In the formalization, E is assumed to be symmetric on the relevant type, ignoring $V \cup \{u\}$.

With Lemmas 5 and 7 we can conclude $R^o(3, 4, 9)$ since the sum of the degrees of the vertices in a hypothetical graph $(V, E) \in \mathcal{R}(3, 4, 9)$ would be both odd and even.

Using Lemma 4 we now immediately obtain $R^o(3, 5, 14)$ and $R^o(4, 4, 18)$, giving us all the upper bounds for small Ramsey numbers we will need.

We now turn to the consideration of $R^o(4, 5, 25)$. For the next steps in the proof, we assume for the sake of contradiction that there exists a graph $(V, E) \in \mathcal{R}(4, 5, 25)$. Let $v \in V$ with degree d and antidegree d' be given. Since $(\mathcal{N}(v), E) \in \mathcal{R}(3, 5, d)$ and $(\mathcal{A}(v), E) \in \mathcal{R}(4, 4, d')$ we know $d < 14$ and $d' < 18$. Since $d + d' = 24$, we must have $d > 6$. This provides our basic upper and lower bounds on degrees of vertices in (V, E) .

These same degree bounds are, of course, given in [18]. The argument in [18] considers graphs in $\mathcal{R}(3, 5, d)$ and corresponding graphs in $\mathcal{R}(4, 4, 24 - d)$ that could hypothetically correspond to $\mathcal{N}(v)$ and $\mathcal{A}(v)$ for a vertex $v \in V$. In [18], the case with $d = 11$ is ruled out since if every vertex had degree 11, the sum of degrees would be odd, giving a contradiction. That is, we can be assured of the existence of a vertex $v \in V$ with degree $d \in \{7, 8, 9, 10, 12, 13\}$. In our proof, we apply Lemmas 5 and 7 more generally to conclude that there must be a vertex $v \in V$ of even degree. Thus, we can be assured there is a $v \in V$ with degree $d \in \{8, 10, 12\}$.

4 Enumeration of Graphs and Construction of Covers

Assuming that there exists a graph $(V, E) \in \mathcal{R}(4, 5, 25)$, there must exist a vertex $v \in V$ of degree $d \in \{8, 10, 12\}$ as proven in Section 3. Thus, if we prove that for all $d \in \{8, 10, 12\}$ and for all pair of graphs $G \in \mathcal{R}(3, 5, d)$ and $H \in \mathcal{R}(4, 4, 24 - d)$, there is no way to color edges connecting G and H without creating a 4-blue or a 5-red clique, then we would have proved that $R^o(4, 5, 25)$ (i.e. $R(4, 5) \leq 25$).

Here is a simple approach. First, enumerate all the graphs in $\mathcal{R}(3, 5, d)$ and in $\mathcal{R}(4, 4, 24 - d)$, and then prove the absence of gluing between each pair of graphs (see Section 5). This is however not efficient enough given our computational means. In Table 3, we estimated that this approach would take more than 16,000 CPU days. To save time in both algorithms, we regroup graphs that are similar to each other, differing only by a few edges, in what we call *generalizations*. This way, our proofs will avoid repeating the same arguments in similar situations. This idea reduces, with the help of a simplicity heuristic, the total computation time to less than 950 CPU days as shown in Table 3.

From a set of graphs \mathcal{G} , we will construct a set of generalizations \mathcal{G}^* (this is a set of set of graphs) with the following properties. Every graph in \mathcal{G} is a member of a generalization in \mathcal{G}^* (we are not missing any case) and every graph in a generalization $G^* \in \mathcal{G}^*$ is in \mathcal{G} (we are not covering extra cases).

► **Definition 8** (cover, exact cover).

A set of generalizations \mathcal{G}^ is a cover of a set of graphs \mathcal{G} if $\mathcal{G} \subseteq \bigcup_{G^* \in \mathcal{G}^*} G^*$.*

A set of generalizations \mathcal{G}^ is an exact cover of a set of graphs \mathcal{G} if $\mathcal{G} = \bigcup_{G^* \in \mathcal{G}^*} G^*$.*

In a cover the generalizations do not need to be disjoint. Furthermore, our proof does not fundamentally require the constructed covers to be exact and better covers may be obtained by dropping this requirement. Yet, having exact covers simplify our presentation of the gluing algorithm as it enable us to ignore all clique constraints containing the splitting vertex (see Section 5).

4.1 Algorithm for Constructing an Exact Cover

In the following, we describe our base algorithm for constructing an exact cover for a set of graphs \mathcal{G} . Our algorithm differs from the one given in [18] where they decide on which vertex to remove from a graph. This is equivalent in our algorithm to ignoring the color of all edges connecting to that vertex. In contrast, our approach is more targeted and can decide whether to ignore the color of an edge individually. Creating such alternative approach was crucial for us. Indeed, following the original vertex removal method resulted in the creation of SAT problems, which were difficult to reconstruct in HOL4 due to memory issues, negating most of the advantage gained by regrouping graphs.

This will be achieved by incrementally growing a set of generalizations $\mathcal{G}_{partial}^*$. We refer to the set of graphs $G \in \mathcal{G}$ that are not currently covered by $\mathcal{G}_{partial}^*$ as $\mathcal{G}_{uncovered}$. Initially, $\mathcal{G}_{partial}^*$ is empty and thus $\mathcal{G}_{uncovered}$ is equal to \mathcal{G} .

At each iteration of our algorithm, we randomly pick a graph G from $\mathcal{G}_{uncovered}$ constructing a singleton generalization $G_0^* = \{G\}$. Then, we color one of the edges of G gray. This represents a generalization G_1^* that contains the two graphs obtained by coloring the gray edge red or blue. Note that one of this graph is G and thus $G \in G_1^*$ and $G_0^* \subseteq G_1^*$. In general, the process starts from a generalization G_n^* represented as a graph with n gray edges. By definition, the generalization G_n^* is defined to be the set of all graphs that can be obtained by coloring its n gray edges red or blue in its representation. Then, the algorithm selects randomly one edge to gray among edges respecting the following conditions: the produced generalization G_{n+1}^* must only contain graphs that are in \mathcal{G} and $(G_{n+1}^* \setminus G_n) \cap \mathcal{G}_{uncovered}$ must contain at least $\lceil 2^{n-3} \rceil$ graphs. The first condition makes the cover exact and the second condition prevents large overlaps between generalizations. The coefficient $\lceil 2^{n-3} \rceil$ was experimentally determined and essentially ensures that at least $\frac{1}{8}$ of the covered graphs by the newly created generalization are not covered by previous generalizations.

This process is repeated graying one more edge per generalization step, as illustrated in Figure 2. It stops when the number of gray edges exceeds a user-given limit or when there are no more edges respecting the conditions.

When the generalization algorithm stops, it creates a maximal generalization G_{max}^* which is added to the set of generalizations $\mathcal{G}_{partial}^*$ and the instantiations of G_{max}^* are removed from the set $\mathcal{G}_{uncovered}$. We keep adding new generalizations to $\mathcal{G}_{partial}^*$ by the same procedure until the set $\mathcal{G}_{uncovered}$ is empty and therefore $\mathcal{G}_{partial}^*$ is an exact cover of \mathcal{G} .

We can reduce the size of the final cover \mathcal{G}^* by sampling multiple graphs in $\mathcal{G}_{uncovered}$ at each iteration of the algorithm. In our implementation, we sample 1000 graphs when $\mathcal{G} = \mathcal{R}(4, 4, k)$ and all graphs when $\mathcal{G} = \mathcal{R}(3, 5, k)$. This produces one maximal generalization for each of those graphs. We then select among them a generalization G^* that contains a maximum number of uncovered graphs. That is to say one for which $|G^* \cap \mathcal{G}_{uncovered}|$ is maximum. We call this strategy for selecting generalization the *greedy cover* strategy. Our final strategy for constructing covers, described in Section 5.2, is a blend of the *greedy cover* strategy and a strategy that minimizes the difficulty of resulting problems with respect to a simplicity heuristic given in Section 5.1.

The nauty algorithm [17] is called to normalize graphs in \mathcal{G} and in each generalization G^* . By normalizing all graphs, we can check that two graphs are isomorphic by simply checking if their normalizations are equal.

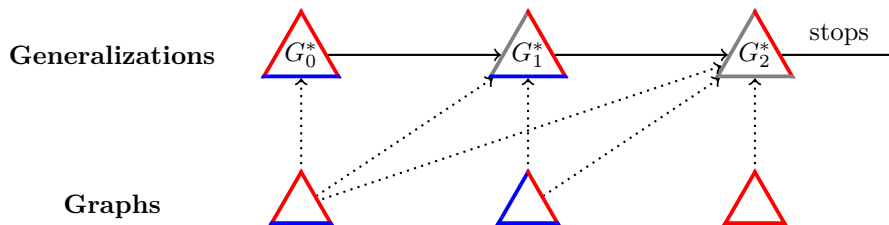
Computing the lists of $\mathcal{R}(3, 5, k)$ -graphs and $\mathcal{R}(4, 4, k)$ -graphs up-to-isomorphism can be done efficiently by simply repeatedly extending graphs in $\mathcal{R}(3, 5, k)$ (resp. $\mathcal{R}(4, 4, k)$) by one vertex while respecting the clique constraints to produce $\mathcal{R}(3, 5, k+1)$ (resp. $\mathcal{R}(4, 4, k+1)$). Such lists have been repeatedly compiled as mentioned in [18] and therefore we will not

■ **Table 1** Number of $\mathcal{R}(3, 5, k)$ -graphs and $\mathcal{R}(4, 4, k)$ -graphs up-to-isomorphism together with the number of generalizations in the respective covers. All the covers were initially constructed with a maximum of 10 gray edges. We later updated the cover for the bold cases using an edge selection algorithm and an improved selection algorithm for generalizations (see Section 5.2).

k	$\mathcal{R}(3, 5, k)$	$\mathcal{R}^*(3, 5, k)$	$\mathcal{R}(4, 4, k)$	$\mathcal{R}^*(4, 4, k)$
1	1		1	
2	2		2	
3	3		4	
4	7		9	1
5	13	3	24	3
6	32	3	84	6
7	71	5	362	11
8	179	27	2079	47
9	290	11	14701	271
10	313	43	103706	1669
11	105	12	546356	7919
12	12	12	1449166	26845
13	1	1	1184231	13078
14			130816	11752
15			640	67
16			2	2
17			1	1

Iteration 0: $\mathcal{G}_{uncovered} = \mathcal{G} = \{ \triangle, \triangle, \triangle \}$, $\mathcal{G}_{partial}^* = \emptyset$

Randomly chosen generalization $G_0^* = \{ \triangle \}$



Iteration 1: $\mathcal{G}_{uncovered} = \emptyset$, $\mathcal{G}^* = \mathcal{G}_{partial}^* = \{ G_2^* \} = \{ \triangle \}$

■ **Figure 2** Construction of an exact cover \mathcal{G}^* of a set of graphs \mathcal{G} . The process of graying edges stops as it would otherwise produce a gray triangle including a blue triangle. The construction of an exact cover terminates in this case after one iteration. The dotted arrows indicate which graph belongs to which generalization.

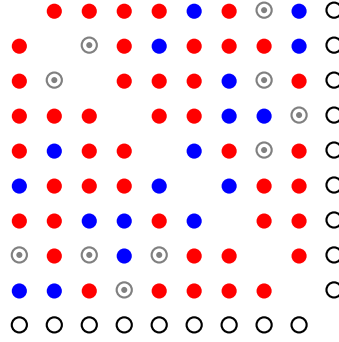
discuss in more detail how to construct them. From those lists, we construct corresponding covers $\mathcal{R}^*(3, 5, k)$ and $\mathcal{R}^*(4, 4, k)$. The size of those constructed covers for the sets of graphs $\mathcal{R}(3, 5, k)$ with $5 \leq k \leq 13$ and the sets of graphs $\mathcal{R}(4, 4, k)$ with $4 \leq k \leq 17$ is presented in Table 1.

4.2 Proof that $\mathcal{R}(3, 5, k)$ is Covered by $\mathcal{R}^*(3, 5, k)$

In this section, we only present the proof for the covers $\mathcal{R}^*(3, 5, k)$ since the proof for the covers $\mathcal{R}^*(4, 4, k)$ follows by an analogous argument. Given the result presented in Section 3, it is enough to consider the cases of a splitting vertex with degree $d \in \{8, 10, 12\}$. Therefore, it would be enough to prove that $\mathcal{R}^*(3, 5, d)$ covers the set of graphs with property $\mathcal{R}(3, 5, d)$. However, to do so, we found it easier to prove the stronger result:

$$\forall 5 \leq k \leq 13. G \text{ has property } \mathcal{R}(3, 5, k) \Rightarrow \exists G^* \in \mathcal{R}^*(3, 5, k). G \in G^*$$

We prove the result by a finite induction over the number of vertices k . The base case $k = 5$ consists of searching for all the possible graphs with property $\mathcal{R}(3, 5, 5)$ and show that they appear modulo isomorphism in one of the generalizations in $\mathcal{R}^*(3, 5, 5)$. The inductive case is similar. The main difference is that we start the search from a generalization G^* instead of the empty generalization. We prove that for all generalizations G^* in $\mathcal{R}^*(3, 5, k)$, any extension of G^* by one vertex that respects the property $\mathcal{R}^*(3, 5, k + 1)$ is isomorphic to an element of $\mathcal{R}^*(3, 5, k + 1)$. This is achieved by exploring all possible colorings (in blue or in red) of edges that are either gray or contain the new vertex. This extension process is depicted in Figure 3.



■ **Figure 3** Extension of an $\mathcal{R}^*(3, 5, 9)$ -generalization depicted as an adjacency matrix. The first 9 rows and columns represent the vertices x_0 to x_8 of the $\mathcal{R}^*(3, 5, 9)$ -generalization. Gray edges are represented by dotted gray circles. Edges containing the extension vertex x_9 (last row and column) are represented by black circles.

The formalization and efficiency of the previous arguments rely on our custom-made solver for labeled graphs. Our solver mostly works like a DPLL SAT solver [8]. The principal difference is that it represents clauses as essentially first-order formulas. We show how we represent the property $\mathcal{R}^*(3, 5, k + 1)$ in first-order. Given a graph G of size $k + 1$, represented by a binary relation E over a set of vertices $V = [0, k] = \{0, 1, \dots, k\}$, our first-order representation of the statement “ G has property $\mathcal{R}^*(3, 5, k + 1)$ ” is given by the two formulas:

$$\begin{aligned} & \forall_{\substack{x_0 x_1 x_2 \\ \text{distinct}}} < k + 1. \neg E x_0 x_1 \vee \neg E x_0 x_2 \vee \neg E x_1 x_2 \\ & \forall_{\substack{x_0 x_1 x_2 x_3 x_4 \\ \text{distinct}}} < k + 1. E x_0 x_1 \vee E x_0 x_2 \vee E x_0 x_3 \vee E x_0 x_4 \vee E x_1 x_2 \vee \\ & \quad E x_1 x_3 \vee E x_1 x_4 \vee E x_2 x_3 \vee E x_2 x_4 \vee E x_3 x_4 \end{aligned}$$

where *distinct* means that we add inequalities between each of pair of quantified variables. For example, $\forall_{\substack{x_0 x_1 x_2 \\ \text{distinct}}} < k + 1. P[x_1, x_2, x_3]$ stands for:

$$\forall x_0 x_1 x_2. (x_0 < k + 1 \wedge x_1 < k + 1 \wedge x_2 < k + 1 \wedge x_0 \neq x_1 \wedge x_0 \neq x_2 \wedge x_1 \neq x_2) \Rightarrow P[x_1, x_2, x_3]$$

Our solver is designed to work exclusively on graphs where edges are represented by first-order literals and the coloring of an edge (i, j) to blue (resp. red) corresponds to assuming the literal $E_{x_i x_j}$ in the branch (resp. the negation of the literal $\neg E_{x_i x_j}$). A gray edge just indicates that no color for that edge is currently assumed. The prover starts by assuming all literals corresponding to colored edges in a generalization G^* with k vertices. It then explores all possible colorings of the gray edges and then all the possible coloring of the edges containing a new vertex x_k . At the leaf, this produces a graph of size $k + 1$ represented by all the literals assumed in the branch.

By representing vertices by variables x_0, \dots, x_k instead of the concrete value $0, \dots, k$, it is easier to prove that the graphs in the leaves are indeed isomorphic to one of the elements of one of the generalizations $G' \in \mathcal{R}^*(3, 5, k + 1)$. To prove that every generalization in $\mathcal{R}^*(3, 5, k)$ extends to graphs belonging to a generalization in $\mathcal{R}^*(3, 5, k + 1)$, we will prove that there is no way to extend a generalization in $\mathcal{R}^*(3, 5, k)$ if we forbid the creation of any graph that is a member of a generalization in $\mathcal{R}^*(3, 5, k + 1)$. Given a generalization $G' \in \mathcal{R}^*(3, 5, k + 1)$ with a set of blue edges *Blue* and a set of red edges *Red*, we use the following formula to forbid the creation of an element of G' :

$$\forall x_0 \dots x_i \dots x_j \dots x_k < k + 1. \left(\bigwedge_{(i,j) \in \text{Blue}} E_{x_i x_j} \wedge \left(\bigwedge_{(i,j) \in \text{Red}} \neg E_{x_i x_j} \right) \right) \Rightarrow \perp$$

Note that this implies that all permutations of graphs that are members of this generalization are forbidden. The first reason is that the formula does not assume any constraints on the gray edges of G' therefore forbids all members of G' . The second reason is one can permute the indices of variables by a simple instantiation of the variables with a permutation being given by the nauty algorithm to make the labeled graph on the branch match with one of the labeled generalizations.

5 Gluing

In the previous section, we constructed covers for $\mathcal{R}(3, 5, d)$ -graphs and $\mathcal{R}(4, 4, 24 - d)$ -graphs. The next step of our proof is to prove that given a generalization G^* in $\mathcal{R}^*(3, 5, d)$ and a generalization in H^* in $\mathcal{R}^*(3, 5, 24 - d)$, there is no way to extend color gray edges and transverse edges to form an $\mathcal{R}(4, 5, 24)$ -graph (see Figure 4) and thus an $\mathcal{R}(4, 5, 25)$ -graph by adding the splitting vertex. In the rest of this section, we can ignore clique constraints that include the splitting vertex as they are already satisfied. This is a consequence of the fact that our covers are exact covers. All our gluing problems are formulated at the propositional level and contain the following clauses representing the property $\mathcal{R}(4, 5, 24)$. Let us number the vertices of an $\mathcal{R}^*(3, 5, d)$ -generalization G^* from 0 to $d - 1$, and the vertices of a $\mathcal{R}^*(4, 4, 24 - d)$ -generalization graph H^* from d to 23.

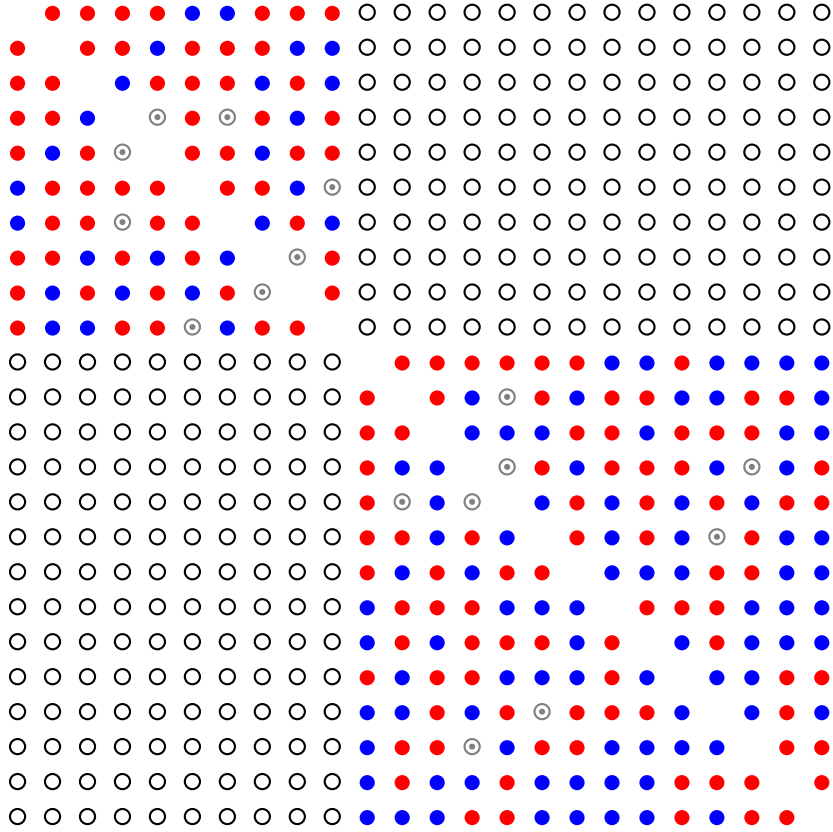
For each subset $S \subset [0, 23]$ of size 4, we create the clause $\bigvee_{a,b \in S \wedge a < b} \neg E_{a,b}$.

For each subset $T \subset [0, 23]$ of size 5, we create the clause $\bigvee_{a,b \in T \wedge a < b} E_{a,b}$.

In all these propositional clauses, $E_{a,b}$ is a propositional variable that is true if there is a blue edge between a and b and that is false if there is a red edge between a and b . One can note that any clauses containing only vertices from G^* or only vertices from H^* can be omitted as G^* and H^* provably avoid any blue 4-clique or any red 5-clique. This removal occurs naturally as a consequence of performing unit propagation. In each gluing problem

16:10 A Formal Proof of $R(4,5)=25$

(G^*, H^*) , we add unit clauses for each colored edge (red or blue) of G^* and H^* . If an edge (a, b) with $a < b$ is blue then we add the unit clause $E_{a,b}$, if it is red then we add the unit clause $\neg E_{a,b}$. If an edge is gray we do not add a unit clause. Together, with the clique clauses this forms our SAT problem that is sent to the MiniSat interface. In practice, we had to perform unit propagation to reduce the number of clauses before sending a problem to the interface. This is due to some limitations in the interface as this does not happen when we call the SAT solver directly.



■ **Figure 4** The adjacency matrix of a graph of size 24 where a partial coloring is given by a generalization G^* with 4 gray edges (dotted gray circles) with vertices numbered from 0 to 9 and a generalization H^* with 4 gray edges with vertices numbered from 10 to 23. The goal of the SAT solvers is to prove that there is no way to assign a color (blue or red) to the gray edges and the transverse edges (black circles) without creating a blue 4-clique or a red 5-clique.

5.1 Simplicity Heuristic

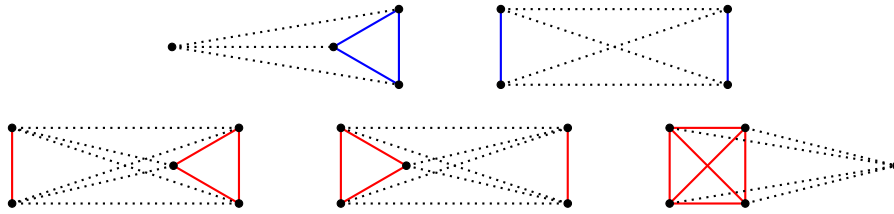
In this section, we design a heuristic that will be used to construct covers resulting in easier problems for the SAT solver. Let b_k represent the number of blue k -cliques in G^* . Let r_k represent the number of red k -cliques in G^* . Let b'_k represent the number of blue k -cliques in H^* . Let r'_k represent the number of red k -cliques in H^* . We use the following formula to estimate the difficulty of a gluing problem (G^*, H^*) :

$$simplicity(G^*, H^*) = \frac{1}{2^3} b_1 b'_3 + \frac{1}{2^4} b_2 b'_2 + \frac{1}{2^6} r_2 r'_3 + \frac{1}{2^6} r_3 r'_2 + \frac{1}{2^4} r_4 r'_1$$

Our formula is originally designed to estimate the simplicity of a problem of gluing an $\mathcal{R}(3, 5, d)$ -graph G with an $\mathcal{R}(4, 4, 24 - d)$ -graph H . There, 5 different types of configurations that may create a blue 4-clique or a 5 red-clique as illustrated in Figure 5. In the resulting SAT solving problem after unit propagation, a clause mentioning only the transverse edges is associated with each configuration. The above heuristic can be derived from a more general heuristic for a SAT problem P :

$$simplicity(P) = \sum_{c \in P} \frac{1}{2^{|c|}}$$

where $|c|$ is the number of literals in a clause c . This heuristic operates under the simplistic assumption that each clause covers separate cases, allowing it to estimate the extent of search space coverage by summing up the contribution of each clause. Experimentally, we found that this heuristic is only effective to compare problems with the same number of variables. Consequently, we ignore clauses containing gray edges from all gluing problems when computing their simplicity, as we will use this heuristic to compare generalizations with varying numbers of gray edges. Another advantage of ignoring clauses containing gray edges is that the heuristic will prefer problems that can delay splitting on the color of gray edges as much as possible, which favors proof sharing.



■ **Figure 5** The five possible types of configurations. In each configuration, a colored clique in an $\mathcal{R}(3, 5, d)$ -graph is displayed on the left and a colored clique in an $\mathcal{R}(4, 4, 24 - d)$ -graph is displayed on the right. Transverse edges are shown as dotted black edges. Transverse edges must not all be blue in blue configurations and they must not all be red in red configurations.

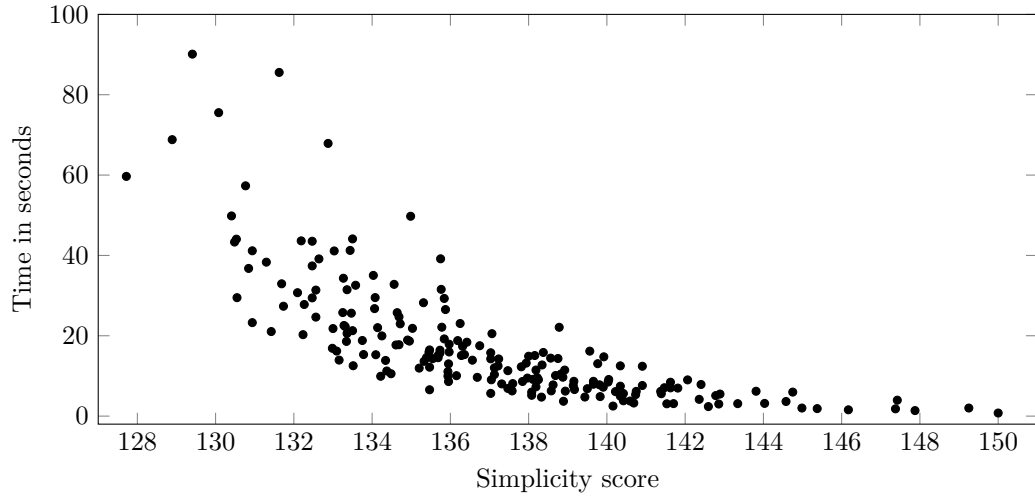
We now test how good the simplicity score is at predicting the run time of MiniSat via the HOL4 interface on 200 gluing problems between $\mathcal{R}(3, 5, 10)$ -graphs and $\mathcal{R}(4, 4, 14)$ -graphs in Figure 6. The results reveal that our simplicity score is a good predictor in this setting.

Finally, during the construction of a cover for \mathcal{G} we are not aware of the corresponding cover for \mathcal{H} . The covers would have to be built simultaneously making the algorithm more complicated. To avoid those complications, we devise a measure to predict if a generalization will create difficult problems on its own without depending on the possible counterparts. To this end, we chose to estimate the simplicity of a generalization G^* by how difficult it is to glue it with an average counterpart graph \bar{H} . Let \bar{b}'_k represent the average number of blue k -cliques per graph in \mathcal{H} and \bar{r}'_k represent the average number of red k -cliques per graph in \mathcal{H} , the simplicity of G^* is:

$$simplicity(G^*) = \frac{1}{2^3} b_1 \bar{b}'_3 + \frac{1}{2^4} b_2 \bar{b}'_2 + \frac{1}{2^6} r_2 \bar{r}'_3 + \frac{1}{2^6} r_3 \bar{r}'_2 + \frac{1}{2^4} r_4 \bar{r}'_1$$

Similarly, let \bar{b}_k represent the average number of blue k -cliques per graph in \mathcal{G} and \bar{r}_k represent the average number of red k -cliques per graph in \mathcal{G} , the simplicity of H^* is:

$$simplicity(H^*) = \frac{1}{2^3} \bar{b}_1 b'_3 + \frac{1}{2^4} \bar{b}_2 b'_2 + \frac{1}{2^6} \bar{r}_2 r'_3 + \frac{1}{2^6} \bar{r}_3 r'_2 + \frac{1}{2^4} \bar{r}_4 r'_1$$



■ **Figure 6** Relation between the simplicity score of a problem and the time required by the HOL4 interface to MiniSat to prove that it is unsatisfiable. Each problem consists of an $\mathcal{R}(3, 5, 10)$ -graph and an $\mathcal{R}(4, 4, 14)$ -graph. Each dot represents one problem among a random sample of 200 gluing problems.

5.2 Creation of Better Covers by Parameter Search

We now improve the exact cover algorithm by relying on our simplicity heuristic in two places. The first one is during the selection of edges. Previously, the edges were selected at random as long as the produced generalization satisfied some conditions. Now, we will select, among the possible edges allowed by the conditions, the one that produces the generalization with the highest simplicity score. We call this strategy *fastest* in Table 2. The second place where the simplicity score will influence the algorithm is during the selection of generalizations. Previously, we selected maximal generalizations G^* with highest coverage value $n_{cover}(G^*) = |G^* \cap \mathcal{G}_{uncover}|$. This strategy was called the *greedy cover* strategy. Now, we will also prefer generalizations with higher simplicity scores. Since we want to optimize for both objectives at the same time, we will select the graph G^* with the highest combined score $simplicity(G^*)^c \times n_{cover}(G^*)$ where c is a real number parameter influencing how much one heuristic is preferred over the other. In Table 2, we call this selection strategy *mixed-c*.

Although the simplicity score is important to reduce the difficulty of the problems, the most important parameter in reducing the total computation time is the number of maximum allowed gray edges in each generalization. In Table 2, we optimize those parameters for the three relevant cases $d = 8, 10, 12$ for the gluing. Each experiment consists of a line in Table 2. There, we compute new covers with different parameters. To figure out the best parameters, we estimate the run time of an average problem by sampling 200 random problems from a pair of covers. We then multiply this estimate by the number of problems this pair of covers would create to get an estimated total run time for this pair of covers. In the end, we decided not to go with the best parameters according to the estimated times given in Table 2. The reason is that by increasing the number of gray edges, the total number of problems is reduced but the difficulty of each problem is increased. This makes the problems harder and they would have taken more memory than we had available. That is why we chose to compromise and instead use the fastest parameter settings, shown in bold in Table 2, that would not use more memory than available on our machines. Table 3 gives a comparison

of the total run time for our final problems with the total runtime that we would get by simply gluing pairs of graphs instead of generalizations. For degree $d = 10$, the number of gluing problems is reduced by a factor of 81.0 and the estimated total time by a factor 14.6. For degree $d = 12$, the number of gluing problems is reduced by a factor of 53.9 and the estimated total time by a factor of 20.6.

■ **Table 2** Tested parameters for creating exact covers. The columns titled 3,5 and 4,4 show the maximum number of gray edges allowed during the construction of the cover.

Gluing	3,5	4,4	Edge sel.	Gen. sel.	CPU-days (estimation)
3,5,8-4,4,16	0	0	none	none	0.055
	4	0	fastest	mixed-0.5	0.018
3,5,10-4,4,14	0	0	none	none	8373
	3	4	fastest	mixed-1.0	725
	4	3	fastest	mixed-1.0	689
	4	4	random	greedy cover	734
	4	4	fastest	mixed-10.0	625
	4	4	fastest	mixed-2.0	595
	4	4	fastest	mixed-1.0	658
	4	4	fastest	mixed-0.5	572
	4	4	fastest	mixed-0.1	706
	5	4	fastest	mixed-1.0	547
	4	5	fastest	mixed-1.0	586
	5	5	fastest	mixed-0.5	396
	3,5,12-4,4,12	0	0	none	none
2		6	fastest	mixed-0.5	641
3		6	fastest	mixed-0.5	782
4		6	fastest	mixed-0.5	784
0		8	fastest	mixed-0.5	374
1		8	fastest	mixed-0.5	353
2		8	fastest	mixed-0.5	538
3		8	fastest	mixed-0.5	360
4		8	fastest	mixed-0.5	419

6 Combining the Different Parts of the Proof

Our proof is expressed using three different formal representations of mathematical statements. In Section 3, we express our statements in a higher-order form allowing us to make counting arguments. In Section 4.2, we rely on an almost first-order representation to implement a custom theorem prover for graphs and in particular to prove isomorphism between graphs. In Section 5, the problems are stated at the propositional level. Here, we first describe how we connect the different representations and as a consequence prove that $R(4, 5) \leq 25$. Then, we give the proof of the existence of an $\mathcal{R}(4, 5, 24)$ -graph and show that $R(4, 5) > 24$.

■ **Table 3** Reduction of the number of SAT solver calls and faster estimated times in days.

d	Graphs				Generalizations			
	3,5,d	4,4,24-d	problems	days	3,5,d	4,4,24-d	problems	days
8	179	2	358	0.055	27	2	54	0.018
10	313	130816	40945408	8373	43	11752	505336	572
12	12	1449166	17389992	7702	12	26845	322140	374

6.1 Connecting Representations

We will start by translating propositional gluing lemmas to first-order formulas. The SAT problems do not explicitly mention on which vertices the graphs are lying on since they are only constraining SAT variables that represent edges. Surprisingly, one can instantiate the SAT variables $E_{i,j}$ by the atom $E x_i x_j$ in the gluing problem. As a consequence, gluing problems for all permutations of edges are proved at once. We can also freely add the following additional constraints. All variables x_i must be distinct and variables with indices less than the degree d must have a value less than d and other variables must have a value greater or equal to d . This ensures that our generalizations G^* and H^* have distinct sets of vertices. We then prove that for each $\mathcal{R}^*(3, 5, d)$ -generalization a single theorem stating that this particular generalization can not be glued to any of the corresponding generalizations in $\mathcal{R}^*(4, 4, 24 - d)$ by regrouping gluing theorems. This step constructs 27 theorems for degree $d = 8$, 43 theorems for degree $d = 10$, and 12 theorems for degree $d = 12$. These numbers correspond to the number of $\mathcal{R}^*(3, 5, d)$ -generalizations presented in Table 1. Together, these 27 theorems (respectively 43 and 12) can be used to prove a theorem stating that the splitting edge, represented by the vertex number 24, cannot have degree 8 (respectively 10 and 12). The higher-order version of these three final theorems do not state on which set of vertices the neighbors and antineighbors should lie although it requires them to form sets of nonnegative integers of size d and $24 - d$ respectively. To prove the more general higher-order formulations, we rely on the fact that there is a bijection from $[0, d - 1]$ to sets of vertices of size d and a bijection from $[d, 23]$ to sets of vertices of size $24 - d$. These three theorems together are enough to prove that $R(4, 5) \leq 25$ according to the proof given in Section 3.

6.2 Existence of an $\mathcal{R}(4, 5, 24)$ -graph

To prove the existence of an $\mathcal{R}(4, 5, 24)$ -graph, we pick a graph from the full list of $\mathcal{R}(4, 5, 24)$ -graphs compiled in 2016 and available at [2]. This was necessary step to prove that $R(5, 5) \leq 48$ as described in [1]. For our purpose, we only need one arbitrary witness graph G_0 from that list. Let B be the set of blue edges in G_0 , we represent the graph G_0 as the relation:

$$E_0 =_{def} \lambda i j. \bigvee_{(a,b) \in B \wedge a < b} (i = a \wedge j = b) \vee (j = a \wedge i = b)$$

We prove on the first-order level that this graph does not contain any blue 4-cliques or any red 5-cliques which can be stated as:

$$\begin{aligned} &\vdash \forall x_0 x_1 x_2 x_3 < 24. \neg E_0 x_0 x_1 \vee \neg E_0 x_0 x_2 \vee \neg E_0 x_0 x_3 \vee \neg E_0 x_1 x_2 \vee \neg E_0 x_1 x_3 \vee \neg E_0 x_2 x_3 \\ &\quad \text{distinct} \\ &\vdash \forall x_0 x_1 x_2 x_3 x_4 < 24. E_0 x_0 x_1 \vee E_0 x_0 x_2 \vee E_0 x_0 x_3 \vee E_0 x_0 x_4 \vee E_0 x_1 x_2 \vee \\ &\quad \text{distinct} \\ &\quad E_0 x_1 x_3 \vee E_0 x_1 x_4 \vee E_0 x_2 x_3 \vee E_0 x_2 x_4 \vee E_0 x_3 x_4 \end{aligned}$$

This was achieved by repeatedly applying the following lemma to eliminate the quantified variables $\vdash (\bigwedge_{x < 24} P(x)) \Rightarrow (\forall x < 24. P(x))$. To speed up the process, we stop applying the lemma as soon as we were able to prove the goal on the branch either because we find a red edge in blue clique (or a blue edge in a red clique) or because we have selected the same vertex twice in the clique. With these optimizations, the existence of a graph can be verified in less than 15 minutes on a single CPU. The connection with the higher-order formulation can be obtained by proving that the set $[0, 23]$ has cardinality 24. And thus we get $R(4, 5) > 24$ which together with $R(4, 5) \leq 25$ gives:

$$\vdash R(4, 5) = 25$$

7 Reproducibility

We provide instructions on how to reproduce the proof in the `README.md` of our repository [10].

The computational resources necessary to run our proof are the following. The final gluing step was run on 4 different machines allowing us to finish the gluing phase in less than 9 days. This is slightly longer than what we expected according to the estimated times. Two machines were used for the $d = 10$ case and the other 2 were used for the $d = 12$ case. Three of them have 512 GB of RAM and one of them has 1024 GB of RAM. All of those machines have 64 hyper-threaded CPU cores for a total of 128 available CPUs. However, we only used 40 cores per machine as our main limitation was memory. In all those machines, the same copy of HOL4 was used to get compatible timestamps in the produced gluing theories. The other phases were much faster and required less memory but were still run on the machine with more memory.

Potential issues one could encounter when trying to reproduce the proof are the following. As expected, all the technical problems come during or after the expensive gluing phase. The first issue we discovered is that the communication files between HOL4 and MiniSat are stored in the temporary directory of the system. Since the proof file produced by MiniSat can be up to 2GB and our temporary directory sits in a partition of only 32GB, we ran out of memory in that partition because we were running 40 processes in parallel. So, we changed the temporary directory used by the HOL4 interface to MiniSat by modifying the file `dimacsTools.sml`. We changed the temporary directory used by MiniSat using the `TMPDIR` bash variable. The second issue is that the reconstruction of a SAT proof in HOL4 can require a lot of memory (in the order of 20GB per problem) and time (about 3 times longer than the SAT solver call). We found out that creating theories with one theorem per theory diminished memory consumption. To guarantee that the memory consumption did not exceed a threshold, we also ran the scripts using `buildheap` instead of `Holmake` as the latter does not provide a way to limit the memory consumption per core. Finally, we were not able to load all the gluing theories together into HOL4 in a reasonable time. Indeed, we observed a slowdown of time taken to load one theory as more and more theories were added making it impossible for us to load more than 200,000 theories. Thus, we used the following workaround instead. We loaded the theorems produced by the 43 theorems for degree $d = 10$ and the 12 theorems for degree $d = 12$, mentioned in Section 6.1, without specifying in which theory they were proved. This was achieved by creating an alternative theory loader where we omitted the call to the function `link_parents`. To ensure the safety of this procedure, we externally check that constructing the complete theory graph for our proof without any broken dependencies is possible in the directory `theorygraph`. There, we make sure that the time stamps are coherent and that there are no cycles. In the complete theory graph, the final theory `r45_equals_24` has 828857 ancestor theories.

8 Related Works

Our formalization is based on the work of McKay and Radziszowski [18]. Their proof already contains the three steps performed in our formalization, namely: the degree constraints for a splitting, the creation of covers, and the proof of the absence of satisfiable gluing problems. We explain here how our proof differs in each of those steps. In the first step, their proof mentions that it is not possible for all vertices to have degree 11. We realized during the formalization that this argument can be applied to prove that it is not possible for all vertices to have odd degrees. This allows us to save time during the formalization compared to their original proof by additionally ignoring the cases of a splitting vertex of degree $d \in \{7, 9, 13\}$. In the second step, their proof removes vertices from graphs to create generalizations which would correspond to graying all edges connected to the removed vertices in our formalization. We tried this approach but the large number of grayed edges made the problems very difficult for the SAT solvers. Moreover, our tests gave an estimated time to completion much larger than with our more parsimonious approach to graying edges. In the final step, they rely on a custom provers for gluing generalizations and spend a large part of the paper describing how they optimize its different components. In contrast, we perform the gluing step with an existing SAT solver. Furthermore, they state that they prefer sparser (redder) generalizations when constructing a cover of $\mathcal{R}(3, 5, 10)$ -graphs and denser (bluer) generalizations when constructing a cover of $\mathcal{R}(4, 4, 12)$ -graphs. Our approach instead relies on a more involved heuristic. We try to estimate the simplicity of a generalization G^* by understanding which type of clauses would appear in a gluing problem where one of the generalizations is G^* .

Proving mathematical theorems in combinatorics with the help of SAT solvers is not a new phenomenon. For instance in [6], the authors prove using SAT solvers that the 3-color Ramsey number $R(4,3,3)$ is equal to 30. This proof contains consideration about the degrees of vertices, a symmetry-breaking component to avoid considering isomorphic graphs and an abstraction component relying on degree matrices to represent sets of graphs. Ultimately, the approach relies on encoding each of these steps into SAT clauses and calling a SAT solver. A more famous example is the proof of the Pythagorean Triple theorem [15]. In that paper, the authors rely on a cube-and-conquer, look-ahead methods and symmetry-breaking arguments. The DRAT proof produced by their custom SAT solver was verified using an independent DRAT proof checker [21]. In a later work, the proof of the Pythagorean Triple theorem has been fully formalized in Coq [7]. There, they verified symmetry-breaking arguments and an encoding of the problem into SAT. For the computational part of the proof, they relied on OCaml code proven correct in Coq. This last step was necessary because of memory limitations but is generally considered slightly less safe than running the computation directly in Coq. In contrast, we were able to run the entire proof of $R(4, 5) = 25$, which is larger in size (approximately a petabyte of proof files produced by MiniSat versus 200 terabytes), through the HOL4 kernel.

9 Conclusion

We have created the first formalization of the theorem $R(4, 5) = 25$. This verification was performed within the HOL4 theorem prover. During this process, we have realized that we can generalize the argument given for degree $d = 11$ to eliminate all odd degree cases. We have designed a verified algorithm for regrouping similar graphs, creating generalizations and ultimately speeding up our subsequent proofs. Finally, we have created and tested a heuristic for predicting the relative run-time of a SAT solver on gluing problems. This helped us choose generalizations that create easier gluing problems.

In the future, we would like to investigate if a proof of $R(4, 5) = 25$ is intrinsically computational or if there exist additional high-level arguments that could eliminate the need for a large computation.

References

- 1 Vigeik Angelteit and Brendan D. McKay. $R(5, 5) \leq 48$. *J. Graph Theory*, 89(1):5–13, 2018. doi:10.1002/JGT.22235.
- 2 Vigeik Angelteit, Brendan D. McKay, and Stanislaw P. Radziszowski. Website section: All maximal Ramsey(4,5)-graphs. <https://users.cecs.anu.edu.au/~bdm/data/ramsey.html>, 2016. Accessed on May 26, 2024.
- 3 Kenneth I Appel and Wolfgang Haken. *Every planar map is four colorable*, volume 98. American Mathematical Soc., 1989.
- 4 Yves Bertot. A short presentation of Coq. In Otmane Ait Mohamed, César Muñoz, and Sofiène Tahar, editors, *Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, volume 5170 of *LNCS*, pages 12–16. Springer, 2008. URL: http://doi.org/10.1007/978-3-540-71067-7_3.
- 5 Béla Bollobás. *Modern Graph Theory*, volume 184 of *Graduate Texts in Mathematics*. Springer, 2002. doi:10.1007/978-1-4612-0619-4.
- 6 Michael Codish, Michael Frank, Avraham Itzhakov, and Alice Miller. Computing the Ramsey number $R(4, 3, 3)$ using abstraction and symmetry breaking. *Constraints An Int. J.*, 21(3):375–393, 2016. doi:10.1007/S10601-016-9240-3.
- 7 Luís Cruz-Filipe, João Marques-Silva, and Peter Schneider-Kamp. Formally verifying the solution to the Boolean Pythagorean triples problem. *J. Autom. Reason.*, 63(3):695–722, 2019. doi:10.1007/S10817-018-9490-4.
- 8 Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, 1960. doi:10.1145/321033.321034.
- 9 Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003. doi:10.1007/978-3-540-24605-3_37.
- 10 Thibault Gauthier and Chad E. Brown. Software accompanying the paper "A formal proof of $R(4,5)=25$ ". <https://github.com/barakeel/ramsey>, 2024. Accessed on May 26, 2024.
- 11 Georges Gonthier. Formal proof the four-color theorem. *Notices of the AMS*, 55(11):1382–1393, 2008. URL: <http://www.ams.org/notices/200811/tx081101382p.pdf>.
- 12 Thomas C. Hales, Mark Adams, Gertrud Bauer, Dat Tat Dang, John Harrison, Truong Le Hoang, Cezary Kaliszyk, Victor Magron, Sean McLaughlin, Thang Tat Nguyen, Truong Quang Nguyen, Tobias Nipkow, Steven Obua, Joseph Pleso, Jason M. Rute, Alexey Solovyev, An Hoai Thi Ta, Trung Nam Tran, Diep Thi Trieu, Josef Urban, Ky Khac Vu, and Roland Zumkeller. A formal proof of the Kepler conjecture. *CoRR*, abs/1501.02155, 2015. arXiv: 1501.02155.
- 13 Thomas C. Hales and Samuel P. Ferguson. A formulation of the Kepler conjecture. *Discret. Comput. Geom.*, 36(1):21–69, 2006. doi:10.1007/S00454-005-1211-1.
- 14 John Harrison. HOL Light: An overview. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, volume 5674 of *LNCS*, pages 60–66. Springer, 2009. doi:10.1007/978-3-642-03359-9_4.
- 15 Marijn J. H. Heule, Oliver Kullmann, and Victor W. Marek. Solving and verifying the boolean pythagorean triples problem via cube-and-conquer. In Nadia Creignou and Daniel Le Berre, editors, *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International*

16:18 A Formal Proof of $R(4,5)=25$

- Conference, Bordeaux, France, July 5-8, 2016, Proceedings*, volume 9710 of *Lecture Notes in Computer Science*, pages 228–245. Springer, 2016. doi:10.1007/978-3-319-40970-2_15.
- 16 JG Kalbfleisch. Construction of special edge-chromatic graphs. *Canadian Mathematical Bulletin*, 8(5):575–584, 1965.
 - 17 Brendan D. McKay and Adolfo Piperno. Practical graph isomorphism, II. *J. Symb. Comput.*, 60:94–112, 2014. doi:10.1016/J.JSC.2013.09.003.
 - 18 Brendan D. McKay and Stanislaw P. Radziszowski. $R(4, 5) = 25$. *J. Graph Theory*, 19(3):309–322, 1995. doi:10.1002/JGT.3190190304.
 - 19 Konrad Slind and Michael Norrish. A brief overview of HOL4. In Otmane Aït Mohamed, César A. Muñoz, and Sofiène Tahar, editors, *Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, volume 5170 of *LNCIS*, pages 28–32. Springer, 2008. doi:10.1007/978-3-540-71067-7_6.
 - 20 Tjark Weber and Hasan Amjad. Efficiently checking propositional refutations in HOL theorem provers. *J. Appl. Log.*, 7(1):26–40, 2009. doi:10.1016/J.JAL.2007.07.003.
 - 21 Nathan Wetzler, Marijn Heule, and Warren A. Hunt Jr. DRAT-trim: Efficient checking and trimming using expressive clausal proofs. In Carsten Sinz and Uwe Egly, editors, *Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, volume 8561 of *Lecture Notes in Computer Science*, pages 422–429. Springer, 2014. doi:10.1007/978-3-319-09284-3_31.

Verifying Software Emulation of an Unsupported Hardware Instruction

Samuel Gruetter 

MIT, Cambridge, MA, USA

Thomas Bourgeat 

EPFL, Lausanne, Switzerland

Adam Chlipala 

MIT, Cambridge, MA, USA

Abstract

Some processors, especially embedded ones, do not implement all instructions in hardware. Instead, if the processor encounters an unimplemented instruction, an unsupported-instruction exception is raised, and an exception handler is run that implements the missing instruction in software. Getting such a system to work correctly is tricky: The exception-handler code must not destroy any state of the user program and must use the control and status registers (CSRs) of the processor correctly. Moreover, parts of the handler are typically implemented in assembly, while other parts are implemented in a language like C, and one must make sure that when jumping from the user program into the handler assembly, from the handler assembly into C, back to assembly and finally back to the user program, all the assumptions made by the different pieces of code, hardware, and the compiler are satisfied.

Despite all these tricky details, there is a concise and intuitive way of stating the correctness of such a system: User programs running on a system where some instructions are implemented in software behave the same as if they were running on a system where all instructions are implemented in hardware.

We formalize and prove such a statement in the Coq proof assistant, for the case of a simple exception handler implementing the multiplication instruction on a RISC-V processor.

2012 ACM Subject Classification Theory of computation → Program reasoning; Theory of computation → Program verification; Theory of computation → Program specifications; Software and its engineering → Formal software verification

Keywords and phrases Software verification, Software-hardware boundary, Coq

Digital Object Identifier 10.4230/LIPIcs.ITP.2024.17

Supplementary Material *Software*: <https://github.com/mit-plv/softmul>
archived at `swh:1:dir:63c00e5c3e541c31662a1ed54cc09045cf71607e`

Funding This work was supported in part by National Science Foundation grants 1521584, 2130671, and 2217064, as well as a gift from Google.

1 Introduction

Assembly language is frequently regarded as the lowest level of software abstraction in software-verification endeavors. However, the ISA (instruction-set architecture) semantics typically employed for software verification present an abstraction of the bare-metal ISA specifications, omitting machine-level aspects of the ISA, like the configuration registers that control the intricate interplay between the hardware's intrinsic capabilities and the meticulously crafted firmware (a piece of software) tasked with maintaining machine configurations and implementing high-privilege handlers in charge of emulating unsupported instructions, as well as managing other forms of low-level exceptions.



© Samuel Gruetter, Thomas Bourgeat, and Adam Chlipala;
licensed under Creative Commons License CC-BY 4.0

15th International Conference on Interactive Theorem Proving (ITP 2024).

Editors: Yves Bertot, Temur Kutsia, and Michael Norrish; Article No. 17; pp. 17:1–17:16

Leibniz International Proceedings in Informatics



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

17:2 Verifying Software Emulation of an Unsupported Hardware Instruction

For example, in the RISC-V ISA, control and status registers (CSRs) shape the behavior and functionality of the machine. These registers serve as a mechanism for controlling various aspects of the processor's operation, ranging from enabling or disabling specific features to controlling where the machine jumps in case of interrupts and exceptions. These registers and the associated exception handlers exert fundamental control over machine behaviors, so their improper configuration can lead to undefined outcomes.

CSRs coupled with the handlers introduce an intriguing specification, implementation, and verification challenge: while they are essential to determining the machine's behavior, the CSRs are themselves set and manipulated by software, and the handlers are themselves software.

There is a bit of a chicken-and-egg problem: We want to provide a nice and simple ISA abstraction, but to implement this abstraction and prove it correct, we have to write a trap handler and want to compile parts of it with a compiler whose proof already relies on this abstraction that we are supposed to implement, so how can we break the circularity?

One might be tempted simply to augment software-verification efforts with more detailed and faithful ISA specifications. We eschew this approach. The simplified ISA abstractions commonly employed are far more practical and productive compared to their cumbersome and heavier bare-metal counterparts, and the intricate details of configurations and handlers should anyway remain irrelevant to software or compilers higher up the stack.

This paper endeavors to disentangle the problem by focusing on a *simplified-yet-illustrative* instance: the specification, implementation, and verification of a RISC-V machine with software-implemented multiply instructions.

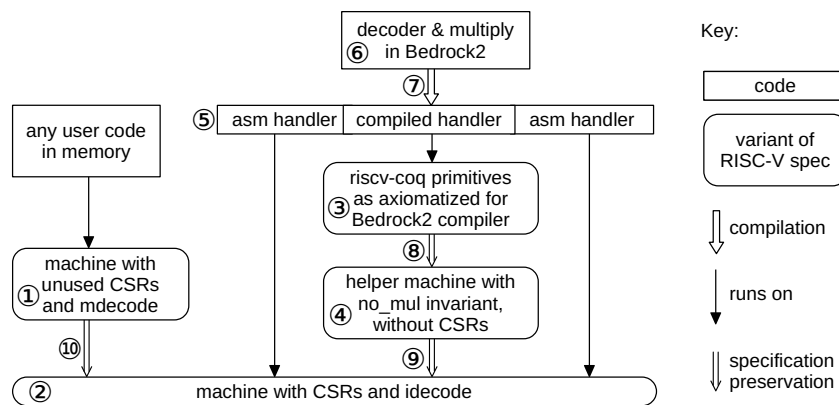
Through this exploration, we aim to shed light on the interesting challenges posed by CSRs and handlers and pave the way for a more coherent understanding of hardware-software interactions.

We will show that for this simple case we can indeed provide (with proofs!) the desired abstractions, and we can leverage tools that were built on top of those nice abstractions to provide the said abstractions without creating a circular conundrum. Our solution is to prove a helper lemma that ports assembly program-correctness proofs against the nice and simple ISA semantics to proofs against the detailed low-level ISA semantics. The helper lemma requires that the program does not contain any unsupported instruction that would trigger the trap handler, and this assumption gets discharged when we instantiate it with the concrete handler code produced by the compiler. However, there are also parts of the handler whose semantics cannot be expressed using the nice and simple ISA semantics, and we implement these manually in assembly and prove their correctness directly at the assembly level.

Our paper makes the following contributions:

- We propose a pleasantly simple specification for a RISC-V system equipped with a software trap handler emulating unsupported instructions: User programs running on a system where some instructions are implemented in software in a trap handler should behave as if they were running on a system with hardware support for these instructions.
- We implement such a trap handler by combining code in a C-like language with handwritten assembly code, and we prove its correctness, in a mechanized and foundational way, down to the binary machine code of the handler, combining symbolic-evaluation proofs at the C level and assembly level with a compiler-correctness proof.

All our code is publicly available at <https://github.com/mit-plv/softmul>.



■ **Figure 1** Overview diagram. The circled numbers are referenced in the text and do not stand for any meaningful order.

2 Overview

We want to show that a machine without hardware support for multiplication, but correctly configured with an exception handler that implements multiplication in software, behaves like a machine that supports multiplication in hardware. This theorem could then be used to simplify reasoning about programs running on a machine without hardware multiplication, because it saves the burden of reasoning about the trap handler and instead makes it as easy as reasoning about the specification with multiplication in hardware:

```

match inst with
| Mul rd rs1 rs2 ⇒
  x ← getRegister rs1;
  y ← getRegister rs2;
  setRegister rd (mul x y)
| ...
end

```

We use the RISC-V instruction-set architecture [1, 2], as formalized in `riscv-coq` [5]. RISC-V splits the instruction set into several extensions, each named with an uppercase letter. The base instruction set that every processor must support is called I, and multiplication, division and modulo operations are in a separate extension called M that small embedded processors may choose not to implement, or to implement in software by catching unsupported-instruction exceptions. In our proof-of-concept case study, we pretend that the M extension only contains one single instruction, namely the multiplication instruction, but we believe that support for the other instructions of the M extension could be added in the same way.

The `riscv-coq` specification defines a set of roughly a dozen *primitives* such as `getRegister`, `setRegister`, `loadByte`, `storeByte`, and then defines the semantics of each RISC-V instruction in terms of these primitives. As explained in [5], the semantics of each primitive is deliberately left unspecified in `riscv-coq`, so that each application that needs a formal specification of RISC-V can instantiate these primitives in a suitable domain-specific way.

Figure 1 presents an overview of our code (boxes ⑤ and ⑥) and specifications (the remaining boxes). Our theorem uses two instantiations of the `riscv-coq` specification: One that implements multiplication in hardware (box ①) and one (box ②) that implements it using a trap handler. Note that since the configurability of this specification is first-class, i.e. expressed in Coq itself rather than in some configuration files of the build process, there is no code duplication between the two instantiations.

17:4 Verifying Software Emulation of an Unsupported Hardware Instruction

Parts of the exception handler (box ⑥) are implemented in the Bedrock2 source language [8] (a small and simple subset of C) and compiled (Ⓣ) using the Bedrock2 compiler, but the handler also needs some low-level operations that are not expressible in the Bedrock2 source language and are therefore implemented by-hand in assembly. That is, our handler (box ⑤) starts and ends in handwritten assembly and calls a compiled Bedrock2 function in the middle. Our proof combines a program-logic proof about the Bedrock2 handler function, the compiler-correctness proof, and a proof about the assembly instructions, guaranteeing that all these parts have been put together correctly, and the final statement only mentions RISC-V semantics. All the other interfaces have been canceled out by combining the proofs and thus are not part of the trusted code base anymore.

In addition to the two instantiations of the RISC-V semantics with and without hardware multiplication, our proof (but not the final statement) also uses a third instantiation (box ④) which does not have any CSRs (control and status registers, required by the exception mechanism). This third instantiation fails (with undefined behavior) on all CSR-related instructions. For the compiler, an axiomatization (box ③) of this instantiation was chosen to simplify the proof, because the compiler does not emit any instructions that depend on CSRs.

3 The Top-Level Theorem Statement

We can state the theorem (arrow ⑩ in Figure 1) as follows:

```
Theorem softmul_correct: forall (initialH initialL: MachineState) (post: State → Prop),
  runsTo (mcomp_sat (run1 mdecode)) initialH post →
  R initialH initialL →
  runsTo (mcomp_sat (run1 idecode)) initialL (fun finalL ⇒
    exists finalH, R finalH finalL ∧ post finalH).
```

It is phrased as a *specification-preservation*¹ statement: If a machine with hardware multiplication runs from an initial state `initialH` to states satisfying a postcondition `post`, then every machine `initialL` with hardware multiplication, related to `initialH` by `R`, runs to a low-level state `finalL` which, when translated back to a high-level state `finalH`, satisfies the same postcondition.

The theorem uses `run1`, which defines how one single instruction is executed:

```
Definition run1(decoder: Z → Instruction): M unit :=
  pc ← getPC;
  inst ← Machine.loadWord Fetch pc;
  Execute.execute (decoder (LittleEndian.combine 4 inst));;
  endCycleNormal.
```

It is parameterized over the instruction decoder, which is instantiated with `mdecode` (a decoder that supports the multiplication instruction) in the hypothesis and with `idecode` (a decoder that returns `InvalidInstruction` for the multiplication instruction) in the conclusion of the theorem.

The `mcomp_sat` function is of type `M unit → State → (State → Prop) → Prop` and asserts that a monadic program (consisting of primitives used in `riscv-coq` such as `getRegister`, `setRegister`, `loadByte`, etc.), applied to some initial state, satisfies a postcondition, and `runsTo`

¹ It can also be seen as a *small-step omnisemantics forward simulation* as defined in [6].

```

Definition R(r1 r2: MachineState): Prop :=
  r1.(regs) = r2.(regs) ∧
  r1.(pc) = r2.(pc) ∧
  r1.(nextPc) = r2.(nextPc) ∧
  r1.(csrs) = map.empty ∧
  basic_CSRFields_supported r2 ∧
  regs_initialized r2.(regs) ∧
  exists mtvec_base scratch_end,
    map.get r2.(csrs) CSRField.MTVecBase = Some mtvec_base ∧
    map.get r2.(csrs) CSRField.MScratch = Some scratch_end ∧
    <{ * eq r1.(mem)
      * mem_available (word.of_Z (scratch_end - 256)) (word.of_Z scratch_end)
      * ptsto_bytes (word.of_Z (mtvec_base * 4)) softmul_binary }> r2.(mem).

```

■ **Figure 2** The predicate relating high-level states (multiplication implemented in hardware) to low-level states (multiplication implemented in software).

lifts it to an arbitrary (but finite) number of steps.² The predicate R (Figure 2) is used to relate a high-level state (i.e. the state of a machine that supports multiplication in hardware) to a low-level state (i.e. the state of a machine that implements multiplication in software using a trap handler), and it also contains all the preconditions on how the low-level machine needs to be configured. That is, R asserts that the two states have the same values for the registers and the program counter, and that the memory (modeled as a partial map from 32-bit addresses to bytes) of the low-level machine contains all of the high-level memory, as well as the instructions of the exception handler and some scratch space that the exception handler can use as its stack (which must be available even if the main program has used up all of its stack). To define at which address in memory the handler and the scratch space are located, RISC-V defines some CSRs [2] that our definition of R mentions:

- The CSR called `MTVecBase` is used to store the address of the trap handler (we use *direct* mode where all exceptions set the PC to the same address, but RISC-V also has a *vectored* mode where the PC is set to the base address in this register plus an offset corresponding to the cause of the exception).
- The CSR called `MScratch` is a read/write register dedicated for use by machine mode, and we use it to store the address of the *end* of the scratch space (we store the end address instead of the start address because it is used like a stack that grows downwards).

The memory (record fields `r1.(mem)` and `r2.(mem)` in Figure 2) is modeled as a finite map from 32-bit words to bytes. In the setup used in this case study, no primitive (nor other operation) changes the domain of that map. If an address outside of the domain of that map is accessed, the memory-access primitives cause undefined behavior, i.e. the `Prop` returned by `mcomp_sat` (and thus also the `Prop` returned by `runsTo`) becomes unprovable. This means that the `runsTo` hypothesis of the top-level theorem assumes a basic form of memory safety of the user program, namely that it does not access memory outside the domain of the memory. The separation-logic formula used in Figure 2 ensures that the memory the user program can write to (`r1.(mem)`) is disjoint from the scratch space and the handler code (second and third bullet points, respectively, in the separation-logic formula). To remove this memory-safety assumption, one could prove memory safety for the user program, i.e. that a `runsTo` holds for

² `runsTo` is defined like the omniseantics *eventually* operator [6].

17:6 Verifying Software Emulation of an Unsupported Hardware Instruction

an arbitrary postcondition (the easiest choice would simply be $\lambda s. \text{True}$). In our setting, user code and handler code both run in machine mode, but in more complex systems that feature both user mode and machine mode and also hardware-based memory-protection support (e.g. by segmentation or virtual memory), the requirement to assume or prove this basic memory safety for user programs could be lifted.

4 The Handler Code

The exception-handler code is implemented partially in handwritten assembly and partially in the Bedrock2 [8] source language and compiled to bytes by the Bedrock2 compiler. In order to *prove* the `softmul_correct` theorem, we use the correctness theorem of the Bedrock2 compiler, but note that the *statement* of the `softmul_correct` theorem does not depend on the Bedrock2 language semantics or on anything related to the fact that we used the Bedrock2 compiler, so the auditing burden for someone (who trusts the Coq proof checker) auditing our handler is much smaller, because one does not need to worry about the compiler, its language semantics, and its interaction with the assembly code.

The handwritten assembly of the handler is shown in Figure 3a. Since we want our software-emulated multiplication to behave as if it were implemented in hardware, we cannot make any assumptions about the remaining space on the user program’s stack, nor about whether the stack pointer `sp` contains any meaningful value at all. Therefore, we reserve a separate scratch space in memory just for our handler, and we require that the CSR `MScratch` contains the address of that scratch space.

As its first action (in `handler_init`), the handler has to store all 32 registers of the user process by which it was triggered. It may only use registers that it has already saved, because otherwise it would destroy state of the user program. We therefore resort to tricks such as temporarily storing the user stack pointer in the `MScratch` CSR and then temporarily storing it in the return-address register. Such tricks are easy to get wrong (and we did; see section 8.2).

After `handler_init`, the registers 3 to 31 are saved to the scratch space as well, and then the Bedrock2-generated part is called by passing it the value of the CSR register `MTVal1`, which contains the invalid instruction that caused the exception, and a pointer to the scratch space in which we saved the registers.

The Bedrock2 code (Figure 3b) is written directly in Coq using the custom-notations feature, a C-like syntax, and operator precedence as suggested by whitespace. It extracts the three 5-bit fields of the instruction that indicate the two source registers (operands of the multiplication operation) and the destination register, respectively, and then calls another Bedrock2 function `rpmul` that implements multiplication in terms of addition, storing the result back into the scratch space. The `rpmul` function iterates over the bits of the second operand while repeatedly doubling the first operand, a technique sometimes called “Russian peasant multiplication.” Both `softmul` and `rpmul` are verified using the Bedrock2 program logic. The spec of the former is given in Figure 4.

Its pre- and postcondition are expressed in terms of an (unused) I/O trace `t` and the memory `m`, for which we assert a list of two separation-logic clauses (a word array corresponding to the scratch space containing the register values, and a generic frame `R` for the rest of the memory).

After the Bedrock2 part, the handwritten snippet `inc_mepc` runs. It increases the CSR called `MEPC`, which stores the address of the instruction that caused the exception. This increment is needed because upon returning from the trap handler (by the `Mret` instruction), execution will jump to `MEPC`, so we have to set it to one instruction (i.e., 4 bytes) past the multiplication instruction.

```

Definition handler_init := [[
  Csrrw sp sp MScratch; (* swap sp and MScratch CSR *)
  Sw sp zero (-128);    (* save the 0 register (for uniformity) *)
  Sw sp ra (-124);     (* save ra *)
  Csrr ra MScratch;    (* use ra as a temporary register... *)
  Sw sp ra (-120);    (* ... to save the original sp *)
  Csrw sp MScratch;    (* restore the original value of MScratch *)
  Addi sp sp (-128)    (* remainder of code will be relative to updated sp *)
]].

Definition call_mul := [[
  Csrr a0 MVal; (* argument 0: value of invalid instruction *)
  Addi a1 sp 0; (* argument 1: pointer to memory with register values before trap *)
  Jal ra (Z.of_nat (1 + List.length inc_mepc + 29 + List.length handler_final) * 4)
]].

Definition inc_mepc := [[
  Csrr t1 MEPC;
  Addi t1 t1 4;
  Csrw t1 MEPC
]].

Definition handler_final := [[
  Lw ra sp 4;
  Lw sp sp 8; (* Bug: used to be `Csrr sp MScratch`, which is wrong if Mul sets sp *)
  Mret
]].

Definition asm_handler_insts := handler_init ++ save_regs3to31 ++
  call_mul ++ inc_mepc ++ restore_regs3to31 ++ handler_final.

```

(a) Assembly part of trap handler (embedded in Coq).

```

Definition softmul := func! (inst, a_regs) {
  a = a_regs + (inst>>15 & 31)<<2;
  b = a_regs + (inst>>20 & 31)<<2;
  d = a_regs + (inst>>07 & 31)<<2;
  unpack! c = rpmul(load(a), load(b));
  store(d, c)
}.

Definition rpmul := func! (x, e) ~> ret {
  ret = $0;
  while (e) {
    if (e & $1) { ret = ret + x };
    e = e >> $1;
    x = x + x
  }
}.

```

(b) Bedrock2 part of trap handler (using custom Coq notations to make it look similar to C).

■ **Figure 3** Trap handler code.


```

Instance spec_of_softmul : spec_of "softmul" :=
  fnspec! "softmul" inst a_regs / rd rs1 rs2 regvals R,
  { requires t m :=
    mdecode (word.unsigned inst) = MInstruction (Mul rd rs1 rs2) ^
    List.length regvals = 32 ^
    seps [a_regs ↦ word_array regvals; R] m;
  ensures t' m' := t = t' ^
    seps [a_regs ↦ word_array (List.upd regvals (Z.to_nat rd) (word.mul
      (List.nth (Z.to_nat rs1) regvals default)
      (List.nth (Z.to_nat rs2) regvals default))); R] m' }.

```

■ **Figure 4** Specification of softmul function.

And finally, in `restore_regs3to31` and `handler_final`, the values of the user program's registers are restored.

5 Combining the Program-Logic Proofs and Compiler-Correctness Proof

By combining the program-logic proofs about the two Bedrock2 functions with the compiler-correctness theorem, we can prove that if we run the compiler within Coq to obtain a list of instructions `mul_insts`, these instructions satisfy the specification shown in Figure 5, a verbose but unsurprising specification, laying out calling-convention details.

Lines 5 to 6 specify in which registers the arguments need to be placed, and line 14 requires that at address `a_regs`, there is an array of 32 words that store the values of the registers of the user program. Lines 18 to 20 state that after running `mul_insts`, the array at address `a_regs` storing the registers is updated at its `rd`'th index with the result of multiplying its `rs1`-th and `rs2`-th elements, and line 23 states that the new registers of the processor (not the ones saved in memory) only differ from the original registers on the callee-saved registers.

Note that the conclusion on line 27 refers to the same machine as the conclusion of the top-level theorem in section 3, namely the one described by `(mcomp_sat (run1 idecode))`, or box ② in Figure 1. However, to get there, two more proof steps (⑧ and ⑨) are needed: In order to keep the Bedrock2 compiler (somewhat) general, it was not proven against a specific instantiation of the `riscv-coq` semantics but against an axiomatization (box ③) of the primitives used in `riscv-coq` such as `getRegister`, `setRegister`, `loadByte`, etc. However, to keep the Bedrock2 compiler proof manageable, the RISC-V machine-state representation appearing in that axiomatization was hardcoded to a record type without CSRs (because compiler-emitted code never touches CSRs).

An additional problem requiring some proof effort to show compatibility is that the compiler correctness proof assumes a machine with hardware support for multiplication, but we want to run its code on one without. By inspecting the code that it generated, we can see that it did not output any multiplication instructions, but if it did, this would lead to a serious bug: If during the execution of the trap handler, a multiplication instruction were encountered, the trap handler would be recursively invoked again, infinitely many times.

We solve these two problems by introducing an intermediate helper machine (box ④) that uses the same state representation (without CSRs) as the compiler, and we prove an invariant `no_mul` saying that the memory region marked as executable (which only includes the compiled handler code in that instance) contains no multiplication instructions.

```

1 Lemma mul_correct: forall initial a_regs regvals invalidIInst R (post: State → Prop)
2   ret_addr stack_start stack_pastend rd rs1 rs2,
3   word.unsigned initial.(pc) mod 4 = 0 →
4   initial.(nextPc) = word.add initial.(pc) (word.of_Z 4) →
5   map.get initial.(regs) RegisterNames.a0 = Some invalidIInst →
6   map.get initial.(regs) RegisterNames.a1 = Some a_regs →
7   map.get initial.(regs) RegisterNames.ra = Some ret_addr →
8   map.get initial.(regs) RegisterNames.sp = Some stack_pastend →
9   word.unsigned ret_addr mod 4 = 0 →
10  word.unsigned (word.sub stack_pastend stack_start) mod 4 = 0 →
11  regs_initialized initial.(regs) →
12  mdecode (word.unsigned invalidIInst) = MInstruction (Mul rd rs1 rs2) →
13  128 ≤ word.unsigned (word.sub stack_pastend stack_start) →
14  seps [a_regs ↦ with_len 32 word_array regvals;
15        initial.(pc) ↦ program idecode mul-insts;
16        mem_available stack_start stack_pastend; R] initial.(MinimalCSRs.mem) ∧
17  (forall newMem newRegs,
18    seps [a_regs ↦ with_len 32 word_array (List.upd regvals (Z.to_nat rd) (word.mul
19      (List.nth (Z.to_nat rs1) regvals default)
20      (List.nth (Z.to_nat rs2) regvals default)));
21          initial.(pc) ↦ program idecode mul-insts;
22          mem_available stack_start stack_pastend; R] newMem →
23    map.only_differ initial.(regs) reg_class.caller_saved newRegs →
24    regs_initialized newRegs →
25    post { initial with pc := ret_addr; nextPc := word.add ret_addr (word.of_Z 4);
26          MinimalCSRs.mem := newMem; regs := newRegs } →
27    runsTo (mcomp_sat (run1 idecode)) initial post.

```

■ **Figure 5** The correctness lemma of the compiler-generated part of the handler.

6 Correctness Proof of the Assembly Part

The assembly part of the handler is proven correct by induction over the `runsTo` hypothesis of `softmul_correct`. If the machine with hardware multiplication executes any instruction besides multiplication, we just need to show that after executing the same instruction on the machine with software multiplication, the R judgment is preserved, but we can do that once-and-for-all by inspecting each *primitive* of the riscv-coq spec (`getRegister`, `setRegister`, `loadByte`, etc.), instead of analyzing the much larger number of RISC-V *instructions*. The interesting case is when the machine with hardware multiplication encounters a multiplication instruction, and we have to show that the machine with software multiplication steps to a related state. We do so by first symbolically executing the specification of what the *hardware* does in case of an exception (Figure 6), which boils down to setting some CSR fields and then setting the PC to the exception-handler address found in the `MTVecBase` CSR.

After that, we symbolically execute the handwritten assembly instructions, using Coq’s proof context to keep track of all the facts that we know about the current state of the machine. For each assembly instruction, we encounter its specification in terms of the primitives of riscv-coq, and for each primitive, we have a helper lemma that updates our symbolic state. At the point where we reach the call to the Bedrock2-generated code, we apply the correctness lemma for the compiled trap handler. After that call, we step through more handwritten assembly instructions that restore the registers and then call the `Mret` instruction that jumps back to one instruction past the multiplication instruction that caused the exception. At that point, we need to prove that the symbolic state accumulated in the Coq proof context implies that the two machines are still related by R , which only works if there are no bugs in the handler code.

17:10 Verifying Software Emulation of an Unsupported Hardware Instruction

```
Definition raiseExceptionWithInfo{A: Type}(isInterrupt exceptionCode info: t): M A :=
  pc ← getPC;
  (* hardcoded simplification: we only support machine mode and no interrupts *)
  addr ← getCSRField MTVecBase;
  setCSRField MTVal (regToZ_unsigned info);;
  (* these two need to be set just so that Mret will succeed at restoring them *)
  setCSRField MPP (encodePrivMode Machine);;
  setCSRField MPIE 0;;
  setCSRField MEPC (regToZ_unsigned pc);;
  setCSRField MCauseCode (regToZ_unsigned exceptionCode);;
  setPC (ZToReg (addr * 4));;
  @endCycleEarly M t MM MW MP A.
```

■ **Figure 6** Specification (in riscv-coq) of what hardware does in case of an exception.

7 What If ...

To explain our specification from a different angle, we list a few potential bugs that an implementor could introduce, and we show how they make our specification unprovable. Note that these are not bugs that actually occurred in our own implementation. For those, we refer to section 8.2. To present each potential bug, we ask: What if ...

- ... the compiler used to compile the handler emitted a multiplication instruction, which would cause the handler to trigger itself recursively infinitely many times? When proving correctness of the handwritten assembly (section 6), when we get to the jump instruction that calls the code emitted by the Bedrock2 compiler, we need to apply the compiler-correctness theorem (instantiated with the Bedrock2 part of our handler), but that theorem talks about execution on a machine with multiplication support, whereas the theorem we are about to prove is about execution on a machine without multiplication support. To make the proof work, we need to introduce box ④ and steps ⑧ and ⑨ in Figure 1 as explained in section 5, which at some point requires us to go through the concrete list of instructions emitted by the compiler and to check that none of them is a multiplication instruction.
- ... the handler runs at a time when no stack exists or the stack does not have enough remaining space? The output of the Bedrock2 compiler contains a number that indicates the amount of stack space that the compiled code needs, and one hypothesis of the compiler-correctness theorem is that at least that much space is available below the current stack pointer. In order to make sure this hypothesis holds, our trap handler uses a separate reserved scratch pad in memory as its stack, and when the correctness theorem for the handwritten assembly applies the instantiated compiler-correctness theorem `mul_correct`, it has to prove that there are at least 128 bytes of space remaining in the scratch pad, as mandated by the hypothesis on line 13 in Figure 5.
- ... the assembly that calls compiled Bedrock2 code makes wrong assumptions about the calling conventions of the compiler, e.g. which registers are used to pass arguments, or whether they are passed on the stack, in which direction the stack grows, or which registers are caller-saved? All these conventions are also captured in the intermediate lemma `mul_correct` in Figure 5.
- ... the handler forgot to increase MEPC, the CSR storing the address to which the machine jumps when we return from the exception handler, which would cause the faulting multiplication instruction to be run again and trigger the handler again? At the

end of the handler correctness proof, this bug would lead to a mismatch between the state of the machine with multiplication support (whose program counter gets advanced past the multiplication instruction) and the state of the machine without multiplication support (whose program counter would still point to the multiplication instruction).

- ... we ran a user program using compressed instructions (2-byte instructions) on our system? The riscv-coq specification only supports the uncompressed instruction format, where all instructions are 4 bytes long. There is no single location where the spec explicitly says “compressed instructions are *not* supported” – it requires an attentive reader who notices that the whole spec never mentions compressed instructions. In this scenario, our trap handler would fail to decode the unsupported instruction, and arbitrary behavior would occur. If riscv-coq did support compressed instructions, and our handler correctly decoded them, that would still require it to decide correctly whether to increase the MEPC by 2 or 4, and like in the previous point, one would notice the mismatch during the proof.

8 Evaluation

We attempt to answer the following evaluation questions (and dedicate one subsection to each of them):

1. Does our verified trap handler run on a RISC-V system implemented by a third party?
2. Did our implementation contain bugs that our verification caught?
3. Did our implementation contain bugs that our verification failed to catch?
4. Was the effort required for verification lower than the effort for debugging would have been?

8.1 Running Our Handler

To validate that our verified handler actually runs on a system not implemented by ourselves, we first looked for small embedded RISC-V processors without multiplication support but could not find any product with enough documentation in English to make us want to try it out. Instead, we chose to test our code in the Spike RISC-V ISA simulator [3], which offers fine-grained control over which RISC-V extensions are enabled.

We want to test that our handler behaves as expected on a system that runs a simple C program with multiplications, compiled by a third-party compiler. We wrote a simple program which computes the factorial of a hardcoded number and saves the result as well as a “done” flag to memory. We compiled it using the GNU RISC-V toolchain.

Our top-level theorem applies to a list of bytes called `softmul-binary` (mentioned in Figure 2 in the definition of the relation R), representing a piece of position-independent RISC-V machine code. However, Spike expects as input an ELF file. We relied on the GNU RISC-V toolchain to transform our binary into an ELF file, using a custom 25-line linker script.

For our theorem to be applicable, the conditions that the relation R (Figure 2) imposes on $r2$ (the machine without support for multiplication) must hold on our Spike machine. The first six conditions above the **exists** are related to the formalization and do not require any special setup action. The two lines below the **exists** require that the `MVecBase` and `MScratch` CSRs have suitable values, which we ensure by running an assembly script at the beginning that initializes these two CSRs with addresses defined in our linker script. The last three lines are a bullet-point separation-logic clause list describing the memory, saying that it must contain all of the specification machine’s memory `r1.(mem)`, as well as 256 bytes of scratch

17:12 Verifying Software Emulation of an Unsupported Hardware Instruction

memory at the address in the `MScratch` CSR and the `softmul`-binary at the address in the `MVecBase` CSR. Our linker script, together with the `memory-layout` command-line argument we pass to Spike, ensures that these conditions hold.

Spike comes with its own small language of debugger commands, and we used it to run the system until the `done` flag in memory is 1, then print the value of the memory at the address where we expect the result, and we also print the value of the CSR `minstret`, the number of retired instructions, to see how many instructions were executed.

No matter whether we invoked Spike with or without multiplication enabled, we observed the same result for `factorial(5)`, namely 120. With multiplication enabled, the number of instructions was 87; and with multiplication disabled, the number of instructions increased to 787, which shows that our handler indeed ran. As an additional sanity check, we also confirmed that it stops working if we set the `MVecBase` CSR to a different value.

Therefore, at least for this one simple example, we can answer question 1 with “yes”.

8.2 Bugs Caught During Verification

At the end of the proof that steps through the handwritten handler assembly, we need to prove that the symbolic state accumulated in the Coq proof context implies that the two machines are still related by R , which only works if there are no bugs in the handler code (see end of section 6). At that point, we found two interesting bugs. The first one was that we forgot to reset the `MScratch` CSR, so one invocation of the exception handler works fine, but the next one will use a wrong address for its scratch space. The second bug was the corner case where the multiplication instruction stores its result into the stack pointer. In that case, we must not override the stack pointer with the original stack pointer that we swapped into the `MScratch` register at the beginning of the handler.

We also found two more obvious bugs related to when to set the stack pointer and what stack-pointer offsets to use.

So we can answer question 2 with “yes”.

8.3 Bugs Encountered While Trying to Run It

We split the development of our experiment into two phases: First, we set up the linker script, with the trap handler already in place, but inactive, because we enabled the `M` extension. Once this experiment produced the expected output, we deactivated the `M` extension, so that our handler would run.

Getting phase 1 to work required some debugging. The most difficult part was to understand how to pass the linker-script-defined address of the heap memory to the C program, and it required reading the relevant page³ of the GNU Linker’s manual, which starts by saying that “accessing a linker script defined variable from source code is not intuitive,” and further down explains that “when you are using a linker script defined symbol in source code you should always take the address of the symbol, and never attempt to use its value”.

None of the code involved in phase 1 was verified, so it is not surprising that debugging was required. And to our delight, in phase 2, as soon as we disabled the `M` extension, our verified trap handler worked on the first try, and no debugging was needed at all.

³ <https://sourceware.org/binutils/docs/ld/Source-Code-Reference.html>

So, to answer question 3, there were bugs in the unverified part, but no bugs in the verified part.

In the future, it would be interesting also to verify ELF file generation, which we believe could have prevented the above bug.

8.4 Effort

For lack of better measures, we resort to lines-of-code counts as a very approximate measure of effort. Table 1 lists the counts of the different components.

It suggests that to produce 76 lines of verified code, a total of 3331 lines of code was necessary, which is more than a $40\times$ blowup. This ratio looks not very appealing, but it still seems fair to say that for tricky code, large proofs are sometimes needed. We also have some (potentially alleviating) remarks for each row of the table:

- The RISC-V helper instance is not referenced by the top-level theorem statement but acts as a bridge between the RISC-V spec used by the Bedrock2 compiler (whose state does not contain any CSRs) and the one used in the top-level theorem (whose state does have CSRs). Additionally, the helper instance maintains the invariant that no executable instructions are from the M extension, which is important during the execution of the trap handler, because if the trap handler contained a multiplication instruction, the trap handler would be invoked recursively over and over again. The helper instance and its accompanying lemmas are mostly copied from the one used in the compiler, and careful refactoring to share the code with the compiler could considerably reduce this count, which also means that these lines were low-effort to produce.
- To verify multiplication and a simple instruction decoder in Bedrock2, we used the original Bedrock2 program logic [8], which only automates the application of weakest-precondition rules but does not provide any automation for side condition solving. Using a framework that provides more automation would have reduced this proof size.
- A large chunk of the proof lines (1454) is in the correctness proof of the trap-handler parts written in assembly. The reason for this verbosity might be that, to our knowledge, this project is the first within the Bedrock2 ecosystem to verify more than two or three lines of assembly at a time, so there was no assembly-specific framework available. About two thirds of the proof code could probably be factored out into a framework that would be reusable for other assembly programs as well. We also did not spend too much time on side-condition automation, which could further reduce the number of proof lines. We conjecture that in a more mature assembly-verification framework, the assembly part of the trap-handler proof might be as short as maybe 100 lines of code. Moreover, the code-to-proof ratio also looks bad because we count the number of lines of Coq code rather than the number of assembly instructions, which matters for `save_regs3to31` and `restore_regs3to31`: Each of these is just a two-line functional program but expands to 29 assembly instructions.
- The compiler compat & invocation code deals with the different RISC-V instances and decoders and also applies the Bedrock2 compiler's correctness theorem for the instruction decoder and multiplier implemented in Bedrock2. It consists of important but not particularly interesting bookkeeping that quickly adds up to many lines of proof.
- Finally, the top-level theorem puts everything together. It requires some helper lemmas that could probably be generalized and moved to a library, but the fact that these lemmas were not already present in any library used in the Bedrock2 ecosystem seems fairly representative of the general verification experience, so it seems fair to count these lines.

17:14 Verifying Software Emulation of an Unsupported Hardware Instruction

■ **Table 1** Lines-of-code counts, excluding the dependencies (coqutil, riscv-coq, Bedrock2, and the Bedrock2 compiler).

	impl	spec	proof	total
RISC-V helper instance	0	101	309	410
Multiplication in Bedrock2	8	5	83	96
Instruction decoder in Bedrock2	7	27	80	114
Trap handler in assembly	36	28	1454	1518
Compiler compat & invocation	14	47	716	777
Top-level theorem	11	18	147	176
Excluded (imports & comments)				240
Total	76	226	2789	3331

Finding the bugs described in section 8.2 through debugging (especially the first two) might have been quite hard but would probably still not have taken as long as our verification effort took, so the answer for question 4 is probably a “no”.

However, we can imagine a promising world where the proof burden becomes lower than the debugging burden and verification becomes a part of most systems developers’ toolboxes.

9 Related Work

A number of projects have attempted to verify the interaction between (some or all of) C code, its compilation, handwritten assembly code, and trap handlers.

In the context of the Verisoft project, Alkassar et al. [4] verified a virtual-memory system that can swap out virtual memory pages onto disk. If an address is accessed that currently is on disk, a page fault is triggered, and a verified page-fault handler runs. Their correctness statement says that a physical machine with the page-fault handler can simulate a virtual machine (by which they mean a machine that provides to a user process a linear memory covering the whole address space). Their handler is implemented in C0 (a subset of C) with some inline assembly, which is modeled as external calls that modify additional state that cannot be modified directly from C0. That is, they call assembly from C, whereas we chose the opposite direction, calling C (or the C-like language Bedrock2, in our case) from assembly. In their project, saving and restoring of registers before and after the handler are not implemented in assembly and verified like we do but are instead part of the semantics of the physical machine.

BabyVMM [16] proves correctness of a simple virtual memory manager by showing that for all kernel implementations, linking the kernel with the virtual memory manager and running it on a machine with only physical memory (“hardware model” HW) behaves like running the kernel on a machine with an address space whose lower part is physical memory and whose upper part is virtual memory (“address space model” AS). It is implemented in a C-like language, and no compiler nor assembly code appears in the formalization. Instead, the theorem is stated in terms of C semantics. It also does not mention any page-fault handlers.

The verified microkernel seL4 [12] is implemented in C, but some small parts are handwritten assembly and are not verified [14, sections 4.4 and 4.8]. Contrary to our approach of using a verified compiler, they apply translation validation to the binary generated by GCC and certify using SMT solvers that it behaves like the C program.

CertiKOS [10, 11, 7] is a verified OS kernel. By means of certified abstraction layers, it fully captures the behavior of each component in a deep specification, so that from the outside, it does not matter whether the component is implemented in C or in assembly, thus achieving

interoperability at the proof level between C and assembly. Its correctness is expressed as a contextual refinement, based on CompCert’s [15] notion of a backward simulation, extended with a universal quantification over all possible surrounding programs (contexts): It states that for all assembly programs, all behaviors of that assembly program when linked with the low-level kernel can be simulated by the same program when linked with the high-level kernel specification. It relies on a notion of linking and uses CompCert’s formalization of assembly, which is still fairly high-level compared to binary machine code, e.g. jumps use labels instead of offsets or addresses, and there are instructions that allocate and free a stack frame that do not correspond to any machine instructions. CompCert’s assembly (which is used to model CertiKOS’s lowest layer) also does not model CSRs, whereas riscv-coq, on which our project is based, does, so to model trap handlers at our level of detail, the assembly (or machine) model would have to be extended.

CompCertELF [17], a different project by the same group, extends CompCert to also cover machine-code generation and uses a more realistic memory model, without the stack-frame allocation/freeing instructions mentioned above. As far as we know, CompCertELF has not (yet) been integrated with CertiKOS and is not publicly available. If it were, and if we managed to make CompCertELF compatible with our project, it could have helped to prevent the bug (section 8.3) we encountered in our unverified usage of the GNU linker to turn our plain binary into an ELF file.

Goel et al. [9] verify a subset of the instructions of an x86 processor which decodes x86 instructions and translates them into micro-operations before executing them. For the more complex instructions, the generated micro-operations contain a trap that causes a jump to microcode stored in a ROM. Similarly to our theorem, they prove that this processor behaves as if there were no micro-operations, traps or microcode, and instructions were executed according to a high-level x86 specification.

The CakeML compiler [13] targets multiple ISAs, and some instructions (e.g. division) are not supported by all of them, so the compiler has to implement some unsupported instructions in software, but contrary to our work, the necessary in-software implementation is emitted directly by the compiler, and no trap handler comes into play.

10 Conclusion and Future Work

We have shown a pleasantly simple way of specifying the correctness of a trap handler that emulates unsupported instructions in software, and we proved that our implementation of such a trap handler combining handwritten assembly and compiler-generated code satisfies this specification by combining symbolic-evaluation proofs about assembly and Bedrock2 programs with the correctness proof of the Bedrock2 compiler, as well as by proving that the output of the Bedrock2 compiler, which assumes a machine without CSRs and with hardware support for multiplication, also runs correctly on a machine with CSRs but without hardware support for multiplication.

This style of proof relating multiple execution models constitutes a first step towards the more ambitious goal of thoroughly proving correctness of a virtual memory system, stated in a similar flavor by saying that user programs running on a system with virtual memory (implemented by a combination of hardware, assembly, and C) behave as if they were running on a machine where the user program can use the full physical address space.

References

- 1 The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191213, December 2019. URL: <https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf>.

17:16 Verifying Software Emulation of an Unsupported Hardware Instruction

- 2 The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Document Version 20211203, December 2021. URL: <https://github.com/riscv/riscv-isa-manual/releases/download/Priv-v1.12/riscv-privileged-20211203.pdf>.
- 3 Spike RISC-V ISA Simulator, July 2023. URL: <https://github.com/riscv-software-src/riscv-isa-sim>.
- 4 Eyad Alkassar, Norbert Schirmer, and Artem Starostin. Formal Pervasive Verification of a Paging Mechanism. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 109–123, Berlin, Heidelberg, 2008. Springer. doi:10.1007/978-3-540-78800-3_9.
- 5 Thomas Bourgeat, Ian Clester, Andres Erbsen, Samuel Gruetter, Pratap Singh, Andy Wright, and Adam Chlipala. Flexible Instruction-Set Semantics via Abstract Monads (Experience Report). *Proceedings of the ACM on Programming Languages*, 7(ICFP):192:108–192:124, August 2023. doi:10.1145/3607833.
- 6 Arthur Charguéraud, Adam Chlipala, Andres Erbsen, and Samuel Gruetter. Omnisemantics: Smooth Handling of Nondeterminism. *ACM Transactions on Programming Languages and Systems*, 45(1):5:1–5:43, March 2023. doi:10.1145/3579834.
- 7 Hao Chen, Xiongnan Wu, Zhong Shao, Joshua Lockerman, and Ronghui Gu. Toward Compositional Verification of Interruptible OS Kernels and Device Drivers. *Journal of Automated Reasoning*, 61(1):141–189, June 2018. doi:10.1007/s10817-017-9446-0.
- 8 Andres Erbsen, Samuel Gruetter, Joonwon Choi, Clark Wood, and Adam Chlipala. Integration verification across software and hardware for a simple embedded system. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, pages 604–619, New York, NY, USA, June 2021. Association for Computing Machinery. doi:10.1145/3453483.3454065.
- 9 Shilpi Goel, Anna Slobodova, Rob Sumners, and Sol Swords. Verifying x86 instruction implementations. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2020, pages 47–60, New York, NY, USA, January 2020. Association for Computing Machinery. doi:10.1145/3372885.3373811.
- 10 Ronghui Gu, Jeremie Koenig, Tahina Ramananandro, Zhong Shao, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. Deep Specifications and Certified Abstraction Layers. Technical Report Technical Report YALEU/DCS/TR-1500, Yale University, October 2014.
- 11 Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. CertiKOS: An extensible architecture for building certified concurrent OS kernels. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI’16, pages 653–669, USA, November 2016. USENIX Association.
- 12 Gernot Heiser. The seL4 Microkernel – An Introduction. Technical report, The seL4@ Foundation, 2020. URL: <https://sel4.systems/About/seL4-whitepaper.pdf>.
- 13 Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony Fox, Scott Owens, and Michael Norrish. The verified CakeML compiler backend. *Journal of Functional Programming*, 29:e2, 2019. doi:10.1017/S0956796818000229.
- 14 Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. Comprehensive formal verification of an OS microkernel. *ACM Transactions on Computer Systems*, 32(1):2:1–2:70, February 2014. doi:10.1145/2560537.
- 15 Xavier Leroy. Formal Verification of a Realistic Compiler. *Commun. ACM*, 52(7):107–115, July 2009. doi:10.1145/1538788.1538814.
- 16 Alexander Vaynberg and Zhong Shao. Compositional Verification of a Baby Virtual Memory Manager. In Chris Hawblitzel and Dale Miller, editors, *Certified Programs and Proofs*, pages 143–159, Berlin, Heidelberg, 2012. Springer. doi:10.1007/978-3-642-35308-6_13.
- 17 Yuting Wang, Xiangzhe Xu, Pierre Wilke, and Zhong Shao. CompCertELF: Verified separate compilation of C programs into ELF object files. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):197:1–197:28, November 2020. doi:10.1145/3428265.

Mechanized HOL Reasoning in Set Theory

Simon Guilloud   

Laboratory for Automated Reasoning and Analysis, EPFL, Lausanne, Switzerland

Sankalp Gambhir   

Laboratory for Automated Reasoning and Analysis, EPFL, Lausanne, Switzerland

Andrea Gilot   

Laboratory for Automated Reasoning and Analysis, EPFL, Lausanne, Switzerland

Viktor Kunčák   

Laboratory for Automated Reasoning and Analysis, EPFL, Lausanne, Switzerland

Abstract

We present a mechanized embedding of higher-order logic (HOL) and algebraic data types (ADTs) into first-order logic with ZFC axioms. Our approach interprets types as sets, with function (arrow) types coinciding with set-theoretic function spaces. We assume traditional FOL syntax without notation for term-level binders. To embed λ -terms, we define a notion of context, defining the closure of all abstractions occurring inside a term. We implement the embedding in the Lisa proof assistant for schematic first-order logic and its library based on axiomatic set theory (presented at ITP 2023). We show how to implement type checking and the proof steps of HOL Light as proof-producing tactics in Lisa. The embedded HOL theorems and proofs are interoperable with the existing Lisa library. This yields a form of soft type system supporting top-level polymorphism and ADTs within set theory. The approach offers tools for Lisa users to carry HOL-style proofs within set theory. It also enables the import of HOL Light theorem statements into Lisa, as well as the replay of small HOL Light kernel proofs.

2012 ACM Subject Classification Theory of computation \rightarrow Logic and verification

Keywords and phrases Proof assistant, First Order Logic, Set Theory, Higher Order Logic

Digital Object Identifier 10.4230/LIPIcs.ITP.2024.18

Related Version *Extended Version with Appendices*: infoscience.epfl.ch/record/309962

Supplementary Material

Software: <https://github.com/epfl-lara/lisa/tree/itp2024-archive> [12]
archived at [swh:1:dir:c3f6e63aa274ed7ea292efcb0eade3f4abb60d2a](https://swh.1:dir:c3f6e63aa274ed7ea292efcb0eade3f4abb60d2a)

Funding This work is supported in part by the Swiss National Science Foundation grant 219474 “Algebraic Reasoning in Adaptive Verifiers”.

1 Introduction

The interactions and combinations of higher-order logic (HOL) with set theory in the context of proof systems have been a long-standing topic of study (e.g. [7, 10, 18, 6, 2]). Set theory (in particular ZF and ZFC) is the prototypical and oldest formalized foundation of mathematics, but it does not naturally admit a concept of “typed” expressions, which are widely used in informal mathematics, for guiding automated proof search, and in programming languages. HOL, on the other hand, naturally supports typed expressions, type checking and reasoning for simply typed lambda calculus with top-level polymorphism.

The first goal of the present work is to study a syntactic embedding of HOL into first-order set theory. It is well known that the Zermelo-Franekel axioms (ZF) imply the existence of a set that is a model of higher-order logic, where types are interpreted as sets, type judgement as set membership and λ -terms as set-theoretic functions [10]. However, the mere fact that



© Simon Guilloud, Sankalp Gambhir, Andrea Gilot, and Viktor Kunčák;
licensed under Creative Commons License CC-BY 4.0

15th International Conference on Interactive Theorem Proving (ITP 2024).

Editors: Yves Bertot, Temur Kutsia, and Michael Norrish; Article No. 18; pp. 18:1–18:18



Leibniz International Proceedings in Informatics

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

models of set theory contain a model of HOL (that is, a semantic embedding) is of little use by itself in practice. To be able to write the syntax and simulate the features of HOL, or to import theorems from HOL libraries, we need a *syntactic* embedding that transforms expressions in HOL into terms and formulas of first order set theory.

Some proof assistants have explored mixing features of set theory and HOL in various ways in their foundations, such as Egal [5], Isabelle/HOLZF [25] or ProofPeer [26]. A translation of statements and proofs from HOL to set theory has been done for some systems, for example between Isabelle/HOL and Isabelle/ZF [18], between Isabelle/HOL and Isabelle/Mizar [17, 7] or between HOL Light and Metamath [8]. However, both Isabelle/ZF and Metamath, as well as other systems using higher-order Tarski-Grothendieck [7], admit the Hilbert ϵ operator (as suggested in [10]) or built-in notations for replacement and comprehension which allow, in some form, writing *terms binding variables*. This is impossible in syntactically strict first-order logic as described in mathematical text books and in first-order automated theorem provers and proof assistants, where only the universal and existential quantifiers, \forall and \exists , can bind variables. This makes it impossible to naturally express arbitrary λ -terms of the form $\lambda x.t$ as standalone terms in FOL.

Nonetheless, we show that it is possible to embed higher-order logic and λ -terms in first-order set theory without these constructs using a notion of *contexts*, i.e. formulas that give local assumptions about terms. We study how this embedding impacts decision procedures for type checking and simulating the proof steps of HOL and implement the embedding in the Lisa proof assistant [13], whose foundations are built on first-order set theory. We obtain from this a form of soft type system over set theory, and support for reasoning about functions with HOL-like proofs steps in Lisa. Specifically, we implement the proof steps of HOL Light [14], for the simplicity of its foundations. This also allows automatic import of theorems from the HOL Light library, and while this embedding works well in practice for human-written proofs, which typically don't contain high towers of nested λ -abstractions, but our initial tests suggest that the embedding has too much overhead in the size of the proofs for the translation of large proofs whose basic building blocks are such λ -abstractions to be of practical use. Nonetheless, if HOL Light is trusted and we do not recheck proofs, our implementation allows importing all theorems and definitions from the HOL Light library efficiently.

In the second part of this paper, with the same motivation of simulating features from type systems into set theory, we describe how Algebraic Data Types (ADTs) can be encoded into set theory. ADTs are types defined inductively by their constructors, one of the simplest examples is that of singly-linked lists of integers, given by $\text{List} = \text{Nil} \mid \text{Cons}(\text{head} : \mathbb{Z}, \text{tail} : \text{List})$. ADTs and their generalizations are essential constructs in type theory based proof assistants (such as Coq [29] and Lean [22]) and in functional programming languages. Given the description of an ADT in terms of the type signature of its constructors, we show how to define the set corresponding to the type and functions representing the constructors, deriving the desired theorems about induction and injectivity. We show that expressions using ADTs and their constructors can be type checked by the same procedure as expressions from higher-order logic embedded in Lisa. Finally, we extend these results to polymorphic ADTs.

1.1 Contributions

The contribution of this paper is to present a practical embedding of simply typed lambda calculus with polymorphism, of the proof steps of HOL Light and of ADTs into classical ZFC within first-order logic, and its implementation in the Lisa proof assistant.

- We describe how to embed simply typed lambda calculus with top level polymorphism (and in particular lambda abstractions, which cannot be syntactically expressed as terms in first-order logic) into set theory. Our approach is based on maintaining a context of local definitions, expressing the desired properties about the subterms of λ -terms of the form $\lambda x.t$. If t contains variables other than x , i.e. free variables, we need to encode the closure of $\lambda x.t$ instead, similar to the compilation of programs containing nested function declarations.
- We explain how this encoding allows simulating proof steps and type checking from HOL by producing the corresponding proofs in set theory.
- We implement this embedding and the proof-producing tactics in the Lisa proof assistant, allowing reasoning about set-theoretic functions using HOL proof steps.
- We use this embedding to import parts of the HOL Light library. We find that importing theorem statements themselves is fast, but replaying HOL proofs within set theory faces scalability challenges for large proofs.
- We describe how algebraic data types (ADTs) can be automatically defined in ZF set theory and how to obtain their key recursive properties derived from the recursion theorem in ZF set theory. We mechanize this system in Lisa, making ADTs and their constructors fully compatible with implemented HOL tactics.

In the present work we picked HOL Light as our reference system for HOL, but with some additional work, the results can be translated to other proof assistants in the HOL-family of proof assistants, such as HOL4 [28], Isabelle/HOL [24], or Candle [1].

Similarly, our target system was Lisa, but none of the results are specific to Lisa, and they transfer to other systems using axiomatic set theory in first-order logic, such as Mizar [23]. The implementation corresponding to the work we describe can be found at:

<https://github.com/epfl-lara/lisa/tree/itp2024-archive> .

The current Lisa repository, which incorporates some of these techniques, is at:

<https://github.com/epfl-lara/lisa/> .

2 Preliminaries

There exist several different variations of higher-order logic (HOL). We consider it as it is defined in HOL Light. Its language is the simply typed lambda calculus with top-level polymorphism (or, the Hindley-Milner type system). Each variable in an HOL term is associated with a single type, which may contain type variables. We note V^λ the set of HOL variables and T^λ the set of type variable symbols of HOL. The deduction rules of HOL Light are described in Figure 1¹. We denote by \mathcal{B} the type of Booleans containing two elements, and by $=_A$ the built-in polymorphic function representing equality on type A .

We assume familiarity with the syntax and deduction rules of first-order logic (see, for example, [21]). We take Sequent Calculus [9] as our proof system, which is used by our proof assistant Lisa, but the results transfer to other proof systems for first-order logic (FOL). We call *first-order set theory* (FOST) the axiomatic system of ZFC [15, 19] in first-order logic. This is also the foundation of the Lisa proof system [13] in which we implement our result.

¹ HOL Light also admits a choice function and an infinity axiom later in the library development, which are justified in ZFC by the choice axiom and infinity axiom. These two additional axioms are largely tangential to the concerns of the present work. While ETA is also formally an axiom in HOL Light, we consider it as a basic rule because set-theoretic functions are naturally extensional, and to handle alpha-equivalence in Section 3.

18:4 Mechanized HOL Reasoning in Set Theory

$\frac{}{\vdash t =_A t} \text{REFL}$	$\frac{\Gamma \vdash s =_A t \quad \Delta \vdash t =_A u}{\Gamma, \Delta \vdash s =_A u} \text{TRANS}$
$\frac{\Gamma \vdash s =_{A \rightarrow B} t \quad \Delta \vdash u =_A v}{\Gamma, \Delta \vdash s(u) =_B t(v)} \text{MK_COMB}$	$\frac{\Gamma \vdash s =_B t}{\Gamma \vdash (\lambda x : A. s) =_{A \rightarrow B} \lambda(x : A. t)} \text{ABS}$
$\frac{}{\vdash \lambda(x : A. t)x =_B t} \text{BETA}$	$\frac{}{p \vdash p} \text{ASSUME}$
$\frac{\Gamma \vdash p =_B q \quad \Delta \vdash p}{\Gamma, \Delta \vdash q} \text{EQ_MP}$	$\frac{\Gamma, q \vdash p \quad \Delta, p \vdash q}{\Gamma, \Delta \vdash p =_B q} \text{DEDUCT_ANTISYM_RULE}$
$\frac{\Gamma \vdash p}{\Gamma[\vec{x} := \vec{t}] \vdash p[\vec{x} := \vec{t}]} \text{INST}$	$\frac{\Gamma \vdash p}{\Gamma[\vec{X} := \vec{A}] \vdash p[\vec{X} := \vec{A}]} \text{INSTTYPE}$
$\frac{}{\vdash (\lambda x. tx) =_A t} \text{ETA}$	

■ **Figure 1** Deduction rules for higher-order logic as implemented in HOL Light.

In both HOL (see [3]) and FOL, the language can be extended conservatively using the concept of *extension by definition*, as described (for example) in [19, Section 2.10] and [11].

► **Theorem 2.1** (Extension by Definition for First Order Logic). *Let K be a first order theory (for example, FOST), and ϕ a formula with free variables y, x_1, \dots, x_n . Suppose $\vdash_K \forall x_1, \dots, x_n \exists! y. \phi$ and let the theory K' be K with the addition of a function symbol f of arity n and the axiom $\forall y, y = f(x_1, \dots, x_n) \iff \phi$. Then K' is fully conservative² over K .*

2.1 Set-Theoretic Semantics of HOL

To motivate our translation from HOL to set theory, we review classical set-theoretic semantics of HOL. This allows us to focus first on the semantics of functions and types, without having to worry about if a certain set is expressible, efficiently or at all, as a term in FOST. We interpret types as sets and HOL functions as total set theoretic functions.

As is usual in set theoretic foundations, we identify a function $f : A \rightarrow B$ with its graph, that is, as a subset of $A \times B$ such that for every element x of A , there exists exactly one element $y \in B$ where $(x, y) \in f$. We write $\text{isFunction}(f, A, B)$ to denote that the set f is a total and functional relation (or simply, a function) from A to B .

We define an operator app such that, for all f such that $\text{isFunction}(f, A, B)$ and for all $x \in A$, $\text{app}(f, x) = y$ iff $(x, y) \in f$.

► **Definition 2.2** (Set theoretic universe). *We use the following concepts to give a classical semantics to HOL.*

- *Fix U to be a set that is a universe of Zermelo set theory, i.e., containing an infinite set, and closed under powersets, unions, and subsets defined by set comprehension (separation axiom). Consequently, U is closed under Cartesian products. For example, we can take U to be the set $V_{\omega+\omega}$ of the cumulative (von Neumann) hierarchy [15, Chapter 6] in ZFC.*
- *Let app and isFunction be as above.*
- *For $A, B \in U$, let $A \Rightarrow B$ denote the set $\{r \in \mathcal{P}(A \times B) \mid \text{isFunction}(r, A, B)\}$.*

² Full conservativity says that any formula in the new language has an equivalent formula in the old language. This is stronger than typical conservativity and necessary in the presence of axiom schemas, but the distinction is not critical here. See [11] for more details.

- Let \mathbb{N} be the set of natural numbers.
- Let $\perp = \emptyset$, $\top = \{\emptyset\}$ and $\mathbb{B} = \{\perp, \top\}$.
- For $A \in U$, let $E(A) =$

$$\{(x, f) \in (A \times (A \Rightarrow \mathbb{B})) \mid f = \{(y, b) \in (A \times \mathbb{B}) \mid (x = y \rightarrow b = \top) \wedge (x \neq y \rightarrow b = \perp)\}\}.$$

Note that for all A , $E(A) \in (A \Rightarrow (A \Rightarrow \mathbb{B}))$.

► **Definition 2.3** (Semantics of HOL). *An assignment $\alpha : (V^\lambda \cup T^\lambda) \rightarrow U$ is a function such that for all $x : A \in V^\lambda$, $\alpha(x : A) \in \alpha(A)$. We define an interpretation of HOL terms with respect to an assignment:*

$$\begin{aligned} \llbracket A \rrbracket_\alpha &= \alpha(A) \\ \llbracket T_1^\lambda \rightarrow T_2^\lambda \rrbracket_\alpha &= \llbracket T_1^\lambda \rrbracket_\alpha \Rightarrow \llbracket T_2^\lambda \rrbracket_\alpha \\ \llbracket \mathcal{B} \rrbracket_\alpha &= \mathbb{B} \\ \llbracket i \rrbracket_\alpha &= \mathbb{N} \\ \llbracket x : A \rrbracket_\alpha &= \alpha(x) \\ \llbracket =_A : A \rightarrow A \rightarrow \mathcal{B} \rrbracket_\alpha &= E(\llbracket A \rrbracket_\alpha) \\ \llbracket (f : A \rightarrow B)(t : A) : B \rrbracket_\alpha &= \text{app}(\llbracket f : A \rightarrow B \rrbracket_\alpha, \llbracket t : A \rrbracket_\alpha) \\ \llbracket (\lambda x : A. t : B) : A \rightarrow B \rrbracket_\alpha &= \{(y, z) \in (\llbracket A \rrbracket_\alpha \times \llbracket B \rrbracket_\alpha) \mid z = \llbracket t \rrbracket_{\alpha[x \mapsto y]}\}. \end{aligned}$$

► **Definition 2.4** (Syntactic and Semantic truth).

For any FOST sequent $s = (l_1, \dots, l_n) \vdash (r_1, \dots, r_n)$, we write:

- $\vdash s$ if s is provable in FOST

For any HOL sequent $s = (l_1, \dots, l_n) \vdash r$, we write:

- $\vdash s$ if s is provable in HOL
- $U \models s$ if, for every assignment α , $(\llbracket l_1 \rrbracket_\alpha = \top \wedge \dots \wedge \llbracket l_n \rrbracket_\alpha = \top) \implies \llbracket r \rrbracket_\alpha = \top$ holds in U

► **Theorem 2.5.** For any term $s : A \in t^\lambda$ and assignment α , $\llbracket s : A \rrbracket_\alpha \in \llbracket A \rrbracket_\alpha$

Proof. By induction on the structure of t . ◀

We can show that all rules of HOL from Figure 1 hold in ZFC, giving the following theorem:

► **Theorem 2.6.** For any assignment α and HOL terms $s_1 : \mathcal{B}, \dots, s_n : \mathcal{B}$ and $t : \mathcal{B} \in t^\lambda$,

$$\text{if } (s_1, \dots, s_n \vdash t \text{ and } \forall i. \llbracket s_i : \mathcal{B} \rrbracket_\alpha = \top) \text{ then } \llbracket t : \mathcal{B} \rrbracket_\alpha = \top$$

While the above argument shows that HOL has an interpretation into first-order set theory, it does not immediately give us a mechanical translation from an HOL proof system to proofs in mechanized set theory. In particular, note that in Definition 2.3, the right-hand side of the lambda case cannot be expressed in the syntax of first-order logic. It tells us neither if and how we can automate the translation of an HOL proof into an FOST proof, nor the production of proofs of a statement $t \in A$ that would correspond to type checking. However, note also that the embedding is shallow, in the sense that HOL functions are interpreted as usual set theoretic functions and types of functions as sets of set theoretic functions.

3 From HOL Formulas to First-Order Set Theory Formulas

We wish to define a translation (\cdot) from HOL sequents to FOL sequents such that if an HOL sequent s is provable in HOL, then (s) is provable in FOST. Technically, a trivial such embedding would map all sequents to the trivially true sequent. We cannot require that

18:6 Mechanized HOL Reasoning in Set Theory

the embedding maps unprovable sequents of HOL to unprovable sequents of FOST, because FOST is strictly more powerful and can prove additional statements. But, we can require that the embedding does not map semantically false statements to provable sequents. This means, for every sequent s of HOL:

1. $\vdash s \implies \vdash \langle s \rangle$
2. $\vdash \langle s \rangle \implies U \models s$

Moreover, for the embedding to be of practical use in theorem proving and for import of proofs, we would like the embedding to be as natural as possible, so that we ideally have an embedding $\langle \cdot \rangle : t^\lambda \rightarrow t$, i.e. from terms of HOL to terms of FOST, such that, for every assignment α , we have $\llbracket \langle s : A \rangle \rrbracket_\alpha = \llbracket s : A \rrbracket_\alpha$. Unfortunately, the syntax of FOST terms does not support λ -abstractions. Of course, the set we denote by

$$\{(y, z) \in (\llbracket A \rrbracket_\alpha \times \llbracket B \rrbracket_\alpha) \mid z = \llbracket t \rrbracket_{\alpha[x \mapsto y]}\}$$

is guaranteed to exist by the Comprehension axiom, but the above expression is not a term in first-order logic. In particular, any variable that appears in a term has to be free, but here, we would want y and z to be bound. While the symbol E was defined similarly with a comprehension, we need to show the existence and uniqueness of E only once to introduce it with a definitional extension once and for all, as in Theorem 2.1.

We represent λ -abstractions using a variable which is only valid under some *context*, a set of formulas. For example, we can represent the term $\lambda x : \mathcal{B}.x$ using:

- a *variable* λ_1 , along with the corresponding
- *context*, the formula $\lambda_1 \in (\mathbb{B} \Rightarrow \mathbb{B}) \wedge \forall x \in \mathbb{B}. \lambda_1(x) = x$

Formally, using the mechanism of extension by definition, we first extend FOST with constant and functional symbols for $\Rightarrow, \mathbb{B}, \mathbb{N}, E$, and app , according to Definition 2.2. We then define $\langle \cdot \rangle$ as follows:

► **Definition 3.1** (Embedding of HOL into FOST). *Reserve a special set of variables $\Lambda = \{\lambda_1, \lambda_2, \dots\}$ that are used to represent lambda expressions and associate to every HOL term t a single i . In practice, we use a global counter. We use the standard application notation for FOST terms, so that, for example, $\lambda_2 x y$ really means $\text{app}(\text{app}(\lambda_2, x), y)$.*

$$\begin{aligned} \langle X \rangle &= X \\ \langle T_1^\lambda \rightarrow T_2^\lambda \rangle &= \langle T_1^\lambda \rangle \Rightarrow \langle T_2^\lambda \rangle \\ \langle \mathcal{B} \rangle &= \mathbb{B} \\ \langle i \rangle &= \mathbb{N} \\ \langle x : A \rangle &= x \\ \langle =_A : A \rightarrow A \rightarrow \mathcal{B} \rangle &= E(\langle A \rangle) \\ \langle (f : A \rightarrow B)(t : A) : B \rangle &= \langle f : A \rightarrow B \rangle \langle t : B \rangle \\ \langle (\lambda x : A. t : B) : A \rightarrow B \rangle &= \lambda_i y_1 \dots y_n \\ &\quad \text{where } y_1, \dots, y_n \text{ are the free variables of } \lambda x.t, \\ &\quad \text{and } \lambda_i \text{ is a variable symbol associated with the term } \lambda x.t. \end{aligned}$$

In the last line, λ_i is a representation of the *closure* of the lambda term $\lambda x.t$, and is intended to only be valid under the appropriate defining assumption.

There is another issue with this encoding, which is that we are losing type information associated to variables, as well as the HOL assumption that type variables cannot represent empty types. Fortunately, this can also be solved with contexts.

3.1 HOL in FOST Using Contexts

To translate propositions of HOL into FOST, we need to compute *contexts* of HOL terms. We will need a *non-emptiness context*, to handle type variables, a *typing context*, to carry over information regarding types of variables, and a *definition context* to handle abstractions.

The following definition defines ctx^N (non-emptiness), ctx^T (variable typing), and ctx^D (definitions). Assume for simplicity and without loss of generality that variables typed differently have different identifiers, so that, for example x , $x : A$, and $x : B$ do not appear together in the same proof.

► **Definition 3.2** (Non-Emptiness Context). *The typing context of an HOL term is the set of assumptions $A \neq \emptyset$ for every type variable A in the term. This also includes type variables in the type signature of polymorphic constant symbols.*

► **Definition 3.3** (Typing Context). *The typing context of an HOL term is a set of FOST formulas of the form $x \in T$ and is computed recursively as follows:*

$$\begin{aligned} \text{ctx}^T(x : T) &= \{x \in T\} \\ \text{ctx}^T(c) &= \emptyset \text{ for } c \text{ a constant symbol} \\ \text{ctx}^T(ft) &= \text{ctx}^T(f) \cup \text{ctx}^T(t) \\ \text{ctx}^T(\lambda x : T. t) &= \text{ctx}^T(t) - \{x \in T\} \end{aligned}$$

► **Definition 3.4** (Definitional Context). *The definition context of an HOL term is a set of FOST formulas whose free variables are from Λ (from Definition 3.1) and from the set of type variables. It is computed as follows:*

$$\begin{aligned} \text{ctx}^D(x : T) &= \emptyset \\ \text{ctx}^D(c) &= \emptyset \\ \text{ctx}^D(ft) &= \text{ctx}^D(f) \cup \text{ctx}^D(t) \\ \text{ctx}^D(\lambda x : T. t) &= \text{ctx}^D(t) \cup \\ &\quad \{(\lambda_i \in (\langle T_1 \rangle \Rightarrow \dots \Rightarrow \langle T_n \rangle \Rightarrow \langle T \rangle \Rightarrow \langle \text{type}(t) \rangle)) \wedge \\ &\quad \quad \forall y_1 \in T_1, \dots, \forall y_n \in T_n, \forall x \in T. \lambda_i y_1 \dots y_n x = \langle t \rangle\} \\ &\quad \text{where } y_1 : T_1, \dots, y_n : T_n \text{ are the free variables of } t \text{ (without } x). \end{aligned}$$

λ_i represents the closure of the λ -expression, as in the supercombinator compilation of functional programming languages [16, Chapter 13]. Having λ_i represent the closure of the lambda abstraction rather than the abstraction itself is necessary because otherwise the y_i 's would be free in the definition. But this should not be the case if they are supposed to be bound in an outer term. This is illustrated in the third formula in the following Example 3.5.

The *context*, $\text{ctx}(t)$, of an HOL term t , is $\text{ctx}^N(t) \cup \text{ctx}^T(t) \cup \text{ctx}^D(t)$.

► **Example 3.5.** Let $x : X$, $y : Y$, $f : Y \rightarrow X$, $g : X \rightarrow Y$. We omit type annotations from lambda terms.

$$\begin{aligned} \langle \lambda x. x \rangle &= \lambda_1 \\ \text{ctx}(\lambda x. x) &= \{X \neq \emptyset, \lambda_1 \in X \Rightarrow X \wedge \forall x \in X. (\lambda_1 x) = x\} \\ \langle (\lambda x. y) (f y) \rangle &= \lambda_2 y (f y) \\ \text{ctx}(\langle (\lambda x. y) (f y) \rangle) &= \{X \neq \emptyset, Y \neq \emptyset, y \in Y, f \in Y \Rightarrow X \\ &\quad (\lambda_2 \in Y \Rightarrow X \Rightarrow Y) \wedge \forall y \in Y. \forall x \in X. (\lambda_2 y x) = y\} \\ \langle (\lambda y. (\lambda x. y) =_{X \rightarrow Y} g) \rangle &= \lambda_3 g \\ \text{ctx}(\langle (\lambda y. (\lambda x. y) =_{X \rightarrow Y} g) \rangle) &= \{X \neq \emptyset, Y \neq \emptyset, g \in X \Rightarrow Y, \\ &\quad (\lambda_2 \in Y \Rightarrow X \Rightarrow Y) \wedge \forall y \in Y. \forall x \in X. (\lambda_2 y x) = y, \\ &\quad (\lambda_3 \in (X \Rightarrow Y) \Rightarrow Y \Rightarrow \mathbb{B}) \wedge \\ &\quad \forall g \in X \Rightarrow Y. \forall y \in Y. (\lambda_3 g y = E(X \Rightarrow Y) (\lambda_2 y) g)\} \end{aligned}$$

18:8 Mechanized HOL Reasoning in Set Theory

Note that in the last example, the definition of λ_3 refers to λ_2 , and binds the variable y which is free in the λ -abstraction represented by λ_2 . Without the closure, y would be free in the definition of λ_2 and could not be bound in the definition of λ_3 . Recall that $E(A)$ denotes the interpretation of the (curried) equality relation on A .

We can now define the embedding of sequents:

► **Definition 3.6.** *Let $s = t_1, \dots, t_n \vdash t$ be an HOL sequent. Define the embedding $\langle s \rangle$ as*

$$\text{ctx}(t_1), \dots, \text{ctx}(t_n), \text{ctx}(t), \langle t_1 \rangle = \top, \dots, \langle t_n \rangle = \top \vdash \langle t \rangle = \top.$$

3.2 Proof of Type Checking

To produce proofs corresponding to type checking, we define a proof tactic in `ProofType(t: Term)` in Lisa, which for any term t of type T outputs a proof of

$$\text{ctx}(t) \vdash \langle t \rangle \in \langle T \rangle$$

As abstractions are represented by typed variables applied to some other free variables in our encoding, the tactic only has to type applicative terms. For example, consider the term $(\lambda x : A. y : A)(z : A)$. The corresponding theorem of first-order set theory is:

$$(\lambda_1 \in A \Rightarrow A \Rightarrow A) \wedge (\forall y \in A. \forall x \in A. \lambda_1 x y = y), y \in A, z \in A \vdash (\lambda_1 y) z \in A$$

for which our approach generates a proof by recursing on the structure of t and T , and using the definition of function spaces.

Polymorphism

More interesting is the typing of polymorphic constants such as HOL equality $=_A$. Its HOL type is $A \rightarrow A \rightarrow \mathcal{B}$ and its interpretation according to Definition 3.1 is $E(A)$. Hence, the corresponding typing judgement proven by `ProofType` should be $E(A) \in A \rightarrow A \rightarrow \mathcal{B}$. In simply typed lambda calculus with explicit polymorphism (like System F, see for example [4]), $=$ would be given the type $\Lambda A. A \rightarrow A \rightarrow \mathcal{B}$ to E . The corresponding property for E in FOST is $\forall A. E(A) \in A \Rightarrow A \Rightarrow \mathbb{B}$. This is conveniently represented in our embedding using free set variables in sequents. We added support for such top-level polymorphism to the `ProofType` tactic, so that it can automatically type polymorphic constants embedded this way.

3.3 Simulating HOL Proofs

The goal of the section is to demonstrate that HOL Light proof steps can be simulated by proofs in our encoding.

► **Theorem 3.7** (Simulating HOL Proofs in FOST). *Let*

$$\frac{s_1 \quad \dots \quad s_n}{s}$$

be an instance of a deduction rule of HOL from Figure 1. Then

$$\frac{\langle s_1 \rangle \quad \dots \quad \langle s_n \rangle}{\langle s \rangle}$$

is admissible in FOST (rules of sequent calculus and axioms of set theory).

We state three auxiliary theorems of FOST which will be necessary for the simulation:

► **Lemma 3.8.** *The following statements are theorems of FOST:*

$$\begin{aligned} x \in A, y \in A \vdash (E(A) \ x y = \top) &\Leftrightarrow (x = y) && \text{(Correctness of } E\text{)} \\ f \in A \Rightarrow B, g \in A \Rightarrow B, \forall x \in A. f x = g x \vdash f = g &&& \text{(Functional Extensionality)} \\ p \in \mathbb{B}, q \in \mathbb{B}, (p = \top) \Leftrightarrow (q = \top) \vdash p = q &&& \text{(Propositional Extensionality)} \end{aligned}$$

The simulation of a proof step can in general be split into two parts: first, produce a proof under arbitrary typing and context assumptions, and then handle the modifications in context. For example, let $x : \mathcal{B}, f : \mathcal{B} \rightarrow \mathcal{B}, g : \mathcal{B} \rightarrow \mathcal{B}$ and consider a **TRANS** step deducing

$$\frac{\Gamma \vdash x =_{\mathcal{B}} f x \quad \Gamma \vdash f x =_{\mathcal{B}} g x}{\Gamma \vdash x =_{\mathcal{B}} g x}$$

and let $c_{\Gamma} = \text{ctx}(\Gamma)$. We wish to obtain a proof of

$$\frac{x \in \mathbb{B}, f \in \mathbb{B} \Rightarrow \mathbb{B}, c_{\Gamma}, (\Gamma) \vdash (E(\mathbb{B}) \ x (f x)) = \top \quad x \in \mathbb{B}, f \in \mathbb{B} \Rightarrow \mathbb{B}, g \in \mathbb{B} \Rightarrow \mathbb{B}, c_{\Gamma}, (\Gamma) \vdash (E(\mathbb{B}) \ (f x) (g x)) = \top}{x \in \mathbb{B}, g \in \mathbb{B} \Rightarrow \mathbb{B}, c_{\Gamma}, (\Gamma) \vdash (E(\mathbb{B}) \ x (g x)) = \top}$$

This should follow from applying Correctness of E (Lemma 3.8) to each premise, using transitivity of first order equality, and applying back Correctness of E (Lemma 3.8) to the result. However, to apply this lemma, we need the facts $f x \in \mathbb{B}$ and $g x \in \mathbb{B}$. Moreover, the $f \in \mathbb{B} \Rightarrow \mathbb{B}$ assumption from the premise will stay in the conclusion, yielding:

$$x \in \mathbb{B}, f \in \mathbb{B} \Rightarrow \mathbb{B}, g \in \mathbb{B} \Rightarrow \mathbb{B}, f x \in \mathbb{B}, g x \in \mathbb{B}, c_{\Gamma}, (\Gamma) \vdash (x =^{\lambda} g x) = \top$$

which is a correct conclusion, but contains too many assumptions. Fortunately, these assumptions can be eliminated.

Eliminating lingering assumptions

Pursuing the example above, let $L = \{x \in \mathbb{B}, g \in \mathbb{B} \Rightarrow \mathbb{B}, c_{\Gamma}, (\Gamma)\}$ and $R = (x =^{\lambda} g x) = \top$. We want to simulate the following proof step:

$$\frac{f \in \mathbb{B} \Rightarrow \mathbb{B}, f x \in \mathbb{B}, g x \in \mathbb{B}, L \vdash R}{L \vdash R}$$

First, we prove (automatically) the non-elementary typing assumptions $(x \in \mathbb{B}, f \in \mathbb{B} \Rightarrow \mathbb{B}) \vdash f x \in \mathbb{B}$ by recursing over the structure of $f x$ (as in Subsection 3.2) and similarly for g . Then, note that f is free everywhere but in its typing assumption: we can quantify it to $\exists f. f \in \mathbb{B} \Rightarrow \mathbb{B}$ using the **LeftExists** rule from first-order logic. Now, this statement is provable, as it can be deduced from the non-emptiness of \mathbb{B} . Formally, we obtain the following proof:

$$\frac{\frac{f \in \mathbb{B} \Rightarrow \mathbb{B}, f x \in \mathbb{B}, g x \in \mathbb{B}, L \vdash R \quad \frac{\dots}{x \in \mathbb{B}, f \in \mathbb{B} \Rightarrow \mathbb{B} \vdash f x \in \mathbb{B}}{\text{Cut}}}{f \in \mathbb{B} \Rightarrow \mathbb{B}, g x \in \mathbb{B}, L \vdash R} \quad \vdots \quad \frac{\frac{f \in \mathbb{B} \Rightarrow \mathbb{B}, L \vdash R}{\exists f. f \in \mathbb{B} \Rightarrow \mathbb{B}, L \vdash R} \text{LeftExists} \quad \frac{\dots}{\vdash \exists f. f \in \mathbb{B} \Rightarrow \mathbb{B}} \text{Cut}}{L \vdash R} \text{Cut}$$

This example covers statements corresponding to type judgement and typing context. In general, there are three kinds of context formulas we need to eliminate: lambda definitions, variable type assignments, and type variables' non-emptiness. We implement a proof tactic called **CLEAN** which eliminates context formulas iteratively:

18:10 Mechanized HOL Reasoning in Set Theory

1. Find in the context a definition $def(\lambda_i)$ such that λ_i does not appear anywhere else. Then, using `LeftExists`, generalize the left-hand side to $\exists\lambda_i.def(\lambda_i)$. Prove $\exists\lambda_i.def(\lambda_i)$. This is always possible using the adequate type non-emptiness assumption. Eliminate $\exists\lambda_i.def(\lambda_i)$ using the `Cut` rule. Iterate on the next definition.
2. Find a variable type assignment $x \in T$. Using `LeftExists`, generalize to $\exists x.x \in T$. Using the type variable's non-emptiness assumptions, prove that $\exists x.x \in T$ (i.e. T is non-empty). Eliminate $\exists x.x \in T$. Iterate on next unused variable.
3. Find a non-emptiness assumption $A \neq \emptyset$ for a type variable that does not appear anywhere else. Using `LeftExists`, generalize to $\exists A.A \neq \emptyset$, which is of course provable without assumption, and eliminate it. Iterate on the next unused type variable.

We make every tactic that possibly eliminates subterms (that is, `TRANS`, `ABS`, `EQ_MP`, `INST` and `INSTTYPE`) call `CLEAN` to eliminate lingering assumptions.

Simulating HOL steps

We briefly hint at how steps of HOL can be simulated in FOST, leaving implicit concerns regarding proofs of type checking and context elimination, which were addressed above.

- `REFL` is simulated with Correctness of E and reflexivity of first-order equality.
- `TRANS` is similarly simulated with Correctness of E and transitivity of first order equality. Note that in HOL Light, `TRANS` requires only alpha-equivalence of the shared terms of the two premises. We explain how this can be handled without assuming that all alpha-equivalent expressions are represented by the same λ_i in the next paragraph.
- `MK_COMB` is simulated with Correctness of E and substitution of equals for equals in first-order logic.
- `ABS` and `ETA` follow from the definition of the λ_i and from Functional Extensionality (Lemma 3.8).
- `BETA` steps are proven directly from the definition of the λ_i .
- `ASSUME` is simply a Hypothesis step in Sequent calculus.
- `EQ_MP` is simulated with Correctness of E and substitution of equals. Similar to `TRANS`, it is subsequently made to support alpha-equivalence.
- `DEDUCT_ANTISYM_RULE` steps are proven with Propositional Extensionality.
- `INST` follows from instantiation of free variables in first order logic, except that doing so changes the shape of embeddings of abstractions to a non-canonical representation, which need to be transformed back into a canonical representation. We explain this mechanism in detail in the following paragraph.
- `INSTTYPE` corresponds to instantiation of free variables, but doing so breaks that assumption about which λ_i represents the term. Just as for `INST`, we then need to substitute the result for its canonical form, but this is simpler.

Alpha Equivalence

The steps `TRANS` and `EQ_MP` each take 2 premises with the added requirement that they share some subterm. For concreteness, consider the `TRANS` step:

$$\frac{\Gamma \vdash s =_A t \quad \Delta \vdash t =_A u}{\Gamma, \Delta \vdash s =_A u} \text{ TRANS}$$

In HOL Light, the two terms t_1 and t_2 in the premises are required to be identical *up to alpha equivalence*. However, alpha equivalence does not naturally hold in our encoding: two alpha-equivalent abstractions may be represented by different variables λ_i from Definition 3.1. (In fact, in the absence of memoization, even two occurrences of the same lambda can be

represented by different variables. In practice, for our import from HOL Light we perform memoization in the constructor of abstractions $\lambda(x:\text{Var}, \text{body}:\text{Term})$ using de Bruijn indices so that alpha equivalent terms HOL terms are mapped to the exact same FOST expression FOST for efficiency. That said, we still wish to show how to support alpha equivalence as a rule.

For concreteness, consider symbols λ_1 and λ_2 , representing abstractions, with the definitions:

$$(\lambda_1 \in A \Rightarrow A) \wedge (\forall x \in A. \lambda_1 x = x)$$

$$(\lambda_2 \in A \Rightarrow A) \wedge (\forall y \in A. \lambda_2 y = y)$$

Here, we can use the fact that our local definitions of lambda terms ensure not only existence, but also uniqueness. In particular, under those two assumptions, $\lambda_1 = \lambda_2$ is a consequence of the extensionality of set-theoretic functions. In fact, using the Eta axiom from HOL (implemented instead as a deduction step **ETA**), alpha equivalence is provable and does not need to be assumed.

► **Definition 3.9** (Tactic for Alpha-Conversion). *Let $_TRANS$ and $_EQ_MP$ be restrictions of $TRANS$ and EQ_MP not supporting alpha-equivalence. We implement a tactic*

$$\frac{}{\vdash \lambda x. t = \lambda y. t[x := y]} \text{ALPHA_CONV}$$

where

$$\text{ALPHA_CONV } x \ y \ t = _TRANS(\text{ETA } y (\lambda x. t)) (\text{ABS}(\text{INST}(\text{BETA } x \ t) \ x \ y) \ y)$$

Note that the first argument of $_TRANS$ proves $\lambda x. t = \lambda y. (\lambda x. t) y$ and the second proves $\lambda y. (\lambda x. t) y = \lambda y. t[x := y]$

We can then define a tactic proving the following:

$$\frac{}{\vdash t = u} \text{ALPHA_EQUIVALENCE} \text{ (if } t \text{ and } u \text{ are alpha-equivalent)}$$

which proves the equality by applying ALPHA_CONV recursively on t and u . Finally, we can define the complete versions $TRANS$ and EQ_MP , which apply ALPHA_EQUIVALENCE if the shared terms in the input are not strictly equal.

INST Proof Step and Normalization

It may seem at first glance that **INST** is a very easily simulated step: first-order logic admits instantiation of free variables across a sequent (in fact, Lisa offers this as a built-in proof step). This however fails to preserve the structure of the embedding. For concreteness again, let $x : A, y : A, p : B$ and consider the following simple provable HOL statement and its embedding in FOST:

$$\begin{aligned} S &= \vdash (\lambda x. p =_B p) y \\ (S) &= p \in B, y \in A, \text{def}_{\lambda_1} \vdash (\lambda_1 p y) = \top \end{aligned}$$

where $\text{def}_{\lambda_1} = (\lambda_1 \in B \Rightarrow A \Rightarrow B) \wedge (\forall p \in B. \forall x \in A. \lambda_1 p x = E(B) p p)$. Now, suppose $f :: B \Rightarrow B$ is also a variable and consider the effect of the instantiation $p := (f p)$ on the HOL sequent:

$$S_{[p := (f p)]} = \vdash (\lambda x. (f p) =_B (f p)) y$$

18:12 Mechanized HOL Reasoning in Set Theory

and on the embedded FOL sequent:

$$\llbracket S \rrbracket_{[p:=(fp)]} = (fp) \in B, y \in A, def_{\lambda_1} \vdash (\lambda_1 (fp) y) = \top$$

But on the other hand, we have

$$\llbracket S_{[p:=(fp)]} \rrbracket = p \in B, f \in B \Rightarrow B, y \in A, def_{\lambda_2} \vdash (\lambda_2 f p y) = \top$$

where

$$def_{\lambda_2} = (\lambda_2 \in (B \Rightarrow B) \Rightarrow B \Rightarrow A \Rightarrow \mathcal{B}) \wedge (\forall f \in (B \Rightarrow B). \forall p \in B. \forall x \in A. \lambda_2 f p x = E(B) p p) \text{ .}$$

So, instantiation and embedding do not commute. Moreover, the shape of $S_{[p:=(fp)]}$ does not correspond to the canonical specification of the embedding of HOL terms described in Definition 3.1.

► **Definition 3.10.** *(The embedding of) an abstracted term t is in closure-canonical form if it is of the form $\lambda_i x y z \dots$ where $x, y, z \dots$ are the free variables of t and λ_i is a symbol whose local context is as defined by Definition 3.4. A term is in closure-canonical form if all its subterms are in closure-canonical form.*

$\lambda_2 f p x$ is in closure-canonical form, as any term produced by Definition 3.1, but $\llbracket S_{[p:=(fp)]} \rrbracket$ is not, because the subterm $\lambda_1 (fp) y$ is not in canonical form. Hence, even though it denotes an equivalent statement, $S_{[p:=(fp)]}$ is not a legal expression whose shape other proof tactics expect to receive. To solve this, we implemented a recursive tactic that recursively transforms any non-canonical representation of an HOL term into its closure-canonical form and prove equality between the two. This then allows the `INST` tactic to output a statement in canonical form.

This concludes our simulation of the various proof tactics in FOST leading to Theorem 3.7.

► **Corollary 3.11.** *Let s be an HOL sequent. Then a proof of s in HOL can be transformed in a proof of $\llbracket s \rrbracket$ in FOST.*

We have implemented the transformation and the above tactics in Lisa.

3.4 Defining new constants

HOL Light allows the introduction of new definitions which serve as shorthand for existing terms. This is also possible in Lisa, wherein if we produce a theorem of the form $\exists! x. P(x)$, we obtain a new constant c and the property $\forall x. P(x) \iff x = c$. However, for this extension to be sound, the definition of a constant can not contain free variables, as otherwise defining $c := x$ would allow proving $\forall x. c = x$. For definitions from HOL, consider for example the term defining the universal quantifier `!` in HOL Light (`bool.ml`: 243):

$$\lambda P : A \rightarrow \mathcal{B}. P = \lambda x : A. \top$$

It is represented as a variable λ_2 while the subterm $\lambda x : A. \top$ is represented by a symbol λ_1 . As in the elimination of contexts, we can prove $\exists! \lambda_2. \text{ctx}(\lambda_2)$, which matches the requirement of extension by definition. The type variable A is reflected as an explicit parameter of the constant (which means that `!` is an applied function symbol `!(A)`). However, λ_1 will be free in $\text{ctx}(\lambda_2)$, so we need to bundle with the definition of λ_2 all the context definitions that it refers to and prove existence and uniqueness accordingly.

It is easy to prove that such a symbol exists under the assumption of the usual context:

$$\text{ctx}(\lambda_1), \text{ctx}(\lambda_2) \vdash \exists!c. c = \lambda_2$$

However when quantifying all assumptions, this will only yield

$$\forall\lambda_1. \text{ctx}(\lambda_1) \implies \forall\lambda_2. \text{ctx}(\lambda_2) \implies \exists!c. c = \lambda_2$$

while the mechanism of extension by definition requires the $\exists!$ quantifier to be the top-level quantifier in the definition. We use an additional FOL theorem that allows us to swap the universal and unique-existential quantifiers

$$\exists!x.P(x) \implies ((\forall x.P(x) \implies \exists!y.Q(x,y)) \iff (\exists!y.\forall x.P(x) \implies Q(x,y)))$$

This fact, alongside proofs that the terms λ_i are uniquely defined by their contexts, we can swap the quantifiers one-by-one to produce the final justification for the definition:

$$\exists!c.\forall\lambda_1. \text{ctx}(\lambda_1) \implies \forall\lambda_2. \text{ctx}(\lambda_2) \implies c = \lambda_2 .$$

We generate this proof automatically in our implementation. The proof of typing is generated alongside the symbol by type checking the definition. The theorem corresponding to the definition under appropriate context $\text{ctx}(\lambda_1), \text{ctx}(\lambda_2) \vdash !(A) = \lambda_2$ is also generated automatically.

4 Formalizing Algebraic Data Types

We want to bring the benefits of types into FOST. In particular, algebraic data types are useful when reasoning about inductive data structures such as lists or trees. Their encoding is generally hidden to the user, who only obtains access to their characteristic theorems and definitional mechanisms. We want Lisa to incorporate such mechanisms. Even though ADTs can be encoded within HOL [20], we choose instead for Lisa's implementation to directly define them in FOST. We therefore avoid going through an intermediate encoding but also lay the foundations of further generalization. We start by giving a syntactic definition of algebraic data types.

► **Definition 4.1** (Algebraic data types). *An algebraic data type in set theory is a set A equipped with a finite set of functions, $c_i : T_1^i \Rightarrow \dots \Rightarrow T_n^i \Rightarrow A$, referred to as constructors. All elements of A have to be in the image of one of exactly one of its constructors. T_j^i can refer to A itself, giving algebraic data types their recursive behaviour.*

We assume for simplicity and without loss of generality that each constructor has exactly n arguments. m refers to the number of constructors in the datatype specification.

We want to allow defining and reasoning about ADTs directly in FOST. Specifically, consider an ADT specification $\{c_i : (S_1^i, \dots, S_n^i)\}_{i \leq m}$, where S_j^i is either a term with no variables, or a special symbol recursively referring to the ADT itself. We want to output a set A and functions $\{c_i\}_{i \leq m}$ with the following properties:

- (Typing) For all $i \leq m$, $c_i \in (S_1^i \Rightarrow \dots \Rightarrow S_n^i \Rightarrow A)$
- (Injectivity 1) Every c_i is injective.
- (Injectivity 2) For $x, y \in A$, if $x \in \text{Im}(c_i)$, $y \in \text{Im}(c_j)$ and $i \neq j$ then $x \neq y$
- (Structural induction) A is the smallest set closed under the constructors c_i 's. This allows us to write proofs by induction on the structure of the ADT.

► **Example 4.2.** Consider the type of boolean linked lists, with specification

$$\text{listbool} = \{\text{nil} : (), \text{cons} : (\mathbb{B}, \text{listbool})\}$$

We want to generate a set listbool and constructors nil and cons such that $\text{nil} \in \text{listbool}$ and $\text{cons} \in \mathbb{B} \Rightarrow \text{listbool} \Rightarrow \text{listbool}$. The above properties should hold; for instance, $\text{cons} \top \text{nil} \neq \text{cons} \perp \text{nil}$.

We next present our formalization as implemented in Lisa within FOST. We represent an algebraic data type as a set A of tuples containing the tag of the constructor and the arguments given to it. This ensures that elements of A are in the image of exactly one constructor. For the listbool example, $\text{cons} \top \text{nil}$ is hence represented as the tuple $(\text{tag}_{\text{cons}}, \top, (\text{tag}_{\text{nil}}))$. In this setting, tags are arbitrary terms that differentiate constructors. They can, for example, be natural numbers or some encoding of the name of the constructor. We define the set A as the least fixpoint of the function

$$F(H) = \bigcup_{i \leq m} \left\{ (\text{tag}_{c_i}, x_1, \dots, x_n) \mid x_k \in \begin{cases} H & \text{if } S_k^i \text{ is a self-reference} \\ S_k^i & \text{otherwise} \end{cases} \right\}$$

The existence of $F(S)$ for every set S is guaranteed by the replacement and the union axioms. In order to characterize the least fix point of F , we use the recursion theorem schema of ZF to obtain a unique function f with domain \mathbb{N} (which is also the smallest infinite ordinal ω) such that

$$f(0) = \emptyset$$

$$\forall a \in \mathbb{N}. f(a+1) = F(f(a))$$

Intuitively, $f(a)$ corresponds to all instances of A of height smaller than or equal to a . For example, for lists of Booleans, $f(2)$ is the set

$$\{(\text{tag}_{\text{nil}}), (\text{tag}_{\text{cons}}, \top, (\text{tag}_{\text{nil}})), (\text{tag}_{\text{cons}}, \perp, (\text{tag}_{\text{nil}}))\}$$

► **Lemma 4.3.** *The class function F admits a least fixpoint given by $A := \bigcup_{n \in \omega} f(n)$.*

Proof. If $x_k \in A$ then there is a $a_k \in \omega$ such that $x_k \in f(a_k)$. Since F is monotonic $x_k \in f(\max_{x_k \in A} a_k)$. Therefore, for every $(\text{tag}_{c_i}, x_1, \dots, x_n) \in F(A)$, we have $(\text{tag}_{c_i}, x_1, \dots, x_n) \in f(\max_{x_k \in A} a_k + 1) \subseteq A$. ◀

► **Definition 4.4.** *We define c_i as the function in $S_1^i \Rightarrow \dots \Rightarrow S_n^i \Rightarrow A$ such that*

$$c_i x_1 \dots x_n = (\text{tag}_{c_i}, x_1, \dots, x_n)$$

Now that we have a formal definition of A and c_i , we prove that they fulfil the above properties.

► **Theorem 4.5.** *Let A and $\{c_i\}_{i \leq m}$ constructed as above. The following statements hold in FOST*

$$x_1 \in S_1^i, \dots, x_n \in S_n^i \vdash c_i x_1 \dots x_n \in A \quad (\text{Typing})$$

$$\bigwedge_{k \leq n} x_k \in S_k^i \wedge y_k \in S_k^i, c_i x_1 \dots x_n = c_i y_1 \dots y_n \vdash \bigwedge_{k \leq n} x_k = y_k \quad (\text{Injectivity 1})$$

$$\bigwedge_{k \leq n} x_k \in S_k^i \wedge y_k \in S_k^j, i \neq j \vdash c_i x_1 \dots x_n \neq c_j y_1 \dots y_n \quad (\text{Injectivity 2})$$

Proof. For typing, we have $c_i x_1 \dots x_n = (\text{tag}_{c_i}, x_1, \dots, x_n) \in F(A) = A$.

We know that tuples are injective, that is

$$\vdash (\text{tag}_{c_i}, x_1, \dots, x_n) = (\text{tag}_{c_j}, y_1, \dots, y_n) \iff \bigwedge_{k \leq n} x_k = y_k \wedge \text{tag}_{c_i} = \text{tag}_{c_j}$$

Moreover, tuples of different arities are not equal. Injectivity 1 follows from the forward implication, while Injectivity 2 from the backward one and the fact that tags are uniquely assigned to constructors. \blacktriangleleft

► **Theorem 4.6.** *Structural induction schema holds on A .*

$$\bigwedge_{i \leq m} \left(\forall x_1^i \in S_1^i. \hat{P}(x_1^i) \implies \dots \implies \forall x_n^i \in S_n^i. \hat{P}(x_n^i) \implies P(c_i x_1^i \dots x_n^i) \right) \vdash \forall x \in A. P(x)$$

$$\text{where } \hat{P}(x_k^i) = \begin{cases} P(x_k^i) & \text{if } S_k^i = A \\ \top & \text{if } S_k^i \neq A \end{cases}$$

Proof. We first show that for every $a \in \mathbb{N}$, the theorem holds when replacing A by $f(a)$. This follows by induction on a . Since by the definition of A , every $x \in A$ is in $f(a)$ for some a , the statement holds for every element of A . \blacktriangleleft

Polymorphic algebraic data types

Algebraic data types can be polymorphic, meaning that the specification of the constructors contain type parameters. This allows, for example, reasoning over generic lists instead of lists of a specific type. We extend our mechanization of ADTs to support such polymorphism. To do so, we generalize A and $\{c_i\}$ to be class functions instead of constant symbols.

Formally, let $\{c_i : (S_1^i, \dots, S_n^i)\}_{i \leq m}$ be the specification of an algebraic data type that possibly contains variable symbols X_1, \dots, X_l . We define $A(X_1, \dots, X_l)$ as the fix point of $F(X_1, \dots, X_l)$ where the construction of F carries the same behaviour as above. Using this definition, all the properties of ADTs are preserved, and the construction is essentially the same.

As in Subsection 3.2, we give a top level polymorphic type to the function symbols A and c_i , so that they can similarly be type checked. This also implies that all the tactics from the previous section are compatible with terms referring to some ADT A and its constructors. To conclude, we show an example of polymorphic lists in Lisa.

► **Example 4.7.** The user can define polymorphic lists with the following syntax:

```
1  val define(list: ADT[1], constructors(nil, cons)) =
2  T --> (() | (T, list))
```

where `list`, `nil`, and `cons` are new function symbols. `ADT[1]` represents ADTs with one type parameter. `list` is such that `list(T)` is a set containing all lists over `T`, `nil(T)` is a constructor taking one type parameter and no argument, and `cons(T)` is a constructor taking one type parameter `T` as well as an element of type `T` and an element of `list(T)` as arguments. This declaration also automatically proves Theorem 4.5 and Theorem 4.6 for this specification. Our typing tactic from Subsection 3.2 can use these properties to type check any expressions containing `list`, `nil`, and `cons`.

5 Importing Proofs from HOL Light

While the embedding of HOL as described above allows writing HOL proofs directly in Lisa, we also implement a prototype to attempt the automatic import of theorems and definitions from HOL Light. We chose HOL Light for our import due to its simple foundations, its large library and easily accessible proof export. With some additional work in matching proof steps, the same method can be adapted to other members of the HOL family.

Since the HOL Light kernel does not keep track of proof objects by default, we rely on the `ProofTrace` export system packaged in the HOL Light repository [27]. The system provides a patch to the HOL Light kernel to track every proven statement in the system's execution. The stored proofs are finally exported to JSON files. We modified the existing JSON output syntax slightly to allow its automatic import using standard JSON libraries for Scala. The terms are exported as strings with a simple and unambiguous grammar, and are parsed back by Lisa. After reading the JSON files, the proofs (with indexed steps) are transformed into proof DAGs in an intermediate representation, and finally transformed to Lisa theorems.

Given the Lisa tactics we developed for this purpose, the translation of HOL Light theorems is straightforward, and proceeds by recursing on the proof DAG obtained from HOL Light, translating each proof step to the equivalent Lisa tactic call. Although constant definitions also appear as a single proof step, `DEFINITION`, they must be dealt with separately, as in Subsection 3.4.

After defining a polymorphic constant, we change its signature compared to the HOL Light version. For example, the universal quantifier $!: (A \rightarrow \text{bool}) \rightarrow \text{bool}$ becomes a class function `!` such that $\forall A. !(A) \in (A \Rightarrow \mathcal{B}) \Rightarrow \mathcal{B}$. On subsequent occurrences of `!` in the import, it occurs with an ascribed type, say $!: (\tau \rightarrow \text{bool}) \rightarrow \text{bool}$. This instantiated type is matched against the original, polymorphic type to find the substitution $A \mapsto \tau$. The definition is correspondingly instantiated, and the occurrence is replaced by the Lisa term `!(τ)`.

The embedding described here produces a large overhead in proof length. The proofs of the first 15 named definitions and theorems as defined in the HOL Light library, involve 1716 HOL Light kernel steps. These are expanded to just over 300,000 Lisa kernel steps, with the reconstruction and verification taking 93 seconds on a laptop running Linux with an i7-1165G7 CPU and 16GB RAM.

However, if we trust theorems from HOL Light and proof rechecking, any theorem from HOL Light Library can be quickly imported with the encoding of Section 3. Then, the first 70 definitions and theorems about logical quantifiers and inductive reasoning are parsed, translated and imported in about 6 seconds instead. A similar approach could also potentially serve as a way to bootstrap the mathematical library and create a theory skeleton in the target system (Lisa) to accelerate further development.

6 Conclusion

We have demonstrated how to embed HOL into conventional first-order logic axiomatization of set theory. Our translation maintains local definitions (at the level of the sequent) of the closure of abstraction terms. We showed that this encoding allows simulating all the core proof steps of higher-order logic. We mechanized this encoding in Lisa, and obtained an interface and tactics for reasoning with typed expressions and set-theoretic functions. We then demonstrated how (possibly polymorphic) ADTs can be mechanized in first-order set theory, and that their representation is compatible with the tactics and type checking we developed for HOL functions.

We also considered alternative encodings of lambda terms. Instead of defining lambdas at the level of the whole sequent, we could place the definition right after the first predicate symbol. In particular, definitions of lambdas become nested, instead of being independent. On one hand, this means that we do not have to compute closures. On the other hand, the defining property of a lambda would often be deep in a formula, and its use would require deconstructing and reconstructing the formula to use the context. Alternatively, we could use an embedding of λ -terms based on combinators from combinatory logic [4]. We did not use fixed combinators such as SKI due to growth in formula size; in the future we may explore the use of parametric combinator families.

While the results we obtain are of practical use and we expect them to become part of the standard Lisa release, the encoding is somewhat complicated and even if most of the machinery can be hidden, it may confuse non-expert users. There is a significant overhead in the size of translated proofs, which is less than ideal in practice. This also prevented us from translating and rechecking a larger number of proofs from the HOL Light library. The syntactic restrictions on terms of FOL is the main source of complexity in the translation. For Lisa, this suggests considering extensions of FOL with terms that refer to formulas, such as the definite description operator $\iota x.P$, denoting an individual that is uniquely characterized by the predicate P .

References

- 1 Oskar Abrahamsson, Magnus O. Myreen, Ramana Kumar, and Thomas Sewell. Candle: A Verified Implementation of HOL Light. In *DROPS-IDN/v2/Document/10.4230/LIPIcs.ITP.2022.3*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.ITP.2022.3.
- 2 Sten Agerholm and Mike Gordon. Experiments with zf set theory in hol and isabelle. In E. Thomas Schubert, Philip J. Windley, and James Alves-Foss, editors, *Higher Order Logic Theorem Proving and Its Applications*, pages 32–45, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.
- 3 Rob Arthan. HOL Constant Definition Done Right. In Gerwin Klein and Ruben Gamboa, editors, *Interactive Theorem Proving*, Lecture Notes in Computer Science, pages 531–536, Cham, 2014. Springer International Publishing. doi:10.1007/978-3-319-08970-6_34.
- 4 Hendrik Pieter Barendregt, Wil Dekkers, and Richard Statman. *Lambda Calculus with Types*. Perspectives in Logic. Cambridge University Press, 2013.
- 5 Chad Brown. *The Egal Manual*, 2014.
- 6 Chad E. Brown. Combining type theory and untyped set theory. In Ulrich Furbach and Natarajan Shankar, editors, *Automated Reasoning*, pages 205–219, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.
- 7 Chad E. Brown, C. Kaliszyk, and Karol Pak. Higher-Order Tarski Grothendieck as a Foundation for Formal Proof. In *ITP*, 2019. doi:10.4230/LIPIcs.ITP.2019.9.
- 8 Mario M Carneiro. Conversion of hol light proofs into metamath. *Journal of Formalized Reasoning*, 9(1):187–200, January 2016. doi:10.6092/issn.1972-5787/4596.
- 9 G. Gentzen. Untersuchungen über das logische Schließen I. *Mathematische Zeitschrift*, 39:176–210, 1935.
- 10 Mike Gordon. Set theory, higher order logic or both? In W. Brauer, D. Gries, J. Stoer, Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, Joakim von Wright, Jim Grundy, and John Harrison, editors, *Theorem Proving in Higher Order Logics*, volume 1125, pages 191–201. Springer Berlin Heidelberg, Berlin, Heidelberg, 1996. doi:10.1007/BFb0105405.
- 11 Simon Guilloud. LISA Reference Manual. EPFL-LARA, February 2023.
- 12 Simon Guilloud, Sankalp Gambhir, Andrea Gilot, and Viktor Kunčák. epfl-lara/lisa. Software, version 0.1., swhId: swh:1:dir:c3f6e63aa274ed7ea292efcb0eade3f4abb60d2a (visited on 2024-08-21). URL: <https://github.com/epfl-lara/lisa/tree/itp2024-archive>.

- 13 Simon Guilloud, Sankalp Gambhir, and Viktor Kuncak. LISA – A Modern Proof System. In *14th Conference on Interactive Theorem Proving*, Leibniz International Proceedings in Informatics, pages 17:1–17:19, Bialystok, 2023. Dagstuhl.
- 14 John Harrison. HOL Light: An Overview. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics*, volume 5674, pages 60–66. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. doi:10.1007/978-3-642-03359-9_4.
- 15 Thomas Jech. *Set theory: The third millennium edition, revised and expanded*. Springer, 2003.
- 16 Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.
- 17 Cezary Kaliszyk and Karol Pąk. Combining higher-order logic with set theory formalizations. *Journal of Automated Reasoning*, 67(2):20, May 2023. doi:10.1007/s10817-023-09663-5.
- 18 Alexander Krauss and Andreas Schropp. A mechanized translation from higher-order logic to set theory. In Matt Kaufmann and Lawrence C. Paulson, editors, *Interactive Theorem Proving*, pages 323–338, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.
- 19 Kenneth Kunen. *Set Theory An Introduction To Independence Proofs*. North Holland, Amsterdam Heidelberg, reprint edition edition, December 1983.
- 20 Thomas F. Melham. *Automating Recursive Type Definitions in Higher Order Logic*, pages 341–386. Springer New York, New York, NY, 1989. doi:10.1007/978-1-4612-3658-0_9.
- 21 Elliott Mendelson. *Introduction to Mathematical Logic*. Springer US, Boston, MA, 1987. doi:10.1007/978-1-4615-7288-6.
- 22 Leonardo de Moura and Sebastian Ullrich. The lean 4 theorem prover and programming language. In André Platzer and Geoff Sutcliffe, editors, *Automated Deduction – CADE 28*, pages 625–635, Cham, 2021. Springer International Publishing.
- 23 Adam Naumowicz and Artur Kornilowicz. A Brief Overview of Mizar. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, volume 5674, pages 67–72, August 2009. doi:10.1007/978-3-642-03359-9_5.
- 24 Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002. doi:10.1007/3-540-45949-9.
- 25 Steven Obua. Partizan games in Isabelle/HOLZF. In *Theoretical Aspects of Computing - ICTAC 2006*, pages 272–286, November 2006. doi:10.1007/11921240_19.
- 26 Steven Obua, Jacques Fleuriot, Phil Scott, and David Aspinall. Type inference for zfh. In Manfred Kerber, Jacques Carette, Cezary Kaliszyk, Florian Rabe, and Volker Sorge, editors, *Intelligent Computer Mathematics*, pages 87–101, Cham, 2015. Springer International Publishing.
- 27 Stanislas Polu. HOL Light / ProofTrace: Modern proof steps recording for hol light. <https://github.com/jrh13/hol-light/tree/master/ProofTrace>. Accessed: 2024-03-18.
- 28 Konrad Slind and Michael Norrish. A Brief Overview of HOL4. In Otmane Ait Mohamed, César Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics*, pages 28–32, Berlin, Heidelberg, 2008. Springer. doi:10.1007/978-3-540-71067-7_6.
- 29 The Coq Development Team. The coq proof assistant, July 2023. doi:10.5281/zenodo.8161141.

Modular Verification of Intrusive List and Tree Data Structures in Separation Logic

Marc Hermes   

Radboud University, Nijmegen, The Netherlands

Robbert Krebbers   

Radboud University, Nijmegen, The Netherlands

Abstract

Intrusive linked data structures are commonly used in low-level programming languages such as C for efficiency and to enable a form of generic types. Notably, intrusive versions of linked lists and search trees are used in the Linux kernel and the Boost C++ library. These data structures differ from ordinary data structures in the way that nodes contain only the meta data (*i.e.* pointers to other nodes), but not the data itself. Instead the programmer needs to embed nodes into the data, thereby avoiding pointer indirections, and allowing data to be part of several data structures.

In this paper we address the challenge of specifying and verifying intrusive data structures using separation logic. We aim for modular verification, where we first specify and verify the operations on the nodes (without the data) and then use these specifications to verify clients that attach data. We achieve this by employing a representation predicate that separates the data structure’s node structure from the data that is attached to it. We apply our methodology to singly-linked lists – from which we build cyclic and doubly-linked lists – and binary trees – from which we build binary search trees. All verifications are conducted using the Coq proof assistant, making use of the Iris framework for separation logic.

2012 ACM Subject Classification Theory of computation → Separation logic; Theory of computation → Hoare logic; Theory of computation → Programming logic; Theory of computation → Data structures design and analysis

Keywords and phrases Separation Logic, Program Verification, Data Structures, Iris, Coq

Digital Object Identifier 10.4230/LIPIcs.ITP.2024.19

Supplementary Material *Software (Code)*: <https://doi.org/10.5281/zenodo.12575047> [10]

Acknowledgements We thank Derek Dreyer, Laila Elbeheiry, Deepak Garg, Jules Jacobs, Ike Mulder and Michael Sammler for discussions, and the anonymous reviewers for their feedback. This research was supported in part by a generous award from Google Android Security’s ASPIRE program.

1 Introduction

Linked data structures such as lists and trees are pervasive in imperative programming and serve as the implementation for various abstract data types such as queues, stacks, deques, sets and maps. Verification of these data structures therefore received a considerable amount of attention in the literature – *e.g.* the seminal papers on separation logic [30, 28] use linked lists and trees as their key examples, and many papers on verification tools use linked data structures as case studies [3, 4, 8, 9, 12, 27]. Yet, there is an unfortunate discrepancy between the way linked data structures are studied in the literature and the way they are implemented in systems programming, *e.g.* the Linux kernel [16, 23] and the Boost C++ library [20].

Let us first review the naive way to represent singly-linked lists in C that are “generic” in the element type:

```
1 struct list {
2     void* data;
3     struct list* next;
4 };
```



© Marc Hermes and Robbert Krebbers;

licensed under Creative Commons License CC-BY 4.0

15th International Conference on Interactive Theorem Proving (ITP 2024).

Editors: Yves Bertot, Temur Kutsia, and Michael Norrish; Article No. 19; pp. 19:1–19:18

Leibniz International Proceedings in Informatics



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

19:2 Modular Verification of Intrusive List and Tree Data Structures in Separation Logic

Linked-list nodes contain the `data` and a pointer to the `next` node (the `NULL` pointer is used to represent the empty list). To avoid fixing the element type, the field `data` is a `void` pointer, meaning it could point to data of arbitrary type. This naive way of representing lists has a number of drawbacks in systems programming where efficiency is a key concern.

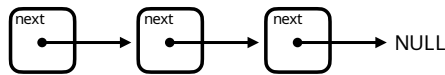
Motivation #1. The use of a pointer indirection for `data` requires additional storage and incurs a run-time cost on every read. This is in contrast to the “nongeneric” version where the data is stored directly in the struct:

```
1 struct int_list {
2     int data;
3     struct int_list* next;
4 };
```

Defining a specific version of linked lists for each element type is clearly undesirable – it means one has to duplicate all methods from the list API for each element type. Modern programming languages such as C++ and Rust offer generics and monomorphization to address this problem. It is also possible to obtain efficient data types with generic elements in plain C, which we will illustrate in the following.

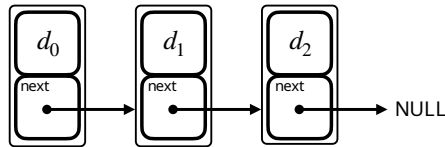
The Linux kernel uses the **intrusive** approach to linked lists, allowing for lists that can be re-used for different data types, while avoiding the overhead caused by pointer indirections. This is achieved by separating the meta data (*i.e.* the pointers to node) from the data. One first defines a `node` structure which is used to achieve the necessary linking:

```
1 struct node {
2     struct node* next;
3 };
```



The list API can now be developed for the `node` structure, independently of the data. These nodes can then be incorporated as fields in other structures to create lists with values of any desired element type. Here we show the instantiation with integer values:

```
1 struct intrusive_int_list {
2     int data;
3     struct node node;
4 };
```



Compared to the naive approach, the functions implemented for the `node` API can be used to operate on lists with attached data, no matter the type of the data. To further illustrate this, we consider the function `replace_at`, which replaces the n -th element of an `intrusive_int_list` with a new integer value. In the implementation, we first define a function `get_pos` which yields a pointer to the n -th position in the `node` structure, and then use this function in `replace_at` to make the replacement at the correct position:

```
1 struct node* get_pos(int n, struct node* v) {
2     if (n == 0 || v == NULL) return v;
3     return get_pos(n-1, v->next);
4 }
5
6 #define container_of(ptr, container, field)
7     (container*)((unsigned char*)(ptr) - offsetof(container, field))
8
9 void replace_at(int n, struct intrusive_int_list* l, data a) {
10    struct node* pos = get_pos(n, &l->node);
11    if (pos == NULL) return;
12    struct intrusive_int_list* lp =
```



```

IntList : val → list int → iProp
IntList v []      ≜ v = None
IntList v (d :: D) ≜ ∃(l : loc). v = Some l * ∃v'. l ↦ [d, v'] * IntList v' D

```

Here, the notation $l \mapsto [d, v']$, or more generally $l \mapsto [v_0, \dots, v_n]$, is an abbreviation for $l \mapsto v_0 * \dots * l + n \mapsto v_n$, which provides unique ownership of an array storing values v_0, \dots, v_n at the locations l to $l + n$. The connective $*$ is the separating conjunction, which is used to describe the disjointness of the resources. NULL pointers are modeled using the constructors `None` and `Some` of the option type.

Using the representation predicate, the specification for a function `replace_int_at` on `int_list` can be expressed in terms of a Hoare triple. We will use $\langle n := d \rangle D$ to denote the mathematical list obtained from D by replacing its n -th element with d . If n is larger than the length of the list, D remains unchanged.

$$\{ \text{IntList } v \ D \} \text{replace_int_at } n \ v \ d \{ \text{IntList } v \ (\langle n := d \rangle D) \}$$

While representation predicates are commonly used to describe non-intrusive structures like `int_list`, our goal is to formulate and prove similar specifications for intrusive data structures like `intrusive_int_list`. Additionally, we would like to achieve this in a way that also formally captures and makes use of the modularity which underlies the definition of `intrusive_int_list` and the implementation of `replace_at`.

To modularly verify `replace_at`, we should state and verify a specification for `get_pos`. This brings us to the key observation about giving a specification to intrusive structures: We should closely follow the definition of `intrusive_int_list`, and first isolate the intrusive `node` part of the structure:

```

Node : val → list loc → iProp
Node v []      ≜ v = None
Node v (l :: L) ≜ v = Some l * ∃v'. l ↦ v' * Node v' L,

```

In contrast to `IntList`, the above does not make reference to any list of data values and is instead linking locations taken from the list L . However, this now allows us to define a representation predicate for the intrusive structure `intrusive_int_list` in a straightforward yet novel way:

$$\text{IntrusiveIntList } v \ D \triangleq \exists L. \text{Node } v \ L * \bigstar_{l, d \in [l_0, \dots, l_n], [d_0, \dots, d_n]} (l - 1) \mapsto d$$

In the above, the big separating conjunction of the form $\bigstar_{l, d \in [l_0, \dots, l_n], [d_0, \dots, d_n]}$ runs over the pairs $(l_0, d_0), \dots, (l_n, d_n)$, implicitly requiring the two appearing lists to be of equal length. The definition of `IntrusiveIntList` clearly exposes the underlying intrusive structure in the form of the `Node` predicate. We can now easily give a specification for `get_pos`, which we can then use in the verification of `replace_at`:

$$\{ \text{Node } v \ L \} \text{get_pos } n \ v \{ x. x = \text{nth } n \ L * \text{Node } v \ L \}$$

$$\{ \text{IntrusiveIntList } v \ D \} \text{replace_at } n \ v \ d \{ \text{IntrusiveIntList } v \ (\langle n := d \rangle D) \}$$

In the above specification, `nth` n L returns `Some` l if l is the n -th element of the list L , and returns `None` if there is no n -th element. Given our definition of `IntrusiveIntList` the verification of `replace_at` is both straightforward and modular. We can easily make use of the specification of `get_pos`, since its precondition `Node` v L conveniently appears as the left part of the separating conjunction in the definition of `IntrusiveIntList`.

Summary of key idea. The example illustrates the key idea we want to push forward in this paper: A separation of concerns when specifying intrusively implemented data types. One concern is the “shape” in which data is supposed to be stored, here given by the `node` structure. The second concern is the actual data itself, which we can think of as being attached to the shape. The underlying shape is then used for navigation on the data. Functions that have been implemented on the shape can, and should, then be used in a modular way when implementing functions that operate on the data. The verification of the specifications should then turn out to be modular as well.

We demonstrate this methodology through two examples: intrusive lists (Section 2) and intrusive binary search trees (Section 3). In both cases, we give representation predicates to specify the intrusive structure and then show how to use them to specify mutable data structures that carry data. In the case of trees, we will implement a similar function to the above `get_pos`, to locate a specific key in the tree, and to obtain a pointer to the corresponding node. Since this function leaves us with a partially traversed tree, we need to deal with the orthogonal challenge of specifying partial trees (Section 3.3), for which we employ ‘the magic-wand as frame’ approach by Cao et al. [7] (which has previously only been applied to ordinary data structures, not intrusive ones).

To summarize, the main contributions of this paper are:

- We introduce a specification for Linux-like intrusive singly-linked lists and sequences in separation logic. We use the specification to modularly build up intrusive singly-linked cyclic lists (Section 2.3) and doubly-linked cyclic lists (Section 2.4), and use them to implement data structures that carry data with them (Section 2.5).
- We apply our approach by verifying locate and insertion operations of binary search trees, which are based on intrusive binary trees (Section 3). In extension to what is done by Cao et al. [7], we consider intrusive data structures, and our definition of partial trees incorporates the invariant of a binary search tree.

All of the included structures, their operations and specifications have been defined and verified [10] in the Coq proof assistant, by making use of the Iris framework [13, 14, 15, 17, 18, 19] for separation logic.

2 Intrusive List Structures

In this section, we will gradually and modularly build intrusive list data structures. These structures link together locations in the heap, and do not carry any data with them. Their API allows a user to attach new locations as nodes to the list. This means that the user is responsible for the allocation and deallocation of the nodes, and the list structure is only used to manage the nodes. This can then be used by the user to form lists keeping track of data, which we will showcase by implementing a deque data structure.

After covering some preliminaries concerning the programming language and separation logic (Section 2.1), we start with the implementation of simple intrusive sequences (Section 2.2), and give specifications for some standard operations. We then continue by modularly using sequences to build intrusive cyclic lists (Section 2.3) and doubly-linked cyclic lists (Section 2.4). Lastly, we illustrate how the intrusive doubly-linked cyclic lists can be complemented with data values to implement a deque data structure (Section 2.5).

2.1 Preliminaries

We briefly go over some specifics of the programming language and program logic that will subsequently be used. We work in a λ -calculus which includes let expressions, if-then-else expressions, matching on terms, equality checks and mutable arrays:

<code>alloc n v</code>	Allocates n successive locations in memory, initializes all of them with the value v , and returns the starting location of this array.
<code>free n l</code>	Deallocates n successive memory locations beginning from l .
<code>l ← v</code>	Assigns the value v to the location l .
<code>!l</code>	Retrieves the value at memory location l .

To reason about these operations, we utilize separation logic [30, 28], a variant of Hoare logic designed for imperative programs with pointers. Separation logic introduces the primitive $l \mapsto v$, separating conjunction $*$, and separating implication $-*$, which can be used to form assertions that are interpreted to describe fragments of the heap.

<code>emp</code>	An empty heap fragment.
<code>l ↦ v</code>	Describes a memory fragment in which location l contains the value v .
<code>P * Q</code>	Disjoint union of the fragments described by P and Q .
<code>P -* Q</code>	Describes a heap fragment which satisfies Q once it is combined with a disjoint fragment for which P holds.

We let `iProp` denote the set of separation logic assertions, differentiating it from the set of assertions `Prop` of the meta-logic (Coq). We allow pointer arithmetic on locations, meaning $l + n$ denotes the location n steps away from l . This allows us to describe arrays of values v_0, \dots, v_n in the heap, by the formula $l \mapsto [v_0, \dots, v_n] \triangleq l + 0 \mapsto v_0 * \dots * l + n \mapsto v_n$. We often encounter situations where we want to make the assumption that a certain location is non-empty. This is expressed by $\exists v. l \mapsto v$, which we abbreviate as $l \mapsto _$, and likewise extend the usage of the wildcard symbol “`_`” to arrays, as for example in $l \mapsto [_, _, v]$.

A key element in reasoning about program correctness within the framework of separation logic are Hoare triples. A Hoare triple is of the form $\{P\} e \{v. \Phi(v)\}$, where:

P	Assertion specifying the state of the heap before the execution of e .
e	An expression in the programming language being analyzed.
$\Phi(v)$	A predicate making an assertion about the return value v and describing the state of the heap after the execution of e .

The semantics of a Hoare triple is that if the initial state satisfies the precondition P , then the program e will not crash, and if it finishes executing, the return value and final state will satisfy the postcondition Φ . If the postcondition does not bind a return value, we simply write $\{P\} e \{Q\}$.

2.2 Sequences

A sequence [30] starting at location $l : \text{loc}$ links together a list $D : \text{list val}$ of values and stores a given default pointer $e : \text{loc}$ in the final node, giving its predicate the type signature $\text{loc} \rightarrow \text{list val} \rightarrow \text{loc} \rightarrow \text{iProp}$. A standard definition of lists is similar to this, but would restrict it by demanding the last pointer to be a NULL pointer. Since our goal is to specify intrusive structures, we decouple the values from the shape of the sequence and define a representation predicate `Seq.pred` with signature $\text{loc} \rightarrow \text{list loc} \rightarrow \text{loc} \rightarrow \text{iProp}$, which only links together a list of locations:

$$\text{Seq.pred } s [] e \triangleq s \mapsto e \qquad \text{Seq.pred } s (l :: L) e \triangleq s \mapsto l * \text{Seq.pred } l L e.$$

Our API for sequences includes a function to create a new intrusive sequence, push a new location to the front of the sequence, pop the first location, and to retrieve the next location in the sequence:

```

Seq.new start end := start ← end
Seq.push start new := new ← !start; start ← new
Seq.pop start := let rem = !start in start ← !rem; rem
Seq.next start := !start

```

As alluded to by the naming, the predicate and functions are enclosed in a module named `Seq`. If the context makes it clear enough, we abbreviate the representation predicate `Seq.pred` by `Seq`, and drop the module name in the function names. We do so for all data structures that follow. The specifications for the operations on sequences are as follows:

$$\begin{array}{l}
\{ s \mapsto _ \} \text{ new } s e \quad \{ \text{Seq } s \ [] e \} \\
\{ l \mapsto _ * \text{Seq } s \ L e \} \text{ push } s l \quad \{ \text{Seq } s \ (l :: L) e \} \\
\{ \text{Seq } s \ (l :: L) e \} \text{ pop } s \quad \{ p. p = l * \text{Seq } s \ L e * p \mapsto _ \} \\
\{ \text{Seq } s \ L e \} \text{ next } s \quad \{ p. p = \text{nth } 0 \ (L ++ [e]) * \text{Seq } s \ L e \}
\end{array}$$

Recall that `nth` n L returns the n -th element of a list or `None` if there is no such element. The specification clarifies that to create a new sequence at location s , the caller needs to own the location s and provide a pointer e to be stored inside s . The function `push` likewise requires the caller to have ownership of the location that is added to the sequence, and `pop` will return ownership of the popped location. This means that the operations do not perform allocation or deallocation of nodes, but are used to manage locations of the sequence.

We want to highlight the specification of `next`, as its postcondition, maybe unexpectedly, seems to return the sequence unchanged. In a non-empty cycle `Seq` $s \ (l :: L) e$ the call `next` s returns the location l . Intuitively, the caller would now continue to operate on the sequence `Seq` $l \ L e$. Formally, this is done using the following “splitting” property:

$$\text{Seq } s_1 \ (L_1 ++ s_2 :: L_2) \dashv\vdash \text{Seq } s_1 \ L_1 \ s_2 * \text{Seq } s_2 \ L_2 \ e \quad (1)$$

Here, $\dashv\vdash$ expresses interderivability of separation logic assertions, making it possible to use the rule in both directions. This means that one can use the rule in left-to-right direction to obtain a `Seq` predicate for any position in the list, then `push` or `pop` elements there, and finally use the right-to-left direction to reattain the `Seq` predicate for the whole sequence.

2.3 Cyclic Lists

We can now use the previously defined sequences to define intrusive cyclic lists. For this, we simply make use of the fact that we can freely choose the pointer stored at the end of a sequence and use it to point back to the start:

```
Cycle.pred c L := Seq c L c
```

The `Cycle` predicate satisfies the following “rotation” property, derived from the “splitting” property for `Seq` (1), reflecting its cyclic structure:

$$\text{Cycle } c \ (c' :: L) \dashv\vdash \text{Cycle } c' \ (L ++ [c]) \quad (2)$$

The API for cyclic lists is similar to the one for sequences, but includes a function that checks if a list is empty. This is done by comparing the starting location of the cycle to the location it points to. If the two are equal, the cycle is considered empty. Once we attach data to the intrusive cycle, the starting location takes the special role of a *sentinel node*.

```

Cycle.new start := Seq.new start start
Cycle.is_empty start := !start = start
Cycle.insert := Seq.push
Cycle.remove := Seq.pop
Cycle.next := Seq.next
    
```

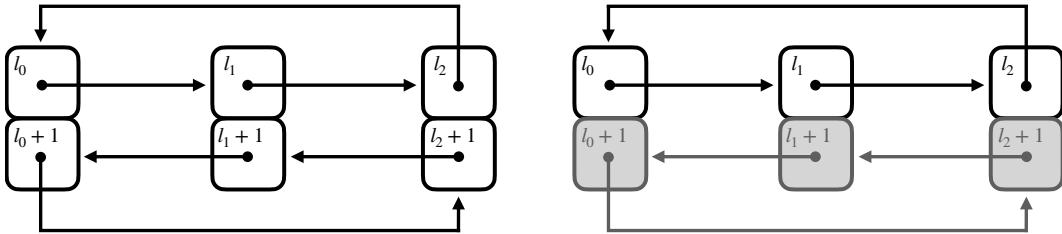
Many of the functions are directly defined in terms of the sequence functions. The specifications are as follows, where `is_nil` returns a boolean reflecting if a list is empty:

$\{c \mapsto _ \}$	<code>new c</code>	$\{ \text{Cycle } c [] \}$
$\{ \text{Cycle } c L \}$	<code>is_empty c</code>	$\{ b. b = \text{is_nil } L * \text{Cycle } c L \}$
$\{ \text{Cycle } c L * l \mapsto _ \}$	<code>insert c l</code>	$\{ \text{Cycle } c (l :: L) \}$
$\{ \text{Cycle } c (l :: L) \}$	<code>remove c</code>	$\{ p. p = l * \text{Cycle } c L * p \mapsto _ \}$
$\{ \text{Cycle } c L \}$	<code>next c</code>	$\{ p. p = \text{nth } 0 (L ++ [c]) * \text{Cycle } c L \}$

Similar to sequences, we can use the specification of `next` in combination with the “rotation” property (2) to insertion and remove elements at arbitrary positions in the cycle.

2.4 Doubly-linked cyclic Lists

We now come to the more interesting example of intrusive doubly-linked cyclic lists, or *dcycles* for brevity. Conceptually, a dcycle is composed of a cyclically arranged nodes, each node containing two pointers, one to the next node, and one to the previous node. Another way to split a dcycle into two parts, is to collect all the forward pointers in a cycle, and all the backward pointers in a separate cycle.



■ **Figure 1** Each node of the dcycle consists of two pointers, one pointing to the next, and one pointing to the previous node.

■ **Figure 2** The dcycle can be decomposed into two cycles, one containing all the `next` pointers (white), and one with all the `prev` pointers (grey).

Both combined make up the doubly-linked cyclic list. It is this latter view that motivates our choice for the representation predicate for dcycles, since it allows us to re-use the previous cycle specification in a straightforward way:

$$\text{DCycle.pred } c L := \text{Cycle } c L * \text{Cycle } (c + 1) (\text{rev_add_1 } L)$$

The function `rev_add_1` reverses the list of locations and adds 1 to every location, generating the cycle of backward pointers. By this definition, a node of the dcycle `DCycle.pred c L` at location $l \in L$ owns the resources $l \mapsto \text{next} * (l + 1) \mapsto \text{prev}$, where `next` points to the next node in the dcycle and is owned by the underlying cycle `Cycle c L`, and `prev` points to

the previous one, and is owned by the cycle `Cycle (c + 1) (rev_add_1 L)` (Figure 2). Basic operations on dcycles are implemented in a way that leverages the functions already provided by the underlying cycles. For readability, we introduce some notation for operations used in accessing the next and previous pointer of a node: given a location l in a dcycle, we write $l.\text{next}$ for $l + 0$ and $l.\text{prev}$ for $l + 1$.

```

DCycle.new c := Cycle.new c.next ; Cycle.new c.prev
DCycle.is_empty c := Cycle.is_empty c.next
DCycle.next c := Cycle.next c.next
DCycle.prev c := (Cycle.next c.prev) + (-1)
DCycle.insert_0 c new := let next = DCycle.next c in
                          Cycle.insert next.prev new.prev ;
                          Cycle.insert c.next new.next
DCycle.remove_0 c := let nn = DCycle.next (DCycle.next c) in
                    let l_0 = Cycle.remove c.next in
                    Cycle.remove nn.prev ; l_0

```

The function `insert_0` and `remove_0` are used to insertion and remove the node that comes after the current node. We can also define functions `insert_1` and `remove_1`, which do the same operations in the other direction of the dcycle. We omit these here, but they can be found in the Coq mechanization. The specifications of the dcycle functions are:

$\{ c \mapsto [_, _] \}$	<code>new c</code>	$\{ \text{DCycle } c \ [\] \}$
$\{ \text{DCycle } c \ L \}$	<code>is_empty c</code>	$\{ b. b = \text{is_nil } L * \text{DCycle } c \ L \}$
$\{ \text{DCycle } c \ L \}$	<code>next c</code>	$\{ p. p = \text{nth } 0 \ (L \ ++[c]) * \text{DCycle } c \ L \}$
$\{ \text{DCycle } c \ L \}$	<code>prev c</code>	$\{ p. p = \text{nth } 0 \ (\text{rev } L \ ++[c]) * \text{DCycle } c \ L \}$
$\{ \text{DCycle } c \ L * l \mapsto [_, _] \}$	<code>insert_0 c l</code>	$\{ \text{DCycle } c \ (l :: L) \}$
$\{ \text{DCycle } c \ (l :: L) \}$	<code>remove_0 c</code>	$\{ p. p = l * \text{DCycle } c \ L * p \mapsto [_, _] \}$

We often need to make use of a “rotation” property, which analogously to the similar property for cycles (2), allows us to cyclically rotate the locations of the dcycle:

$$\text{DCycle } c \ (c' :: L) \dashv\vdash \text{DCycle } c' \ (L \ ++[c]) \quad (3)$$

To illustrate in how far the chosen definition of the dcycle representation predicate allows for modular reasoning, let us consider what happens during the verification of `DCycle.remove_0`. Its specification as given above is:

$$\{ \text{DCycle } c \ (l :: L) \} \text{DCycle.remove}_0 \ c \ \{ p. p = l * \text{DCycle } c \ L * p \mapsto [_, _] \}$$

The function `DCycle.remove_0` first makes two calls to `DCycle.next`. The specification of `DCycle.next` keeps the initial `DCycle` predicate unaltered, so we make use of the rotation property. Next, there are two calls to the function `Cycle.remove`, each separately effecting one of the two underlying cycles of the dcycle. At this point, we would like to use the already verified specification of `Cycle.remove` from Section 2.3. Fortunately, we can achieve this by unfolding the definition of the `DCycle` predicate:

$$\begin{aligned} & \text{DCycle } c \ (l :: L) \\ & \dashv\vdash \text{Cycle } c \ (l :: L) * \text{Cycle } (c + 1) \ (\text{rev_add_1 } (l :: L)) \end{aligned}$$

This then allows us to reason on the separate cycles and make use of the `Cycle.remove` specification twice.

2.5 Deques

We now illustrate how the intrusively treated dcycles can be used to implement and verify a deque data structure. A deque represents a linearly arranged list of elements, supporting push and pop operations for the addition and removal of elements at both the front and the end of the list. The cyclic nature of dcycles makes it convenient to implement a deque, since we can use the first node as a sentinel. Using a dcycle as the underlying structure, defining the representation predicate is also rather simple: we associate the nodes of the dcycle with the data that is supposed to be stored next to it:

$$\text{Deque.pred } c D := \exists L. \text{DCycle } c L * \bigstar_{l,d \in L,D} (l - 1) \mapsto d$$

Since a node of the underlying dcycle at location l stores two pointers, the data is stored at location $l - 1$. From the entry-point c of the dcycle (which does not get any data), we can then make changes at the head and tail of the deque. Creation, push and pop operations for the deque are defined as follows:

```

Deque.new () := let l = alloc 2 None in DCycle.new l ; l
Deque.push_front x c := let l_x = alloc 3 x in DCycle.insert_0 c (l_x + 1)
Deque.push_back x c := let l_x = alloc 3 x in DCycle.insert_1 c (l_x + 1)
Deque.pop_front c := if DCycle.is_empty c then None else
  let next = DCycle.next c in
  let x = !next.data in
  let rem = DCycle.remove_0 c in
  free 3 (rem - 1) ; Some x
Deque.pop_back c := if DCycle.is_empty c then None else
  let prev = DCycle.prev c in
  let x = !prev.data in
  let rem = DCycle.remove_1 c in
  free 3 (rem - 1) ; Some x

```

Here, $l.\text{data}$ is notation for $l - 1$. Note that `Deque.new` only makes an allocation for an array of length 2, since it only creates the sentinel node of the underlying dcycle, which does not get to hold any data.

Thanks to the already verified specifications for the operations on dcycles, available lemmas about the big separating conjunction in the library of Iris, and the usage of the proof automation framework Diaframe [26], verifying the specifications of the deque API surmounts to less than 30 lines of proof in Coq.

3 Intrusive Binary Search Trees

In this section, we discuss and specify intrusive trees (Section 3.1), akin to those found in the Linux kernel [32, 22], where they are used for the implementation of the red-black trees. In the introduction (Section 1) we gave an example where the replacement of an element in a list was split into two parts: first, finding the right position in the intrusive list and returning a pointer to that location, and second, using this pointer to make the replacement in the corresponding data field. In the case of binary search trees, our implementation of the insertion operation will likewise be done in two steps. It first uses a function `locate` to find the correct position for the insertion – making use of the intrusive structure for navigation – and then carry out the insertion in this position. Again, it will be our goal to show that the verification of the final insertion operation can be done modularly (Section 3.3).

Since `locate` searches for – and potentially stops – at an arbitrary position inside a tree, we need to deal with partially traversed binary trees. In Section 3.3 we will show how we can deal with this by using the “magic wand as frame” approach [7], which we have also adapted (Section 3.2) to deal with the verification of properties of the functional `locate` function.

3.1 Representation Predicates

To illustrate a use of binary search trees, our overall goal will be to use them to implement a map data structure, which keeps track of key-value pairs by making use of an underlying binary search tree.

To start, we specify the intrusive tree structure, which relates a tree $t : \text{tree } \text{loc}$ labeled with locations – each one the location of a node of the heap representation of the tree – to the root-location $l : \text{loc}$ of the tree.

```
Tree.pred : loc → tree loc → iProp
Tree.pred l Leaf      := l ↦ None
Tree.pred l (Node p t1 t2) := l ↦ Some p * Tree.pred p t1 * Tree.pred (p + 1) t2
```

In the above, we make use of polymorphic typed trees in Coq, since we will use them with different types for the labels.

```
tree (A : Type) ::= Leaf : tree A | Node : A → tree A → tree A → tree A
```

We also define standard `map` and `inorder` functions on those trees. So far we have only specified the intrusive binary tree shape. To get binary search trees, our next step is to add a key of type K to every node in the tree, changing the signature to take trees of type `tree (K * loc)`, and to enforce the binary search tree invariant. We restrict our attention to binary search trees that use natural numbers as their keys (*i.e.* $K = \text{nat}$), and we rely on a Coq predicate `BST_inv_nat : tree nat → Prop` to describe binary search trees on the level of Coq. Combining the above, the desired heap predicate for binary search trees is:

```
BST.pred : loc → tree (K * loc) → iProp
BST.pred l t := Tree.pred l (π1 t) * BST_inv_nat (π2 t) * *(k,l) ∈ inorder t (l + 2) ↦ k
```

The predicate is a separating conjunction of three parts:

- The shape of the tree, which must hold for the tree of locations that we get from the first projection $\pi_1 t = \text{map fst } t$ of the tree $t : \text{tree } (K * \text{loc})$ of key-location pairs.
- The binary search tree invariant, which must hold for the tree of natural numbers that we get from the second projection $\pi_2 t = \text{map snd } t$ of the tree.
- A big separating conjunction over every key-location pair (k, l) in the tree t , and indicating where the key is stored. Notably, the structure of the tree plays no role here.

Using binary search trees, we can now specify finite maps $K \stackrel{\text{fin}}{\mapsto} \text{val}$ of key-value pairs. This is first done by specifying a finite map connecting keys and locations, which can then be used to attach data to the locations.

```
MapNode.pred : loc → (K  $\stackrel{\text{fin}}$  loc) → iProp
MapNode.pred l m := ∃ t. BST.pred l t * m = to_map t

Map.pred : loc → (K  $\stackrel{\text{fin}}$  val) → iProp
Map.pred l m := ∃ m'. MapNode.pred l m' * *(l,v) ∈ m',m (l - 1) ↦ v
```

19:12 Modular Verification of Intrusive List and Tree Data Structures in Separation Logic

The specification of the insertion operation for intrusive maps is then:

$$\begin{aligned} & \{ new \mapsto [None, None, k] * \text{MapNode } l \ m \} \\ & \quad \text{MapNode.insert } l \ new \\ & \{ ml. \text{MapNode } l \ (\langle k := new \rangle m) \\ & \quad * \text{ if } m(k) = \text{Some } l' \text{ then } ml = \text{Some } l' * l' \mapsto [_, _, k] \text{ else } ml = \text{None} \} \end{aligned}$$

Here, $\langle k := new \rangle m$ is the map m extended with the key-value pair (k, new) (the value of k is overwritten if it already exists). Note that by definition of `Tree.pred` the location l is always the entry point of the tree and not subject to any changes the function `insert` makes.

To verify the above, we make immediate use of the specification of insertion for trees, which will be discussed in Section 3.3. After using it we are left with proof obligations about the mathematical trees, which can be resolved by previously established lemmas, and lemmas from the library. Finishing the example, we can then use the above to verify the insertion operation on maps that have values attached:

$$\{ \text{Map } l \ m \} \text{Map.insert } l \ k \ v \ \{ \text{Map } l \ (\langle k := v \rangle m) \}$$

In the next two subsections we cover the definition and verification of the locate and insertion functions on binary search trees, first as functional versions on the level of the meta-theory (Section 3.2), and then implemented in the imperative object language (Section 3.3).

3.2 Functional Implementation of Tree Functions

To specify representation predicates for binary search trees we made use of polymorphic trees on the level of Coq. To give specifications for the `locate` and `insert` operations, we also need to implement functional versions of them. Our choice of implementing `insert` by making use of `locate`, instead of the more standard recursive definition, will lead to an interesting challenge when it comes to verifying that `insert` preserves the binary search tree invariant. Dealing with this is the main technical aspect we would like to highlight in this section.

We again try to keep things polymorphic and assume an arbitrary type K of keys, and a boolean comparison function $p : K \rightarrow K \rightarrow \text{bool}$. Functional implementations of `locate` and `insert` are then given by the following Coq code:

```

1 Fixpoint locate (k : K) (Γ : tree K → tree K) (t' : tree K) : (tree K → tree K) * tree K :=
2   match t' with
3   | Leaf      ⇒ (Γ , Leaf)
4   | Node k' l r ⇒ if p k k' then locate k (λ h, Γ (Node k' h r)) l
5                 else if p k' k then locate k (λ h, Γ (Node k' l h)) r
6                 else (Γ , t')
7   end.
8
9 Definition insert (k : K) (t : tree K) : tree K :=
10  match locate k id t with
11  | (Γ' , Leaf)      ⇒ Γ' (Node k Leaf Leaf)
12  | (Γ' , Node _ l r) ⇒ Γ' (Node k l r)
13  end.

```

The argument $\Gamma : \text{tree } K \rightarrow \text{tree } K$ of `locate` is used as a form a functional zipper [11] or *context* – it keeps track of the tree that is left behind, as we traverse down in search of the key. We can also think of the function Γ as a partial tree with one hole, waiting for a tree as input in order to be completed to a full tree. Since not all functions correspond to partial

trees that result from traversing down a tree (e.g. $\lambda t, \text{Node } t \ t$), we define a predicate ctx to define properly formed contexts. The constructors capture the ways in which `locate` enlarges the context during a recursive call.

```

1 Inductive ctx : (tree K → tree K) → Prop :=
2 | ctx_id : ctx id
3 | ctx_ht k t Γ : ctx Γ → ctx (λ h, Γ (Node k h t))
4 | ctx_th k t Γ : ctx Γ → ctx (λ h, Γ (Node k t h)).

```

Making use of the comparing function p we define the binary search tree invariant `BST` for trees over K , and make the assumption that p is asymmetric and antisymmetric to ensure that the invariant has the expected properties.

```

1 Inductive BST_inv : tree K → Prop :=
2 | BST_Leaf : BST_inv Leaf
3 | BST_Node x l r : BST_inv l → (∀ y, y ∈ to_set l → p y x) →
4   BST_inv r → (∀ y, y ∈ to_set r → p x y) →
5   BST_inv (Node x l r).

```

The above is the invariant which we specialized to natural numbers (`BST_inv_nat`) in Section 3.1. Next we want to prove that `insert` preserves the `BST` invariant, since this is a property that will be needed in the verification of the imperative implementation of `insert`.

Compared to the recursive implementation of `insert`, the usage of `locate` complicates this verification. A specification for `locate` needs to faithfully capture that the function can potentially stop in the middle of a tree, leaving behind a partially traversed tree and the root of a tree that has the sought after key.

To formally capture partially traversed trees, we define the predicate `BST_ctx`.

```

1 Definition BST_ctx (Γ : tree K → tree K) C C' :=
2   forall t, BST_inv t → to_set t ⊆ C → BST_inv (Γ t) ∧ to_set (Γ t) ⊆ C'.
3
4 Lemma BST_ctx_locate_spec {k Γ t Γ' t'} C :
5   locate k Γ t = (Γ', t') →
6   BST_inv t ∧ BST_ctx Γ ({k} ∪ to_set t) ({k} ∪ C) →
7   BST_inv t' ∧ BST_ctx Γ' ({k} ∪ to_set t') ({k} ∪ C).

```

The assertion `BST_ctx Γ C C'` expresses that for every BST t with keys in the set C , passing it to Γ will yield another binary search tree Γt whose keys are a subset of C' .

The above specification of `locate` can now be shown by induction on the `BST` invariant of the input tree t and making sure that the induction hypothesis generalizes over Γ . It also makes use of some properties of `BST_ctx`, which establish base cases and compositionality, and readily follow from the definition:

- $C \subseteq C' \rightarrow \text{BST_ctx id } C C'$
- $(\forall y. y \in C \rightarrow p y k) \rightarrow \text{BST_inv } t \rightarrow \text{BST_ctx } (\lambda h, \text{Node } k h t) C (\{k\} \cup C \cup \text{to_set } t)$
- $(\forall y. y \in C \rightarrow p k y) \rightarrow \text{BST_inv } t \rightarrow \text{BST_ctx } (\lambda h, \text{Node } k t h) C (\{k\} \cup C \cup \text{to_set } t)$
- $\text{BST_ctx } \Gamma A B \rightarrow \text{BST_ctx } \Gamma' B C \rightarrow \text{BST_ctx } (\Gamma' \circ \Gamma) A C$

To prove that `insert` preserves the `BST` invariant, we only need the special case of the above lemma, where $\Gamma = \text{id}$ and $C = \text{to_set } t$. This gives us:

```

1 locate k id t = (Γ', t') → BST_inv t →
2 BST_inv t' ∧ BST_ctx Γ' ({k} ∪ to_set t) ({k} ∪ to_set t).

```

By case analysis on the tree t , we can then show the desired preservation property of `insert`:

```

1 Lemma BST_inv_insert k t : BST_inv t → BST_inv (insert k t).

```

3.3 Locate and Insertion on Binary Search Trees

We come to a minimal API for binary search tree, only including a function to create a new tree and one to insert a new element. We again introduce notations for accessing the fields of a node: $l.\text{left}$ for $l + 0$, $l.\text{right}$ for $l + 1$, and $l.\text{key}$ for $l + 2$.

```

BST.new l := l ← None
BST.locate pos k := match !pos with
  None   => pos
| Some l => let k' = !l.key in
           if k < k' then BST.locate l.left k
           else if k > k' then BST.locate l.right k
           else pos
end
BST.insert root new := let new_key = !new.key in
                       let pos = BST.locate root new_key in
                       match !pos with
                         None => pos ← new ; None
                       | Some l => new.left ← !l.left ;
                                   new.right ← !l.right ;
                                   pos ← new ; Some l
                       end

```

Running `locate` to find a key k in the tree t , will return a value with the location of the node in t that contains the key k , or `None` if no such node exists. Notably, if the key is found, this means we get a pointer to a node in the tree. The challenge is now to find a way to give a specification for `locate`, since it must somehow mention this pointer to an internal node of the tree. This is not yet possible with the tree predicate we have given so far. The specification of `locate` should assure that the function will always successfully run on a BST.

$$\{ \text{BST } l \ t \} \text{ locate } l \ k \ \{ l'. \Phi \}$$

We still need to determine the predicate for the postcondition Φ . On the one hand, it will need to express that the returned pointer l' is the root of some subtree, which can be done by using the `BST` predicate. But apart from this subtree, Φ also has to account for the remainder of the initial tree t . Instead of coming up with a new data structure in Coq to describe these partial trees, we deal with this by following the “magic wand as frame” approach [7], which makes use of the separating implication \ast and functions Γ to define partial trees.

```

part_BST : loc → (tree (nat * loc) → tree (nat * loc)) → loc → iProp
part_BST l Γ p := ∀ t'. (BST.pred p t' * BST_inv_nat (π1(Γ t'))) → BST.pred l (Γ t')

```

In the predicate `part_BST l Γ p`, the location l is the root of the partial tree, p is the location which is missing a subtree, and Γ can intuitively be viewed as the remaining partial tree. Formally, during the proof, we require Γ to correspond to proper contexts, so we use the `ctx` predicate to enforce this. We can now formulate the specification for `locate` as follows:

$$\text{ctx } \Gamma \rightarrow \text{tree.locate } k \ s \ \text{id } t = (\Gamma', t') \rightarrow \{ \text{BST } l \ t \} \text{ locate } l \ k \ \{ l'. \text{part_BST } l \ \Gamma' \ l' \ * \ \text{BST } l' \ t' \}$$

Here, `tree.locate` is the functional implementation of `locate` in Coq (Section 3.2). The verification of the above specification is done by induction on the tree t . Since `locate` is

running a loop, we will need a loop invariant. A first proof attempt quickly reveals that during a loop of `locate`, the precondition involving `BST` can usually not be restored, since the tree is only partially traversed. But we can generalize the precondition to fix this issue.

$$\text{ctx } \Gamma \rightarrow \text{tree.locate } k \ s \ \Gamma \ t = (\Gamma', t') \rightarrow \\ \{ \text{part_BST } l \ \Gamma \ p * \text{BST } p \ t \} \text{locate } p \ k \ \{ p'. \text{part_BST } l \ \Gamma' \ p' * \text{BST } p' \ t' \}$$

The above can then be proven by induction on the tree t while making sure to generalize over all other parameters. The specification of `locate` can then be used to verify the insertion function, only requiring a case distinction on the tree, and no further proof by induction.

$$\{ \text{new} \mapsto [\text{None}, \text{None}, k] * \text{BST } l \ t \} \text{insert } l \ \text{new} \{ \text{BST } l \ (\text{tree.insert } k \ \text{new } t) \}$$

The proof makes use of the fact that the functional implementation of insertion preserves the `BST` invariant of the tree, which was discussed in Section 3.2.

4 Mechanization

All of the above presented data structures, specification and related proofs have been fully mechanized in the Coq proof assistant, making use of the Iris framework for separation logic. Apart from some notational short hands, the definitions and theorem statements in the paper directly reflect their counterparts in Coq.

In the verification, we additionally make use of Diaframe [26], which is a proof automation framework for Iris. It employs a goal-directed proof search strategy which can be extended by the user. The object programming language we use and study with Iris is `HeapLang`, which is the standard language provided by Iris, and is used without further adjustments. Regarding Diaframe, we added a few lines of code to enhance some simple handling of pointer arithmetic. These latter lines can be found in the `Setup.v` file.

The proofs remain rather simple for the sequences and cyclic lists, but start to get slightly more involved once we layer the intrusive lists in the case of the `dcycles` (Section 2.4). The representation predicate needs to be unfolded and its constituents manipulated in very deliberate ways, by making use of the rotation property of the cycle predicate. The same can be said about the verification in the case of the binary search trees. Here, the main formalization overhead (`Util.v`) was in relation to the underlying mathematical trees, which turns out to be significantly larger than the corresponding one for lists. This was mainly due to the choice of implementing insertion via `locate`, which required different proof approaches compared to the recursive version, but allowed us to discuss the problem of internal pointers in tree structures.

5 Related Work

Intrusive Data Structures. As part of effort in verifying Google’s pKVM hypervisor for Android, Pulte et al. [29] verify the buddy allocator [1] used in the hypervisor. The corresponding C code makes use of intrusive lists (`list_head`), which are specified as part of the main invariant in the verification. The intrusive data structure is however not identified and specified as an independent structure. Lee et al. [21] consider intrusive data structures in the context of Rust, where they focus the issue of type-checking intrusive structures in ownership type-systems.

Linked List in Separation Logic. Linked lists are a standard data structure covered in any introduction to separation logic [3, 5, 30]. They also serve as a natural target to benchmark verification tools and verifications can therefore for example be found in among others Bedrock [9], Charge [4], VST [3, 6], Viper [27] and Verifast [31, 12]. In all of these cases, lists are specified in a non-intrusive way, similar to `is_int_list` in the introduction (Section 1).

Magic Wand as Frame. Cao et al. [7] utilize the magic wand to describe partial data structures, which avoids the introduction of another recursively defined predicate to specify these kinds of data “frames”. While we have adopted their approach for defining partial trees by usage of the magic wand, our definition still differs. They define a partial tree that only represents the partial tree shape. We however actually define a partial *binary search tree*, *i.e.* the values in our partial structure define are sorted according to the BST invariant.

Higher-Order Representation Predicates in Separation Logic. Charguéraud [8] showcases how higher-order representation predicates can be used to describe polymorphic mutable data structures. He covers a wide range of standard structures, which includes mutable lists, list segments, records, trees and arrays. Similarly to the example we have given in Section 1, he treats the example of a function to read the n -th cell of a mutable list, and uses a predicate designating a list segment to be able to formulate the specification. Likewise, he continues by treating trees and showing techniques of how to represent trees with holes, *i.e.* trees, where the ownership of possibly several subtrees has been detached. Charguéraud uses recursively defined predicates to describe the structures with holes, whereas we have made use of the “magic wand as frame” approach. Lists are treated non-intrusively, and he does not cover cyclic data structures or binary search trees.

6 Conclusion

In this paper, we have presented an approach to specify intrusive data structures by first separately specifying the underlying node structure before the addition of data. One key feature of intrusive data structures is that they can be combined: they can be used to keep track of data across multiple intrusive data structures. With our given approach, this can easily be captured, since we can combine specifications of intrusive data structures. We illustrated this by using two cyclic lists to track data, effectively giving us the implementation of a deque (Section 2.5).

Throughout the presented examples, we have chosen a modular approach when it came to specifying the list and tree data structures. This is particularly evident in the specifications of the cyclic and doubly-cyclic lists. But this was not only limited to specifications; whenever we implemented a new function on a data structure, we made sure to reuse operations provided by any underlying structure. As a consequence, verifying the specifications of the presented data structures also ended up decomposing in a modular way. Since our mechanization makes use of the Iris framework and Diaframe, most proofs get simplified to the point where the only proof obligations that were left were related to the logical representations of the data.

Looking ahead, it remains to be determined to which extent this modular approach can be applied to more complex graph-like structures. The Linux kernel makes use of a task structure [25], which contains multiple intrusive list node occurrences (`list_head`) and with the addition of other intrusive structures, and the buddy allocator in pKVM [1, 2] makes use of intrusive lists to keep track of free blocks. So there are natural examples of data structures in which many intrusive structures are embedded. Some scaling challenges will

probably appear when tying the mathematical structures – which outline the shape and carry information about the location of nodes – to the heap representations of the intrusive structures. In the dcycle example (Section 2.4) this happened by the usage of the `rev_add_1` function. Many similar functions, or a complex relation, will most likely be needed in order to specify a structure that involves several embedded intrusive data structures.

References

- 1 Android Open Source Project. Buddy allocator in pKVM, 2024. URL: https://github.com/torvalds/linux/blob/master/arch/arm64/kvm/hyp/nvhe/page_alloc.c.
- 2 Android Open Source Project. Source code of the `hyp_pool` structure used in the pKVM implementation of the buddy allocator, 2024. URL: <https://github.com/torvalds/linux/blob/bf3a69c6861ff4dc7892d895c87074af7bc1c400/arch/arm64/kvm/hyp/include/nvhe/gfp.h#L12>.
- 3 Andrew W. Appel. *Program Logics for Certified Compilers*. Cambridge University Press, 2014.
- 4 Jesper Bengtson, Jonas Braband Jensen, and Lars Birkedal. Charge! In *ITP*, pages 315–331, 2012. doi:10.1007/978-3-642-32347-8_21.
- 5 Lars Birkedal and Aleš Bizjak. Lecture notes on Iris: Higher-order concurrent separation logic, 2024. URL: <https://iris-project.org/tutorial-material.html>.
- 6 Qinxiang Cao, Lennart Beringer, Samuel Gruetter, Josiah Dodds, and Andrew W. Appel. VST-Floyd: A Separation Logic Tool to Verify Correctness of C Programs. *JAR*, 61(1):367–422, 2018. doi:10.1007/s10817-018-9457-5.
- 7 Qinxiang Cao, Shengyi Wang, Aquinas Hobor, and Andrew W Appel. Proof Pearl: Magic Wand as Frame, 2018.
- 8 Arthur Charguéraud. Higher-order representation predicates in separation logic. In *CPP*, pages 3–14, 2016. doi:10.1145/2854065.2854068.
- 9 Adam Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In *PLDI*, pages 234–245, 2011. doi:10.1145/1993498.1993526.
- 10 Marc Hermes. Coq Mechanization of “Modular Verification of Intrusive List and Tree Data Structures in Separation Logic”, 2024. doi:10.5281/zenodo.12575047.
- 11 Gérard P. Huet. The zipper. *JFP*, 7(5):549–554, 1997. doi:10.1017/S0956796897002864.
- 12 Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In *NASA Formal Methods*, pages 41–55, 2011. doi:10.1007/978-3-642-20398-5_4.
- 13 Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. Higher-order ghost state. In *ICFP*, pages 256–269, 2016. doi:10.1145/2951913.2951943.
- 14 Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *JFP*, 28:e20, 2018. doi:10.1017/S0956796818000151.
- 15 Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *POPL*, pages 637–650, 2015. doi:10.1145/2676726.2676980.
- 16 Kernel Newbies. How does the kernel implements linked lists?, 2017. URL: <https://kernelnewbies.org/FAQ/LinkedLists>.
- 17 Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser, Amin Timany, Arthur Charguéraud, and Derek Dreyer. MoSeL: A general, extensible modal framework for interactive proofs in separation logic. *PACMPL*, 2(ICFP):77:1–77:30, 2018. doi:10.1145/3236772.
- 18 Robbert Krebbers, Ralf Jung, Aleš Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. The Essence of Higher-Order Concurrent Separation Logic. In *ESOP*, pages 696–723, 2017. doi:10.1007/978-3-662-54434-1_26.

- 19 Robbert Krebbers, Amin Timany, and Lars Birkedal. Interactive proofs in higher-order concurrent separation logic. In *POPL*, pages 205–217, 2017. doi:10.1145/3009837.3009855.
- 20 Olaf Krzikalla and Ion Gaztanaga. Boost.intrusive, part of Boost C++ documentation, 2024. Chapter 17, version 1.84.0. URL: https://www.boost.org/doc/libs/1_84_0/doc/html/intrusive.html.
- 21 Keunhong Lee, Jeehoon Kang, Wonsup Yoon, Joongi Kim, and Sue Moon. Enveloping Implicit Assumptions of Intrusive Data Structures within Ownership Type System. In *PLoS*, pages 16–22, 2019. doi:10.1145/3365137.3365403.
- 22 Linux Kernel. Source code of red-black trees, 2021. URL: https://github.com/torvalds/linux/blob/v6.8/include/linux/rbtree_types.h#L5.
- 23 Linux Kernel. Source code of linked lists library, 2023. URL: <https://github.com/torvalds/linux/blob/v6.8/include/linux/list.h>.
- 24 Linux Kernel. Source code of container_of macro, 2023. URL: https://github.com/torvalds/linux/blob/v6.8/include/linux/container_of.h.
- 25 Linux Kernel. Source code of task_struct, 2024. URL: <https://github.com/torvalds/linux/blob/v6.8/include/linux/sched.h#L748>.
- 26 Ike Mulder, Robbert Krebbers, and Herman Geuvers. Diaframe: Automated verification of fine-grained concurrent programs in Iris. In *PLDI*, pages 809–824, 2022. doi:10.1145/3519939.3523432.
- 27 Peter Müller, Malte Schwerhoff, and Alexander J. Summers. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *VMCAI*, pages 41–62, 2016. doi:10.1007/978-3-662-49122-5_2.
- 28 Peter W. O’Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *CSL*, volume 2142, pages 1–19, 2001. doi:10.1007/3-540-44802-0_1.
- 29 Christopher Pulte, Dhruv C. Makwana, Thomas Sewell, Kayvan Memarian, Peter Sewell, and Neel Krishnaswami. CN: Verifying Systems C Code with Separation-Logic Refinement Types. *PACMPL*, 7(POPL):1:1–1:32, 2023. doi:10.1145/3571194.
- 30 John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74, 2002. doi:10.1109/LICS.2002.1029817.
- 31 Jan Smans, Bart Jacobs, and Frank Piessens. VeriFast for Java: A Tutorial. In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, volume 7850, pages 407–442. Springer, 2013. doi:10.1007/978-3-642-36946-9_14.
- 32 The Linux Kernel. What are red-black trees, and what are they for?, 2007. URL: <https://docs.kernel.org/core-api/rbtree.html>.

Formalizing the Algebraic Small Object Argument in UniMath

Dennis Hilhorst  

Department of Mathematics, Utrecht University, The Netherlands

Paige Randall North  

Department of Information and Computing Sciences, Department of Mathematics, Utrecht University, The Netherlands

Abstract

Quillen model category theory forms the cornerstone of modern homotopy theory, and thus the semantics of (and justification for the name of) *homotopy type theory / univalent foundations* (HoTT/UF). One of the main tools of Quillen model category theory is the *small object argument*. Indeed, the particular model categories that can interpret HoTT/UF are usually constructed using the small object argument.

In this article, we formalize the *algebraic* small object argument, a modern categorical version of the small object argument originally due to Garner, in the Coq **UniMath** library. This constitutes a first step in building up the tools required to formalize – in a system based on HoTT/UF – the semantics of HoTT/UF in particular model categories: for instance, Voevodsky’s original interpretation into simplicial sets.

More specifically, in this work, we rephrase and formalize Garner’s original formulation of the algebraic small object argument. We fill in details of Garner’s construction and redefine parts of the construction to be more direct and fit for formalization. We rephrase the theory in more modern language, using constructions like *displayed categories* and a modern, less strict notion of *monoidal categories*. We point out the interaction between the theory and the foundations, and motivate the use of the algebraic small object argument in lieu of Quillen’s original small object argument from a constructivist standpoint.

2012 ACM Subject Classification Theory of computation → Logic and verification

Keywords and phrases formalization of mathematics, univalent foundations, model category theory, algebraic small object argument, coq, unimath

Digital Object Identifier 10.4230/LIPIcs.ITP.2024.20

Supplementary Material *Software (Source code)*: <https://github.com/UniMath/UniMath/tree/6605a4a/UniMath/ModelCategories>

archived at `swh:1:rev:6605a4a22a7dcda5d27025b9c75ae2f905338b6f`

InteractiveResource (Interactive version of source code): <https://math.asoa.dennishilhorst.nl/>

Other (Contribution PR): <https://github.com/UniMath/UniMath/pull/1822>

Other (Optimization PR): <https://github.com/UniMath/UniMath/pull/1858>

Funding *Paige Randall North*: This material is based upon work supported by the Air Force Office of Scientific Research under award number FA9550-21-1-0334.

1 Introduction

The tools of model category theory underpin the semantics of homotopy type theory and univalent foundations (HoTT/UF) (starting with [7, 19]). This work is the first step in building up this toolkit within HoTT/UF, with the ultimate goal of formalizing and verifying the semantics of HoTT/UF within HoTT/UF – or more specifically, within the Coq library **UniMath**. We focus on formalizing one major tool in the envisioned kit: the algebraic small object argument. This is the main tool for constructing particular model categories, and it (or versions) is used to model HoTT/UF in particular categories [19, 5, 8, 11, 10, 6].



© Dennis Hilhorst and Paige Randall North;
licensed under Creative Commons License CC-BY 4.0

15th International Conference on Interactive Theorem Proving (ITP 2024).

Editors: Yves Bertot, Temur Kutsia, and Michael Norrish; Article No. 20; pp. 20:1–20:18

Leibniz International Proceedings in Informatics



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

Model category theory and the algebraic small object argument

Model category theory, first introduced by Quillen [25], forms the foundation of modern homotopy theory. It provides a language and tools for this branch of mathematics, expanding the use of methods originally developed for the study of topological spaces to other domains: e.g. (higher) category theory [18, 17, 26], derived algebraic geometry [21], motivic homotopy theory [23], and now type theory as mentioned above.

A model category consists of two interacting *weak factorization systems* (WFSs) on a category, and, in specific examples, these are usually constructed via the *small object argument* [25] or variations. In this article, we study and formalize *algebraic/natural* weak factorization systems (NWFSs) [14] and the *algebraic* small object argument [13, 12], categorical improvements to the original versions that are well-suited for formalization. (Though *algebraic weak factorization system* is the more modern terminology, we will adhere to the vocabulary from [13] and refer to them as *natural weak factorization systems*.)

WFSs, while important for the theory of model categories, are lacking from a categorical or constructivist viewpoint. The definition and the problems created will be explicated in Section 3, but in summary a WFS consists of some structure A together with the property that some other structure B merely exists. NWFSs solve the constructivist problems that this creates by including both A and B explicitly, as structure, not property. In addition, to take advantage of machinery from category theory, B is not only given explicitly as structure, but as arising from (co)algebras of a (co)monad. Though the definition of NWFS is, *a priori*, more restrictive than that of WFS, most WFSs of interest to us (e.g. the ones in [19, 5, 8, 11, 10, 6]) are actually NWFSs. Additionally, NWFSs are an important tool in recent efforts towards producing constructive models of HoTT/UF [11, 10].

Quillen’s *small object argument* (SOA) [25] generates a WFS from a sufficiently well-behaved subclass of maps of a category. Garner introduced a variant, the *algebraic small object argument* (ASOA) [13] which produces NWFSs and which takes advantage of the fact that NWFSs are defined in terms of (co)monads in order to produce a cleaner construction. The NWFSs of [19, 5, 8, 11, 10, 6] are generated by the ASOA or variations.

Coq and UniMath

Coq in conjunction with the `UniMath` library (henceforth just `UniMath`) is a formalization framework for HoTT/UF. It adds insights from homotopy theory to type theory to produce a foundation of mathematics that is well-suited for the formalization of mathematics closely related to homotopy theory, especially the category theory that we are using here.

We only make light use of the additional assumptions that `UniMath` adds to Coq: we use functional extensionality and the homotopy levels of propositions and sets, as well as propositional truncation for the classical theory in Section 3 (though *not* in the actual ASOA construction in Section 5), but nothing else (including univalence). We do however take significant advantage of the technology developed in `UniMath` for formalizing category theory. In particular, we use the machinery of *displayed categories* [2] and the extensive formalization of *monoidal categories* [28]. Furthermore, we do expect the univalence axiom to become important and useful for the further development of model category theory in `UniMath`.

That being said, some of the high level machinery employed in [13] is not available in the `UniMath` library. For example, much of the theory of (co)ends and the two-fold monoidal categories employed there are not yet available in `UniMath`. Instead, we make more direct arguments, making for a more concrete and detailed description of the construction.

Contributions and related work

We formalize [13, Proposition 4.22] with the “smallness requirement” (in the language of [13]) that every object is finitely presentable (see Theorem 50), and its prerequisites. Garner also proves Proposition 4.22 for other smallness requirements. What we have formalized constitutes the most important parts of the algebraic small object argument construction in the most general setting with the current available theory, and is still applicable to situations of interest. See Remark 39 for technical details.

We have modified the proofs to be feasible in `UniMath`. We use the machinery of displayed categories [2] to construct various categories used in [13]; this is necessary to talk about functors commuting strictly (i.e., up to definitional equality), as is done in [13]. Where [13] uses strict monoidal categories, we use the weaker monoidal categories of [28], to make our constructions more widely applicable. We give the construction of the free monoid (see Section 5.4) and the proofs of [13, Proposition 4.18] (Section 5.3.1) and [13, Propositions 4.19 and 4.22] (Section 5.5) more directly. Furthermore, the formalization provides a more detailed and streamlined account for the proof of [13, Proposition 4.17].

Our formalization has been accepted into the `UniMath` library in commit `6605a4a`.

This paper is based on a master’s thesis by the first-named author [15]. This provides an expanded account of our formalization with more details and diagrams.

Outline and preliminary remarks

We first introduce the reader to relevant aspects of `UniMath` in Section 2. We then introduce WFSs and the SOA in Section 3, pointing out constructive issues. We introduce NWFSS in Section 4, showing that they fix the issues encountered in WFSs. Finally, we go over the algebraic small object argument in Section 5, going into some detail on our modifications.

Composition of morphisms in this paper is written in *diagrammatic order*, adopting the conventions in the `UniMath` library (so the composite of $f : X \rightarrow Y$ and $g : Y \rightarrow Z$ is written $f \cdot g$). Throughout, we assume \mathcal{C} to be a cocomplete category.

2 Preliminary theory in HoTT/UF

2.1 Homotopy levels

There is a hierarchy of *homotopy n -types* in `UniMath`, indexed by $n : \mathbb{N}$. We use 1-types, called *mere propositions*, and 2-types, called *sets*. A type P is a *mere proposition* if any two points are equal (meaning it may be empty): i.e., if there is a term of $\prod_{x,y:P}(x =_P y)$. We denote the type of mere propositions by `hProp` [27, `hProp`]. A type is a *set* if all of its identity types are propositions. We denote the type of sets by `hSet` [27, `hSet`].

Sometimes, instead of needing a term of a type A , it is sufficient (or perhaps even necessary!) to only know of the *mere existence* of a term of A . That is to say, we want a propositional type witnessing only that “a term of type A exists”, ignoring what this term is exactly or how it is constructed. This idea is captured in the *propositional truncation* of A , denoted $\|A\|$. `UniMath` defines $\|A\|$ through an “impredicative encoding” [27, `ishinh_UU`]

$$\|A\| := \prod_{P:\mathbf{hProp}} ((A \rightarrow P) \rightarrow P).$$

► **Remark 1.** It is important to note that we can *not* (in general) obtain a term of type A given a term of type $\|A\|$. We are effectively *losing* information when truncating a type. This is the root cause for the constructive issues arising in the classical theory of WFSs.

2.2 Category theory

In set-based mathematics, a category consists of a *set* O of objects, and for each $X, Y \in O$ a *set* of morphisms $\text{hom}(X, Y)$. In HoTT, one might naively define a category to consist of a *type* of objects and *types* of morphisms. We call this a *precategory*. Though allowing for basic constructions like limits [27, `Limits.v`] or colimits [27, `Colimits.v`], these are not sufficient for our purposes.

If we restrict the types $\text{hom}(X, Y)$ to be sets, we get the notion of a *category*. In the rest of this paper, we will only be considering categories.

We can add more restrictions to this notion of category to produce *univalent categories* [1] or *setcategories* [4]. Interestingly enough, we *never* need to assume either of these restrictions for our categories, which makes our construction applicable to both.

2.3 Displayed categories

It is common practice to construct a new category \mathcal{D} out of a category \mathcal{C} by adding data or properties to the objects and morphisms, often expressed in terms of a forgetful functor $F : \mathcal{D} \rightarrow \mathcal{C}$. Instead of mapping the objects and morphisms of \mathcal{D} to those of \mathcal{C} , it is useful to index them as families of objects and morphisms “lying over” those of \mathcal{C} . This idea is captured in the notion of *displayed categories*, analogous to dependent type families in HoTT [2].

- **Definition 2** ([27, `disp_cat`]). A displayed category \mathcal{D} over \mathcal{C} consists of the following:
- (i) For each object $X : \mathcal{C}$, a type \mathcal{D}_X of “objects over X ”;
 - (ii) For each morphism $f : X \rightarrow Y$ with $X, Y : \mathcal{C}$, and for each displayed object $\bar{X} : \mathcal{D}_X$ and $\bar{Y} : \mathcal{D}_Y$ a set of “morphisms from \bar{X} to \bar{Y} over f ”, denoted by $\bar{X} \rightarrow_f \bar{Y}$;
 - (iii) For each object $X : \mathcal{C}$ and each $\bar{X} : \mathcal{D}_X$, a displayed identity morphism $1_{\bar{X}} : \bar{X} \rightarrow_{\text{id}_X} \bar{X}$;
 - (iv) For all $X, Y, Z : \mathcal{C}$, $\bar{X} : \mathcal{D}_X$, $\bar{Y} : \mathcal{D}_Y$, $\bar{Z} : \mathcal{D}_Z$ and $f : X \rightarrow Y$, $g : Y \rightarrow Z$, a composition

$$(-) \cdot (-) : (\bar{X} \rightarrow_f \bar{Y}) \times (\bar{Y} \rightarrow_g \bar{Z}) \rightarrow (\bar{X} \rightarrow_{f \cdot g} \bar{Z});$$

satisfying displayed versions of associativity and identity axioms.

One can easily construct an “actual” category from a displayed category, analogous to the construction of dependent pair types from type families in HoTT. UniMath calls the constructed category \mathcal{D}^{tot} : the *total category of \mathcal{D}* . It provides *definitional* information on the relation of \mathcal{D}^{tot} to \mathcal{C} , whereas a forgetful functor would provide *propositional* information.

In the rest of this section, \mathcal{D} will always denote a displayed category over \mathcal{C} .

- **Definition 3** ([27, `total_category`]). The total category \mathcal{D}^{tot} of \mathcal{D} is defined to have
- **Objects:** $\sum_{X:\mathcal{C}} \mathcal{D}_X$
 - **Morphisms** from the dependent pairs (X, \bar{X}) to (Y, \bar{Y}) : the set $\sum_{f:X \rightarrow Y} \bar{X} \rightarrow_f \bar{Y}$ with the natural unit and composition.

► **Remark 4.** For readability, we will mostly refer to total categories *without* their explicit notation. That is to say, we may use \mathcal{D} to denote \mathcal{D}^{tot} for a displayed category \mathcal{D} .

► **Example 5 (arrow).** The *arrow category* \mathcal{C}^2 of \mathcal{C} is the total category of a displayed category over $\mathcal{C} \times \mathcal{C}$ with

- **Displayed objects** over $(X, Y) : \mathcal{C} \times \mathcal{C}$ as the type of arrows $X \rightarrow Y$.
- **Displayed morphisms** between displayed objects $f : X \rightarrow Y$ and $g : A \rightarrow B$ over a morphism $(h, k) : (X, Y) \rightarrow (A, B)$: the proposition $f \cdot k =_{(X \rightarrow B)} h \cdot g$.

It is the functor category from the poset $\mathbf{2} := \{0, 1\}$ to \mathcal{C} .

► **Example 6 (three).** The *three category* $\mathcal{C}^{\mathbf{3}}$ of \mathcal{C} is the total category of a displayed category over $\mathcal{C}^{\mathbf{2}}$ with

- **Displayed objects** over $f : X \rightarrow Y : \mathcal{C}^{\mathbf{2}}$:

$$\sum_{(E_f : \mathcal{C})} \sum_{(f_{01} : X \rightarrow E_f)} \sum_{(f_{12} : E_f \rightarrow Y)} f_{01} \cdot f_{12} =_{(X \rightarrow Y)} f.$$

- **Displayed morphisms** between displayed objects (E_f, f_{01}, f_{12}) and $(E_{f'}, f'_{01}, f'_{12})$ over a morphism (g_{00}, g_{22}) :

$$\sum_{g_{11} : E_f \rightarrow E_{f'}} \left(f_{01} \cdot g_{11} =_{(X \rightarrow E_{f'})} g_{00} \cdot f'_{01} \right) \times \left(f_{12} \cdot g_{22} =_{(E_f \rightarrow Y')} g_{11} \cdot f'_{12} \right).$$

It is the functor category from the poset $\mathbf{3} := \{0, 1, 2\}$ to \mathcal{C} .

► **Remark 7.** Though both of the previous examples are equivalent to certain functor categories from posets, defining them in terms of displayed categories provides definitional equalities that are much simpler to reason with in formalization. From a classical point of view, one may consider them to be functor categories.

UniMath defines displayed variants of functors and natural transformations, in such a way that they “lift” to functors and natural transformations on the total categories. We are mostly interested in one kind of displayed functor: *sections of displayed categories*. Given a category \mathcal{C} and a displayed category \mathcal{D} over \mathcal{C} there is a projection functor $\pi_1^{\mathcal{D}} : \mathcal{D}^{\text{tot}} \rightarrow \mathcal{C}$ projecting a pair (X, \bar{X}) down to X . Morally, a section is a *strict* right inverse to $\pi_1^{\mathcal{D}}$.

► **Definition 8** ([27, section_disp]). A section from \mathcal{C} to \mathcal{D} consists of a dependent function of objects $F : \prod_{X : \mathcal{C}} \mathcal{D}_X$ and a corresponding dependent function, also denoted F , of type $\prod_{f : X \rightarrow Y} F(X) \rightarrow_f F(Y)$, such that $F(\text{id}_X) =_{F(X) \rightarrow F(X)} \mathbf{1}_{F(X)}$ and $F(f \cdot g) =_{F(X) \rightarrow F(Z)} F(f) \cdot F(g)$ for morphisms $f : X \rightarrow Y$ and $g : Y \rightarrow Z$ in \mathcal{C} . Such a section lifts to a functor $\mathcal{C} \rightarrow \mathcal{D}^{\text{tot}}$.

► **Remark 9.** For any section $F : \mathcal{C} \rightarrow \mathcal{D}$ and any $X : \mathcal{C}$, the composite $(F \cdot \pi_1^{\mathcal{D}})(X)$ is in fact *definitionally equal* to X . There is no way to specify this definitional equality using a forgetful functor. The definitional equality is much more convenient to reason with, greatly simplifying the formalization process. Additionally, it is necessary to faithfully capture the classical theory that we are formalizing: see Remark 27.

We define a notion of natural transformation between sections, again to capture *strict* commutation over \mathcal{C} (one corresponding to whiskering a natural transformation with $\pi_1^{\mathcal{D}}$).

► **Definition 10** (section_nat_trans_disp). Let F, F' be sections of the displayed category \mathcal{D} over \mathcal{C} . A natural transformation of sections from F to F' is a family of displayed morphisms

$$\prod_{X : \mathcal{C}} F(X) \rightarrow_{\text{id}_X} F'(X),$$

making the appropriate diagrams commute.

3 Weak factorization systems

Weak factorization systems consist of two subclasses of morphisms (`morphism_class`) of a category \mathcal{C} , related through lifting properties, as well as a factorization of all morphisms. We define the lifting properties in terms of *lifting problems* and *fillers*. We use propositional truncation in both the lifting properties and the factorization.

► **Definition 11** (`filler`). For morphisms f, g in \mathcal{C} , an (f, g) -lifting problem is a morphism $f \rightarrow g$ in \mathcal{C}^2 . More precisely, it is a commutative square as in the left-hand diagram below.

$$\begin{array}{ccc} X & \longrightarrow & A \\ f \downarrow & & \downarrow g \\ Y & \longrightarrow & B \end{array} \rightsquigarrow \begin{array}{ccc} X & \longrightarrow & A \\ f \downarrow & \dashrightarrow & \downarrow g \\ Y & \longrightarrow & B \end{array}$$

We call a diagonal map $l : Y \rightarrow A$ that makes the whole diagram commute a filler.

► **Definition 12** (`lp`). Given morphisms f, g in \mathcal{C} , we say that (f, g) has the lifting property if there merely exists a filler for every (f, g) -lifting problem. That is,

$$\text{lp} := \prod_{f : X \rightarrow Y} \prod_{g : A \rightarrow B} \prod_{x : f \rightarrow g} \left\| \sum_{l : Y \rightarrow A} l \text{ is a filler for } x \right\|.$$

► **Remark 13**. We use propositional truncation here so that `lp` is a proposition, as we want to use it to define subclasses of morphisms: see Remark 17 below. Still, we are able to show interesting properties using the recursion principle of the propositional truncation (`WFS.v`).

► **Definition 14** (`rlp`, `llp`). We say that g has the right lifting property with respect to a subclass of morphisms \mathcal{L} if (f, g) has the lifting property for all $f \in \mathcal{L}$. We denote the class of all such g as \mathcal{L}^\square . Dually, we say that f has the left lifting property with respect to a class \mathcal{R} if (f, g) has the lifting property for all $g \in \mathcal{R}$. We denote the class of all such f by ${}^\square\mathcal{R}$.

► **Definition 15** (`wfs_fact_ax`). A pair of subclasses of morphisms $(\mathcal{L}, \mathcal{R})$ factors \mathcal{C} if for any $f : X \rightarrow Y$ there merely exists an object $E_f : \mathcal{C}$ and morphisms $\lambda_f : X \rightarrow E_f$ in \mathcal{L} and $\rho_f : E_f \rightarrow Y$ in \mathcal{R} such that $f = \lambda_f \cdot \rho_f$.

► **Definition 16** (`wfs`). A weak factorization system (WFS) is an ordered pair $(\mathcal{L}, \mathcal{R})$ of subclasses of morphisms in \mathcal{C} that factors \mathcal{C} and satisfies

$$\mathcal{L} = {}^\square\mathcal{R} \quad \text{and} \quad \mathcal{R} = \mathcal{L}^\square.$$

► **Remark 17**. The definition of a WFS shows why we need the propositional truncation in the lifting property: the equalities $\mathcal{L} = {}^\square\mathcal{R}$ and $\mathcal{R} = \mathcal{L}^\square$ would otherwise be ill-typed. Without the propositional truncation, corresponding notions of ${}^\square\mathcal{R}$ or \mathcal{L}^\square would not simply be subclasses of morphisms, but rather subclasses of morphisms *with extra data*, containing information about the fillers in any appropriate lifting problem.

In any WFS $(\mathcal{L}, \mathcal{R})$, \mathcal{L} is *left saturated* and \mathcal{R} is *right saturated* [22, Prop 14.1.8]. Left saturation tells us that \mathcal{L} contains all isomorphisms (`wfs_L_contains_isos`) and is closed under retracts (`wfs_L_retract`), pushouts (`wfs_closed_pushouts`), transfinite composition and coproducts (`wfs_closed_coproducts`). Right saturation is defined dually. Constructive issues arise in the last two closure properties. To illustrate, consider the following lemma.

► **Lemma 18** (`wfs_closed_coproducts`). Assume the axiom of choice. A WFS $(\mathcal{L}, \mathcal{R})$ is closed under coproducts. That is to say: for a set I and a family of maps $\{f_i : X_i \rightarrow Y_i\}_{i:I}$ such that $f_i \in \mathcal{L}$ for all $i : I$, the coproduct $f := \bigsqcup_{i:I} f_i$ is also in \mathcal{L} .

Proof. Consider a $g \in \mathcal{R}$ and an (f, g) -lifting problem. By the universal property of the coproduct, this is equivalent to a (f_i, g) -lifting problem for each $i : I$. We obtain the *mere existence* of fillers $l_i : Y_i \rightarrow A$ through the lifting properties of the f_i .

$$\begin{array}{ccc} \bigsqcup_{i:I} X_i & \longrightarrow & A \\ \bigsqcup_{i:I} f_i \downarrow & & \downarrow g \\ \bigsqcup_{i:I} Y_i & \longrightarrow & B \end{array} \rightsquigarrow \begin{array}{ccc} X_i & \longrightarrow & A \\ f_i \downarrow & \nearrow l_i & \downarrow g \\ Y_i & \longrightarrow & B \end{array}$$

Using the axiom of choice, we infer the *mere existence* of a morphism $\bigsqcup_{i:I} l_i : \bigsqcup_{i:I} Y_i \rightarrow A$ as a filler for the original lifting problem [24]. ◀

► **Remark 19 (Constructive issues).** Why do we need the axiom of choice in this proof? To put it shortly: because we lose information through the propositional truncation. We know of the *mere existence* of a lift in every individual diagram, but need to put all the lifts together to infer the mere existence of a lift in a “combined” diagram. We effectively want to construct a function

$$\left(\prod_{i:I} \left\| \sum_{l_i:Y_i \rightarrow A} f_i \cdot l_i = h \times l_i \cdot g = k \right\| \right) \rightarrow \left\| \sum_{l:\prod_{i:I} Y_i \rightarrow A} \prod_{i:I} f_i \cdot l(i) = h \times l(i) \cdot g \right\|.$$

This is precisely the statement of the axiom of choice in **UniMath**, which says that for any set X and any $L : X \rightarrow \mathbf{hSet}$, and any $P : \prod_{x:X} L(x) \rightarrow \mathbf{hProp}$, we have

$$\left(\prod_{x:X} \left\| \sum_{l_x:L(x)} P(x, l_x) \right\| \right) \rightarrow \left\| \sum_{l:\prod_{x:X} L(x)} \prod_{x:X} P(x, l(x)) \right\|.$$

Thus, we are only able to show that the left class of a WFS is closed under coproducts by assuming the axiom of choice.

As described in Remark 13 and Remark 17, we cannot drop the propositional truncation in the definition of a WFS to fix this issue. It is resolved in the theory of NWFSs however, where the added algebraic structure provides a canonical choice function in analogous lifting problems, see Remark 33 and Lemma 34.

3.1 The small object argument

In this section, we briefly describe the SOA [25], following the account in [16]. We have not formalized the SOA; this section is intended to motivate and build intuition for the ASOA.

The SOA allows us to construct WFSs given a sufficiently well-behaved subclass of morphisms. The constructed WFS is related to the generating class J through the lifting property itself: its right class will be J^\square , and its left class will be ${}^\square(J^\square)$.

For the rest of this paper, we assume J to be a subclass of morphisms in our category \mathcal{C} .

► **Definition 20.** A relative J -cell complex is a transfinite composition of pushouts of morphisms in J . We denote this class by J -cell.

► **Example 21.** The main motivating examples are in the categories of topological spaces (**TOP**) and simplicial sets (**SSET**). In both cases, J is the class of boundary inclusions (where $S^{-1} := \emptyset$)

$$J := \{ j_n : S^n \hookrightarrow D^{n+1} \mid n = -1, 0, 1, \dots \}.$$

We think of pushing out one map of J along a function $f : S^n \rightarrow X$ (the *attaching map*) as producing a relative cell complex $X \rightarrow X \sqcup_{S^n} D^{n+1}$: that is, the inclusion of X into X with

a copy of D^{n+1} (a *cell*) “glued” to it along f . Then we think of a relative J -cell complex as the inclusion of a space X into X with many cells attached. In topological spaces, these are called *relative CW-complexes*. When $X := \emptyset$ they are called *CW-complexes*.

► **Remark 22.** Assuming the axiom of choice, the class J -cell is a subclass of $\square(J^\square)$ since then $\square(J^\square)$ is closed under pushouts and transfinite compositions.

Let us briefly go over the SOA. We omit an explicit definition of the “smallness” giving rise to the name *small object argument*, but we will indicate when we use it.

► **Theorem 23** (Small object argument (SOA)). *Suppose the domains of all the maps in J are “small” relative to J -cell. Then there is a factorization $f \mapsto (\lambda_f, \rho_f)$ on \mathcal{C} such that, for all morphisms f in \mathcal{C} , λ_f is in J -cell and ρ_f is in J^\square .*

Proof sketch. Let $f : X \rightarrow Y$ be a morphism in \mathcal{C} . We inductively construct factorizations

$$X \xrightarrow{\lambda_f^\alpha} E_f^\alpha \xrightarrow{\rho_f^\alpha} Y$$

of f , for ordinals α . First, we set $\lambda_f^0 = \text{id}_X$ and $\rho_f^0 = f$. Since the composition of 0 morphisms is an instance of transfinite composition, id_X is in J -cell. However, f is not necessarily in J^\square ; we continuously “improve” this factorization in the inductive step until some ρ_f^i is in J^\square .

Consider the factorization $f \mapsto (\lambda_f^\alpha, \rho_f^\alpha)$ corresponding to step α . Let S^α be the set of all (g, ρ_f^α) -lifting problems with g ranging over J . For any lifting problem $x : S^\alpha$, denote by $g_x : A_x \rightarrow B_x$ the corresponding map in J . We define $E_f^{\alpha+1}$ and $\rho_f^{\alpha+1}$ through the pushout on the left-hand side of the following diagram. The right-hand side shows the first step in the transfinite sequence.

$$\begin{array}{ccc} \bigsqcup_{x \in S^\alpha} A_x \longrightarrow E_f^\alpha & \xrightarrow{\rho_f^\alpha} & Y \\ \bigsqcup_{x \in S^\alpha} g_x \downarrow & \lrcorner \downarrow s_\alpha & \rho_f^{\alpha+1} \dashrightarrow \\ \bigsqcup_{x \in S^\alpha} B_x \longrightarrow E_f^{\alpha+1} & \xrightarrow{\rho_f^{\alpha+1}} & Y \end{array} \quad \xrightarrow{\sim \alpha=0} \quad \begin{array}{ccc} \bigsqcup_{x \in S^0} A_x \longrightarrow X & \xrightarrow{f} & Y \\ \bigsqcup_{x \in S^0} g_x \downarrow & \lrcorner \downarrow \lambda_f^1 & \rho_f^1 \dashrightarrow \\ \bigsqcup_{x \in S^0} B_x \longrightarrow E_f^1 & \xrightarrow{\rho_f^1} & Y \end{array} \quad (1)$$

We set $\lambda_f^{\alpha+1} := \lambda_f^\alpha \cdot s_\alpha$, which is in J -cell by construction.

Note that the inductive process simply repeats the first step, meaning that

$$\rho_f^\alpha := \rho_{\rho_f^1 \dots \rho_f^1}^1 \quad (\alpha \text{ times})$$

This defines the (successor ordinal part of the) transfinite construction of the small object argument. One can show that the smallness of the domains in J means that there is some ρ_f^α which is in J^\square . Very roughly, the cells being attached to the domain of f at each step are solutions to lifting problems between J and f ; the smallness guarantees that at some point, solutions to all possible lifting problems have been added. ◀

► **Remark 24** (Constructive issues). Besides the use of the axiom of choice mentioned in Remark 22, we note some other constructive and categorical issues in the argument. In the categories of topological spaces and simplicial sets, the construction boils down to the following: at every step in the transfinite construction, we glue on cells for every possible lifting problem. This means that at every step, we glue duplicate cells, as we can glue all the cells that we have glued before (in addition to new ones). Thus, the construction never converges; we simply just *stop* whenever we have gone far enough, dictated by the smallness assumption on J . This again implies some sort of choice, and may introduce massive ordinal sequences (which pose a challenge in itself, see Remark 39). From a categorical perspective,

we might describe this issue as the fact that this construction has no universal property: how long you run the construction before stopping is not uniquely determined by the input morphism f . This is what was noticed and rectified in [13].

4 Natural weak factorization systems

Natural weak factorization systems (NWFSs) are an algebraic refinement of WFSs due to Grandis and Tholen [14]. An NWFS is based on a *functorial* factorization, which gives a canonical, well-behaved choice for the factorizations, as opposed to the structureless factorization in a WFS, see Definition 15. We impose additional algebraic structure, making it so that being an \mathcal{L} - or \mathcal{R} -map is no longer a *property* like for WFSs, but an algebraic *structure* on morphisms. This fixes the constructive issues in the closure properties of WFSs.

4.1 Functorial factorizations

Recall Example 6, defining \mathcal{C}^3 as a displayed category over \mathcal{C}^2 .

► **Definition 25** (`functorial_factorization`). *A functorial factorization F over a category \mathcal{C} is a section from \mathcal{C}^2 to \mathcal{C}^3 .*

► **Remark 26.** Compare this with the definition of factorization Definition 15. That was a section of the composition *function* $\text{ob}\mathcal{C}^3 \rightarrow \text{ob}\mathcal{C}^2$, as opposed to the projection *functor* $\mathcal{C}^3 \rightarrow \mathcal{C}^2$.

There are three natural functors $d_0, d_1, d_2 : \mathcal{C}^3 \rightarrow \mathcal{C}^2$ which take a composable pair $X \xrightarrow{\lambda_f} E_f \xrightarrow{\rho_f} Y$ to $\rho_f, \lambda_f \cdot \rho_f$, and λ_f , respectively (here d_1 coincides with the canonical projection). We obtain two endofunctors $\mathcal{C}^2 \rightarrow \mathcal{C}^2$ by considering $R := F \cdot d_0$ (which sends an f to its right map ρ_f) and $L := F \cdot d_2$ (which sends an f to its left map λ_f).

► **Remark 27** (The need for sections). With our definition, the left and right functors L and R are automatically compatible. That is to say, for any morphism $f : X \rightarrow Y$, *definitional equalities* arising from the definition of a section make for a well-typed (and trivial) equality $L(f) \cdot R(f) =_{X \rightarrow Y} f$.

A more naive approach would be to specify F as a section of d_1 in the usual sense:

$$\sum_{F: \mathcal{C}^2 \rightarrow \mathcal{C}^3} F \cdot d_1 = \text{id}_{\mathcal{C}^2} .$$

However, with this definition the composite $L(f) \cdot R(f)$ may not have the same domain and codomain as f . We merely know that their domains and codomains are *propositionally* equal, so the equality $L(f) \cdot R(f) = f$ is now *ill-typed*. One could use the `idtoiso` function [27], mapping equalities of objects to isomorphisms between them, but this does not capture the classical theory, which asks for $L(f) \cdot R(f)$ to be strictly equal, not just isomorphic, to f .

4.2 Natural weak factorization systems

Using only the data of a functorial factorization F , we can view the left and right functors L and R as a copointed endofunctor (L, Φ) and a pointed endofunctor (R, Λ) by defining:

$$\Phi_f := \begin{array}{ccc} X & \xlongequal{\quad} & X \\ \lambda_f \downarrow & & \downarrow f \\ E_f & \xrightarrow{\rho_f} & Y \end{array} \quad \text{and} \quad \Lambda_f := \begin{array}{ccc} X & \xrightarrow{\lambda_f} & E_f \\ f \downarrow & & \downarrow \rho_f \\ Y & \xlongequal{\quad} & Y \end{array} . \tag{2}$$

► **Definition 28** (`nwfs`). A natural weak factorization system (NWFS) is given by a functorial factorization F , together with an extension of (R, Λ) to a monad $\mathbf{R} = (R, \Lambda, \Pi)$ and the extension of the (L, Φ) to a comonad $\mathbf{L} = (L, \Phi, \Sigma)$. Such an NWFS is said to lie over F .

It will be useful to split this definition into two halves: LNWFS and RNWFS. The former contains only the data concerning the left comonad, the latter contains only the data concerning the right monad. We define the notion of LNWFS, an RNWFS is defined dually.

► **Definition 29** (`lnwfs_over`, cf. [13, 4.5]). The left half of an NWFS (LNWFS) is given by a functorial factorization F , with an extension of (L, Φ) to a comonad $\mathbf{L} = (L, \Phi, \Sigma)$.

4.3 Algebraic structure

Now we form a category \mathbf{Ff}_C of functorial factorizations on C (\mathbf{Ff}) by defining morphisms.

► **Definition 30** (`fact_mor`, cf. [13, 3.3]). A morphism of functorial factorizations $\tau : F \rightarrow F'$ is a natural transformation between sections.

Since we defined NWFSs as functorial factorizations “with added structure”, we define the category of NWFSs on C as a displayed category over \mathbf{Ff}_C . We again split this construction into LNWFSs and RNWFSs, yielding two displayed categories: \mathbf{LNWFS}_C and \mathbf{RNWFS}_C over \mathbf{Ff}_C . Together they form a displayed category \mathbf{NWFS}_C over \mathbf{Ff}_C .

In order to do this, we require some additional structure on the morphisms in \mathbf{Ff}_C . Take $F, F' : \mathbf{Ff}_C$. A morphism $\tau : F \rightarrow F'$ induces canonical natural transformations $\tau_L : L \Longrightarrow L'$ and $\tau_R : R \Longrightarrow R'$ by whiskering with d_2 and d_0 .

► **Definition 31** (`LNWFS`, `RNWFS`, `NWFS`, cf. [13, 3.3,4.5]). A morphism $\tau : F \rightarrow F'$ in \mathbf{Ff}_C is

- a morphism of LNWFSs if F and F' underlie LNWFSs and τ_L is a comonad morphism;
- a morphism of RNWFSs if F and F' underlie RNWFSs and τ_R is a monad morphism;
- a morphism of NWFSs if F and F' underlie NWFSs and τ is both a morphism of LNWFSs and of RNWFSs.

These three properties define the displayed morphisms of displayed categories \mathbf{LNWFS}_C , \mathbf{RNWFS}_C , and \mathbf{NWFS}_C over \mathbf{Ff}_C .

Similar to a WFS, an NWFS (L, R) has left- and right maps. These are defined to be the coalgebras of the comonad \mathbf{L} and the algebras of the monad \mathbf{R} respectively. We denote the categories of left and right maps of an (L, R) as $\mathbf{L-Map}$ and $\mathbf{R-Map}$ respectively.

4.4 Fixing the constructive issues

The algebraic structure in NWFSs allows us to construct fillers in lifting problems between any $\mathbf{L-Map}$ and $\mathbf{R-Map}$, fixing the constructive issues in the theory of WFSs.

► **Lemma 32** (`L_map_R_map_e1p`, cf. [13, 2.15]). Let (L, R) be an NWFS over F , $f : X \rightarrow Y$ an $\mathbf{L-Map}$, $g : A \rightarrow B$ an $\mathbf{R-Map}$. There exists a filler for any (f, g) -lifting problem (h, k) .

Proof. The (co)algebra axioms force the (co)algebra $\alpha_f : f \rightarrow \lambda_f$ and $\alpha_g : \rho_g \rightarrow g$ to be of specific forms. Specifically, they ensure that the morphisms $X \rightarrow X$ and $B \rightarrow B$ are in fact identities in the left-hand diagrams below. Consider then the right-hand diagram, obtained by applying F to (h, k) and attaching the (co)algebra morphisms. The filler $Y \rightarrow A$ can be read off the diagram as $s \cdot F(h, k)_{11} \cdot p$.

$$\begin{array}{ccc}
 \begin{array}{ccc}
 X & \xlongequal{\quad} & X \\
 f \downarrow & \alpha_f & \downarrow \lambda_f \\
 Y & \xrightarrow{s} & E_f \\
 E_g & \xrightarrow{p} & A \\
 \rho_g \downarrow & \alpha_g & \downarrow g \\
 B & \xlongequal{\quad} & B
 \end{array} & \rightsquigarrow &
 \begin{array}{ccccc}
 X & \xlongequal{\quad} & X & \xrightarrow{h} & A \\
 f \downarrow & \alpha_f & \lambda_f \downarrow & & \downarrow \lambda_g \\
 Y & \xrightarrow{s} & E_f & \xrightarrow{F(h,k)_{11}} & E_g & \xrightarrow{p} & A \\
 & & \rho_f \downarrow & & \downarrow \rho_g & \alpha_g & \downarrow g \\
 & & Y & \xrightarrow{k} & B & \xlongequal{\quad} & B
 \end{array}
 \end{array}$$

► Remark 33. Note that we construct the actual filler, and not just the *mere existence* of one. This is an important difference with WFSs, where we only know of the *mere existence* of a filler. We get a canonical choice function for the filler in any given lifting problem between an **L-Map** and an **R-Map**, fixing the problems we had with plain WFSs, see Remark 19. It allows us to prove analogues of the desired closure properties of WFSs, like the following.

► Lemma 34 (`nwfs_closed_coproducts`). *Let I be a set and $\{f_i : X_i \rightarrow Y_i\}_{i:I}$ a family of maps, such that f_i is an **L-Map** for every $i : I$. Then $\bigsqcup_{i:I} f_i$ is also an **L-Map**.*

5 The algebraic small object argument

The construction of the ASOA is inductive like its classical counterpart, the SOA. At each step, we construct an object of $\mathbf{LNWFS}_{\mathcal{C}}$. We then apply a general transfinite construction [20], giving us a full NWFS. There will be many similarities between the constructions, but also some obvious differences. The construction also resolves the constructive and categorical issues that we touched upon in Remark 22 and Remark 24.

We will be following Garner [13, 12], rephrasing the theory using univalent foundations and redefining part of the construction to be more direct and fit for formalization.

5.1 The one-step comonad

Let $f : X \rightarrow Y$ be a morphism in \mathcal{C} . Recall the first step from the iterative process in the small object argument in equation (1). This construction is in fact functorial, yielding a functorial factorization F^1 : the *one-step factorization* (`one_step_factorization`).

There is always a natural transformation $\Sigma^1 : L^1 \Longrightarrow L^1 \cdot L^1$, extending (L^1, Φ^1) to a comonad (where $L^1 := F^1 \cdot d_2$, the left part of F^1 , and Φ^1 is the copoint of L^1 given in 2), giving us an object of $\mathbf{LNWFS}_{\mathcal{C}}$: the *one-step comonad* (`one_step_comonad`), c.f. [12, Section 5.2]. The pointed endofunctor (R^1, Λ^1) does *not*, in general, extend to a monad, so we do not yet obtain an object of $\mathbf{NWFS}_{\mathcal{C}}$.

The first step in the algebraic small object argument corresponds with the first step in the classical counterpart. The “left part” of the initial factorization already satisfies the properties we need, while the “right part” of the first step has to be “fixed”.

5.2 Monoidal categories

Now we construct an NWFS from our one-step comonad L^1 . This uses Kelly’s *free monoid construction*. In [13], this takes place in a *strict monoidal category*. We instead use a more general notion of *weak monoidal categories*, formalized in `UniMath` in [28]. This is because in `UniMath` it is not possible to sensibly define strict monoidal categories, where associators and unitors are equalities on objects, unless one is working with setcategories (categories whose types of objects are sets). By using weak monoidal categories, our construction applies to more general categories (in particular, univalent categories).

Garner in fact uses *two-fold monoidal categories*, which comprise two interacting monoidal structures. This fits the theory perfectly, but is not yet formalized in `UniMath`, so we formalize only one monoidal structure. We only need one result relating to the other, which we prove directly (`LNWFS_comon_structure_whiskercommutes`). This simplifies our construction.

5.2.1 The monoidal structure on $\mathbf{LNWFS}_{\mathcal{C}}$

The idea behind the construction is to define a monoidal structure on $\mathbf{Ff}_{\mathcal{C}}$, such that a monoid corresponds with an object in $\mathbf{RNWFS}_{\mathcal{C}}$. This monoidal structure lifts to one on $\mathbf{LNWFS}_{\mathcal{C}}$, so that a monoid in $\mathbf{LNWFS}_{\mathcal{C}}$ corresponds with an object of $\mathbf{NWFS}_{\mathcal{C}}$. We define the unit of the monoidal structure on $\mathbf{Ff}_{\mathcal{C}}$ to be the initial object I mapping

$$X \xrightarrow{f} Y \quad \mapsto \quad X \xrightarrow{\text{id}_X} X \xrightarrow{f} Y. \quad (\mathbf{Ff_1comp_unit}, \text{ cf. [13, Theorem 4.14]})$$

For two functorial factorizations F, F' , we define their tensor product $F' \otimes F$ to be

$$X \xrightarrow{f} Y \quad \mapsto \quad X \xrightarrow{\lambda_f \cdot \lambda'_{\rho_f}} E'_{\rho_f} \xrightarrow{\rho'_{\rho_f}} Y. \quad (\mathbf{Ff_1comp}, \text{ cf. [13, Theorem 4.14]})$$

► **Lemma 35** (`Ff_monoidal`, `Ff_monoid_is_RNWFS`, cf. [13, Theorem 4.14]). *The pair (\otimes, I) defines a monoidal structure on $\mathbf{Ff}_{\mathcal{C}}$. A monoid structure on $F : \mathbf{Ff}_{\mathcal{C}}$ corresponds to an object of $\mathbf{RNWFS}_{\mathcal{C}}$ over F .*

Noting how \otimes acts on the right functor, the second claim boils down to the fact that a monad is a monoid in the category of endofunctors. Garner mentions the lifting of the monoidal structure to $\mathbf{LNWFS}_{\mathcal{C}}$ in the more general setting of two-fold monoidal categories [13, 4.11], but in the absence of this theory in `UniMath`, we take a direct approach. Proving this took about 1000 lines of formalization (`LNWFSMonoidalStructure.v`), and is the file that takes the longest to compile on various setups (see for example the discussion in PR 1858).

► **Remark 36.** The machinery used to lift the monoidal structure on $\mathbf{Ff}_{\mathcal{C}}$ to one on $\mathbf{LNWFS}_{\mathcal{C}}$ is that of *displayed monoidal categories* [3, `disp_monoidal`]. It allows one to define a monoidal structure on (the total category of) a displayed category over some monoidal category, by defining displayed analogues of the monoidal data in the base category.

► **Lemma 37** (`LNWFS_tot_monoidal`, `LNWFS_tot_monoid_is_NWFS`, cf. [13]). *Let $L, L' : \mathbf{LNWFS}_{\mathcal{C}}$ over $F, F' : \mathbf{Ff}_{\mathcal{C}}$ respectively. Then there is an \mathbf{LNWFS} structure on $F \otimes F'$. There is also an \mathbf{LNWFS} structure on I , lifting (\otimes, I) to a monoidal structure on $\mathbf{LNWFS}_{\mathcal{C}}$. Furthermore, a monoid $L : \mathbf{LNWFS}_{\mathcal{C}}$ over some $F : \mathbf{Ff}_{\mathcal{C}}$ corresponds with an object of $\mathbf{NWFS}_{\mathcal{C}}$ over F .*

► **Remark 38.** The classical small object argument boils down to a transfinite tensor product

$$L^\alpha := L^1 \otimes L^1 \otimes \dots \otimes L^1 : f \mapsto \lambda_f^\alpha.$$

This is not satisfactory, as it leaves us with the same issues discussed before, see Remark 24. We fix this by defining the iterative step with a coequalizer, associating duplicate cells (`next_pair_diagram_coeq`), and a simple convergence condition, removing the need for arbitrary truncation (`T_preserves_diagram_on`).

5.3 The iterative step

Garner generalized a transfinite construction by Kelly [20] to generate a monoid in a monoidal category \mathcal{V} , given certain “smallness requirements” on a generating object $T : \mathcal{V}$ and on \mathcal{V} itself. The construction defines a sequence indexed by the category of small ordinals [13, 4.16], converging at some limit ordinal.

► **Remark 39 (Limitations of ordinals in UniMath).** What limit ordinal Garner’s generalized sequence converges at [13, 4.16] is dictated by the hypothesis in [13, Proposition 4.19], requiring that “ $T \otimes (-)$ preserves λ -filtered colimits”, which is reduced to smallness requirement (*) in [13]. We limit ourselves to the first (finitely filtered) limit ordinal ω by replacing (*) with finite presentability, and substituting the hypothesis on T with **(V3)**.

We do this since the theory of (filtered) ordinals has not been developed enough in UniMath (or HoTT in general [9]). This is still sufficient to apply the theorem to important examples in, for instance, **SSET** and [5].

Formalizing requirement [13, (†)] or other ordinals should only involve adapting the proofs in (**GenericFreeMonoidSequence.v**), most notably up to the convergence of the sequence (**T_preserves_diagram_impl_convergence_on**), with appropriate hypotheses.

5.3.1 The transfinite sequence

For this section, we assume \mathcal{V} to be a monoidal category that has all connected colimits and T to be a pointed object in \mathcal{V} , with point $t : I \rightarrow T$. In the algebraic small object argument, \mathcal{V} will be **LNWFS_C** and T will be L^1 . It is easier and more performant to define this sequence on an abstract monoidal category in formalization, but it is useful to keep our main application in mind, particularly in the cases of **TOP** or **SSET**.

We assume the following “smallness requirements” on \mathcal{V} and T .

- (V1)** \mathcal{V} has ω -colimits and coequalizers.
- (V2)** \mathcal{V} is *right closed* (so the functor $(-) \otimes A$ preserves colimits for all $A : \mathcal{V}$).
- (V3)** The functor $T \otimes (-)$ preserves ω -colimits and coequalizers.

Given objects $X_0 := A, X_1 := T \otimes A$ in \mathcal{V} and $\sigma_0 := \text{id}_{T \otimes A} : T \otimes X_0 \rightarrow X_1$, we inductively define a transfinite sequence, called the *free T -algebra sequence for A* [13, 4.16]. For a successor ordinal $\alpha+ := \alpha + 1$ we define $X_{\alpha++}$ and $\sigma_{\alpha+} : T \otimes X_{\alpha+} \rightarrow X_{\alpha++}$ as the following coequalizer:

$$\begin{array}{ccccc}
 T \otimes X_\alpha & \xrightarrow{\sigma_\alpha} & X_{\alpha+} & \xrightarrow{t \otimes X_{\alpha+}} & T \otimes X_{\alpha+} & \xrightarrow{\sigma_{\alpha+}} & X_{\alpha++} \\
 & \searrow_{T \otimes (t \otimes X_\alpha)} & & \nearrow_{T \otimes \sigma_\alpha} & & & \\
 & & T \otimes (T \otimes X_\alpha) & & & &
 \end{array}$$

For any step α , we define $x_\alpha : X_\alpha \rightarrow X_{\alpha+}$ to be $(t \otimes X) \cdot \sigma_\alpha$. The full sequence becomes

$$\begin{array}{ccccccc}
 & & T \otimes X_0 & & T \otimes X_1 & & \dots & & T \otimes X_\alpha & & \\
 & & \nearrow_{t \otimes X_0} & \downarrow_{\sigma_0} & \nearrow_{t \otimes X_1} & \downarrow_{\sigma_1} & \nearrow_{t \otimes X_\alpha} & \downarrow_{\sigma_\alpha} & & & \\
 X_0 & \xrightarrow{x_0} & X_1 & \xrightarrow{x_1} & X_2 & \xrightarrow{x_2} & \dots & \longrightarrow & X_\alpha & \xrightarrow{x_\alpha} & X_{\alpha+} & \longrightarrow & \dots
 \end{array} \tag{3}$$

► **Remark 40.** The morphism $t \otimes X_\alpha$ is actually a morphism $I \otimes X_\alpha \rightarrow T \otimes X_\alpha$, so the diagram is actually ill-typed. By definition, there is a natural isomorphism to correct for this. Morphisms like this one are left out for simplicity, reading closer to the notion of *strict monoidal categories*, but they are accounted for in the formalization.

► **Remark 41.** In the examples in **TOP** and **SSET**, the functor $T \otimes (-)$ corresponds to “gluing cells”. Then X_α corresponds to “ α steps of gluing cells to A , *without duplicates*.” The coequalizers σ_α are continuous maps that identify duplicate cells with ones glued previously.

The sequence is defined inductively, using the previous *two* objects and the previous morphism to define the next morphism and object. In order to do this properly, with *definitional* equalities, we introduce a helper type, capturing the data of one “triangle” in the sequence.

► **Definition 42** (`pair_diagram`). A “pair diagram”, corresponding to the “triangle” of step α in the sequence displayed in equation (3), is an object of \mathcal{C}^3 of the form

$$X_\alpha \xrightarrow{t \otimes X_\alpha} T \otimes X_\alpha \xrightarrow{\sigma_\alpha} X_{\alpha+}.$$

► **Remark 43.** The only real data in this object are X_α , $X_{\alpha+}$ and $\sigma_\alpha : T \otimes X_\alpha \rightarrow X_{\alpha+}$.

Indeed, one can define the $(\alpha + 1)$ -th pair diagram using only the data from the α -th pair diagram. The inductive definition allows us to make sure left object of the $(\alpha + 1)$ -th pair diagram is in fact *definitionally equal* to the right object of the α -th pair diagram. Assumptions **(V1)** and **(V3)** ensure this sequence converges (`T_preserves_diagram_impl_convergence_on`), cf. [13, Proposition 4.17]. The limit when $A := I$ yields a T -algebra (T^∞, τ^∞) , consisting of an object $T^\infty : \mathcal{V}$ and a morphism $\tau^\infty : T \otimes T^\infty \rightarrow T^\infty$, which we will show is a monoid.

► **Remark 44.** Intuitively, keeping **TOP** or **SSET** in mind, we may view the object T^∞ as the “space with *all* cells attached”. The T -algebra map τ^∞ describes how one more step of attaching cells (through tensoring with T) can be collapsed back into T^∞ itself.

5.4 Obtaining the free monoid

In [12, Proposition 27], the forgetful functor from the category of T -algebras to \mathcal{V} is used to obtain a monoid. Instead, we define the monoid structure more directly, allowing for a much more direct and intuitive construction. There is an obvious choice for the unit $\eta^\infty : I \rightarrow T^\infty$: the canonical inclusion into the colimit $X_0 \hookrightarrow T^\infty$. It remains to find a multiplication. By assumption **(V2)**, we have

$$T^\infty \otimes T^\infty \cong \text{colim}(X_\alpha \otimes T^\infty).$$

We define the multiplication $\mu^\infty : T^\infty \otimes T^\infty \rightarrow T^\infty$ by defining a family of morphisms $\{\tau_\alpha : X_\alpha \otimes T^\infty \rightarrow T^\infty\}$ that forms a cocone on $\{X_\alpha \otimes T^\infty\}$.

► **Lemma 45** (`Tinf_pd_Tinf_map`). *There is a family of maps $\{\tau_\alpha : X_\alpha \otimes T^\infty \rightarrow T^\infty\}$ such that the following diagram commutes for any α .*

$$\begin{array}{ccc} T \otimes X_\alpha \otimes T^\infty & \xrightarrow{\sigma_\alpha \otimes T^\infty} & X_{\alpha+} \otimes T^\infty \\ T \otimes \tau_\alpha \downarrow & & \downarrow \tau_{\alpha+} \\ T \otimes T^\infty & \xrightarrow{\tau^\infty} & T^\infty \end{array}$$

► **Remark 46.** Intuitively, the τ_α “collapse α steps of gluing cells into T^∞ ”. The diagram tells us that it does not matter if we first collapse α steps of cells into T^∞ , and then the last step, or if we first collapse the last step into the first α steps, and then collapse that into T^∞ .

We define the τ_α inductively (`free_monoid_coeq_sequence_on_Tinf_pd_Tinf_map`), with obvious choices for τ_0 and τ_1 . In the inductive step, we use assumption **(V2)** to define $\tau_{\alpha++}$ as the unique map out of the coequalizer

$$\begin{array}{ccccc} T \otimes X_\alpha \otimes T^\infty & \xrightarrow{\sigma_\alpha \otimes T^\infty} & X_{\alpha+} \otimes T^\infty & \xrightarrow{t \otimes X_{\alpha+} \otimes T^\infty} & T \otimes X_{\alpha+} \otimes T^\infty & \xrightarrow{\sigma_{\alpha+} \otimes T^\infty} & X_{\alpha++} \otimes T^\infty \\ & & & & \downarrow T \otimes \tau_{\alpha+} & & \downarrow \exists! \tau_{\alpha++} \\ T \otimes t \otimes X_\alpha \otimes T^\infty & \xrightarrow{\quad} & T \otimes T \otimes X_\alpha \otimes T^\infty & \xrightarrow{\quad} & T \otimes \sigma_\alpha \otimes T^\infty & \xrightarrow{\quad} & T \otimes T^\infty & \xrightarrow{\tau^\infty} & T^\infty \end{array}$$

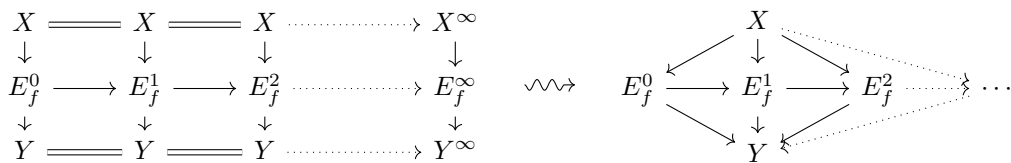
The required commutativity constraint on $\tau_{\alpha++}$ can be read off the diagram directly. The smallness assumptions allow one to show that $(T^\infty, \eta^\infty, \mu^\infty)$ is indeed a monoid in \mathcal{V} (`Tinf_monoid`).

5.5 Smallness requirements on $\mathbf{LNWFS}_{\mathcal{C}}$ and L^1

We used assumptions on \mathcal{V} and T that we now need to show for $\mathcal{V} = \mathbf{LNWFS}_{\mathcal{C}}$ and $T = L^1$. We show (V1) and (V2) and reduce (V3) to a much weaker requirement, only involving the subclass of morphisms J used to define L^1 . In this section, we do not have access in `UniMath` to some of the categorical machinery (e.g., the full theory of (co)ends) used in [13], so our arguments are more direct.

5.5.1 Cocompleteness of $\mathbf{LNWFS}_{\mathcal{C}}$

Issues commonly arise in the displayed nature of $\mathbf{LNWFS}_{\mathcal{C}}$ over $\mathbf{Ff}_{\mathcal{C}}$, or that of \mathcal{C}^3 over \mathcal{C}^2 , requiring definitional equalities where naive arguments only produce propositional ones. Consider the image of a morphism $f : X \rightarrow Y$ under an ω -chain of functorial factorizations $\{F_{\alpha}\}$, as well as its colimit, in the left-hand diagram below.



The colimit exists (`three_colims`), but its domain X^{∞} and codomain Y^{∞} need not be *definitionally equal* to X and Y respectively. We merely know they are isomorphic. We could correct the domain and codomain with these isomorphisms to define an object of \mathcal{C}^3 over f , and in turn a colimit $F_{\infty} : \mathbf{Ff}_{\mathcal{C}}$. However, this is quite cumbersome to work with as we want to define a comonad structure on the left functor of F_{∞} to define colimits in $\mathbf{LNWFS}_{\mathcal{C}}$.

Instead, recall what the actual data is in functorial factorizations and transformations between them: the middle objects in the image, and the morphisms between them. We “collapse” the (definitional) equalities in the left-hand diagram to form the right-hand diagram above. Let E_f^{∞} be the colimit of the E_f^{α} . We always get a map $E_f^{\infty} \rightarrow Y$, but a map $X \rightarrow E_f^{\infty}$ can only be defined when the colimit is non-empty and connected, namely as the canonical inclusion of $X \rightarrow E_f^{\alpha} \rightarrow E_f^{\infty}$ for an arbitrary α . Indeed, we have the following.

► **Lemma 47** (`ColimFfCocone`, `ColimLNWFSCocone`, cf. [13, Prop. 4.18]). *Both $\mathbf{Ff}_{\mathcal{C}}$ and $\mathbf{LNWFS}_{\mathcal{C}}$ have all connected, non-empty colimits, where colimits in $\mathbf{Ff}_{\mathcal{C}}$ are constructed as described above, and colimits in $\mathbf{LNWFS}_{\mathcal{C}}$ lie over those of the projected diagrams in $\mathbf{Ff}_{\mathcal{C}}$.*

5.5.2 Right closure of $\mathbf{LNWFS}_{\mathcal{C}}$

Here too, Garner uses a high level argument [13, Proposition 4.18], but we take a more direct approach. One can show that the functor $(-) \otimes A : \mathbf{Ff}_{\mathcal{C}} \rightarrow \mathbf{Ff}_{\mathcal{C}}$ preserves colimits for any $A : \mathbf{Ff}_{\mathcal{C}}$ quite easily. The following lemma then proves requirement (V2).

► **Lemma 48** (`Ff_iso_inv_LNWFS_mor`). *Let $L, L' : \mathbf{LNWFS}_{\mathcal{C}}$ over $F, F' : \mathbf{Ff}_{\mathcal{C}}$ respectively. Let $\tau : F \rightarrow F'$ be an isomorphism. Then τ^{-1} is a morphism of \mathbf{LNWFS} s whenever τ is.*

5.5.3 Reducing the smallness requirement on L^1

To reduce requirement (V3) to a simpler one, we mostly follow [12, Proposition 32]. The last part of this reduction has again been rephrased to be more direct and fit for formalization, using low level arguments (`OneStepMonadSmall.v`). In the end, the smallness requirement we are left with is phrased in terms of *presentable* objects in a category.

► **Definition 49** (`presentable`). *Let $X : \mathcal{C}$. Then X is called ω -presentable if and only if the covariant hom-set functor $\text{hom}(X, -) : \mathcal{C} \rightarrow \mathbf{SET}$ preserves ω -colimits.*

► **Theorem 50** (`small_object_argument`). *Let J be a subclass of morphisms in a cocomplete category \mathcal{C} , such that any $g \in J$ is ω -presentable in \mathcal{C}^2 . Then there exists an NWFS in \mathcal{C} .*

► **Remark 51.** Besides using arbitrary ordinals and not just ω , Garner describes another, more involved smallness requirement [13, (†), Proposition 4.22]. Still, the case where we work is sufficient for us. See also Remark 39.

6 Scaling

Besides the strategies we used to make our formalization mathematically feasible, we also used the following strategies to make our formalization computationally feasible.

- Proper use of abstraction: Ending a lemma with `Qed`, makes it *opaque*, preventing the proof checker from *unfolding* it in other proofs. This significantly speeds up the proof checker, and the formalization process as a whole. The `abstract` tactic allows one to construct opaque terms within a proof. This alone sped up the compile time for the (`FFMonoidalStructure.v`) file from 30 minutes to 30 seconds on one setup.
- Avoiding `rewrite`: Though useful, this tactic produces large and unwieldy proof terms, which take a long time to verify. Instead we often used the `etrans` and `apply` tactics.
- Removing `cbn`, `simpl`, `unfold` or other “unfolding” tactics from finished proofs: These unfolded terms take much longer to type check. Avoiding the `rewrite` tactic allows us to completely remove these tactics from finished proofs, as tactics like `etrans` and `apply` do not consider the precise syntactic form of a goal term, but only consider its value up to definitional equality.
- Sectioning and local opacity: Compile times were also reduced by using context variables and proper sectioning. Local opacity (through the `Opaque` vernacular, used in e.g. (`LNWFSClosed.v`)) provided the benefits of opaque proof terms when the precise definition of a certain construction was not needed in a file, without enforcing opacity globally.

7 Conclusion

We have rephrased and formalized Garner’s algebraic small object argument [13] using machinery more appropriate for formalization in `UniMath`, like displayed categories and weak monoidal categories.

Let us briefly go over some of the main differences in the argument by Garner and this work. First, we filled in many details which [13] left implicit. For example, the explicit construction of lifting of the monoidal structure on $\mathbf{Ff}_{\mathcal{C}}$ to $\mathbf{LNWFS}_{\mathcal{C}}$ was left out in [13], but took over 1000 lines of formalization and is the file that takes the longest to compile in the entire formalization.

Secondly, we introduced more modern language, in the form of displayed categories [2] and a weak notion of monoidal categories [28].

Thirdly, we left out a lot of complex theory that Garner uses. This is, again, partly due to the limited available results in `UniMath`, but it contributes to the accessibility of the proofs. Complex constructions like two-fold monoidal categories are left out, Garner’s construction of the free monoid is replaced with a more direct and intuitive one.

The formalization gave more insight into the details of the theory, pointing out constructive issues in the theory of WFSs and showing how few assumptions Garner’s algebraic small

object argument really needs. In the formalization of the construction, we never assumed any categories to be setcategories (as they are in any classical theory, including [13]) or univalent categories.

There is still some work that may be done in the formalization of Garner’s article, for example overcoming our limitations mentioned in Remark 39 and Remark 51. Other than that, there are further results beyond the main theorem of [13] that could be formalized, for instance [13, Proposition 5.4]. Some theory on this has already been formalized (`algebraically_free`), but once complete more examples could be worked out as well. Beyond that, our ultimate goal is to use this to formalize semantics of HoTT/UF within UniMath.

References

- 1 Benedikt Ahrens, Kapulkin Krzysztof, and Michael Shulman. Univalent categories and the rezk completion. *Mathematical Structures in Computer Science*, 25(5):1010–1039, 2015. doi:10.1017/S0960129514000486.
- 2 Benedikt Ahrens and Peter LeFanu Lumsdaine. Displayed categories. *Logical Methods in Computer Science*, Volume 15, Issue 1, May 2017. doi:10.23638/LMCS-15(1:20)2019.
- 3 Benedikt Ahrens, Ralph Matthes, and Kobe Wullaert. Formalizing monoidal categories and actions for syntax with binders, 2023. arXiv:2307.16270.
- 4 Benedikt Ahrens, Paige Randall North, Michael Shulman, and Dimitris Tsementzis. The Univalence Principle, 2022. arXiv:2102.06275.
- 5 Steve Awodey. A cubical model of homotopy type theory, 2016. arXiv:1607.06413.
- 6 Steve Awodey. Cartesian cubical model categories, 2023. arXiv:2305.00893.
- 7 Steve Awodey and Michael A. Warren. Homotopy theoretic models of identity types. *Math. Proc. Cambridge Philos. Soc.*, 146(1):45–55, 2009. doi:10.1017/S0305004108001783.
- 8 Evan Cavallo, Anders Mörtberg, and Andrew W Swan. Unifying Cubical Models of Univalent Type Theory. In Maribel Fernández and Anca Muscholl, editors, *28th EACSL Annual Conference on Computer Science Logic (CSL 2020)*, volume 152 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 14:1–14:17, Dagstuhl, Germany, 2020. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.CSL.2020.14.
- 9 Tom de Jong, Nicolai Kraus, Fredrik Nordvall Forsberg, and Chuangjie Xu. Set-theoretic and type-theoretic ordinals coincide. In *2023 38th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. IEEE, June 2023. doi:10.1109/lics56636.2023.10175762.
- 10 Nicola Gambino and Simon Henry. Towards a constructive simplicial model of Univalent Foundations. *Journal of the London Mathematical Society*, 105(2):1073–1109, 2022. doi:10.1112/jlms.12532.
- 11 Nicola Gambino and Marco Federico Larrea. Models of Martin-Löf type theory from algebraic weak factorisation systems. *The Journal of Symbolic Logic*, 88(1):242–289, June 2021. doi:10.1017/jsl.2021.39.
- 12 Richard Garner. Cofibrantly generated natural weak factorisation systems, 2007. arXiv:math/0702290.
- 13 Richard Garner. Understanding the small object argument. *Applied Categorical Structures*, 17(3):247–285, April 2008. doi:10.1007/s10485-008-9137-4.
- 14 Marco Grandis and Walter Tholen. Natural weak factorization systems. *Archivum Mathematicum*, 42, January 2006.
- 15 Dennis Hilhorst. A Formalization of the Algebraic Small Object Argument in UniMath, November 2023. Available at <https://studenttheses.uu.nl/handle/20.500.12932/45658>.
- 16 Mark Hovey. *Model Categories*. Mathematical surveys and monographs. American Mathematical Society, 2007.
- 17 André Joyal. Notes on quasi-categories. *Preprint*, 2008.

- 18 André Joyal and Myles Tierney. Strong stacks and classifying spaces. In Aurelio Carboni, Maria Cristina Pedicchio, and Guiseppe Rosolini, editors, *Category Theory*, pages 213–236, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg.
- 19 Chris Kapulkin and Peter LeFanu Lumsdaine. The Simplicial Model of Univalent Foundations (after Voevodsky), 2018. [arXiv:1211.2851](https://arxiv.org/abs/1211.2851).
- 20 G. M. Kelly. A unified treatment of transfinite constructions for free algebras, free monoids, colimits, associated sheaves, and so on. *Bulletin of the Australian Mathematical Society*, 22(1):1–83, 1980. doi:10.1017/S0004972700006353.
- 21 Jacob Lurie. *Derived Algebraic Geometry III: Commutative Algebra*, 2009. [arXiv:math/0703204](https://arxiv.org/abs/math/0703204).
- 22 J. P. May and K. Ponto. *More Concise Algebraic Topology: Localization, Completion, and Model Categories*. University of Chicago Press, Chicago, 2011. URL: <https://cds.cern.ch/record/1416976>.
- 23 Fabien Morel and Vladimir Voevodsky. A^1 -homotopy theory of schemes. *Publications Mathématiques de l’IHÉS*, 90:45–143, 1999.
- 24 nLab authors. weak factorization system. Available at <https://ncatlab.org/nlab/show/weak+factorization+system>, March 2024. Revision 46.
- 25 Daniel G. Quillen. *Homotopical Algebra*. Lecture notes in mathematics. Springer-Verlag, 1967.
- 26 Charles Rezk. A model for the homotopy theory of homotopy theory. *Transactions of the American Mathematical Society*, 353(3):973–1007, 2001.
- 27 Vladimir Voevodsky, Benedikt Ahrens, Daniel Grayson, et al. Unimath — a computer-checked library of univalent mathematics. Available at <http://unimath.org>. doi:10.5281/zenodo.7848572.
- 28 Kobe Wullaert, Ralph Matthes, and Benedikt Ahrens. Univalent Monoidal Categories. In Delia Kesner and Pierre-Marie Pédro, editors, *28th International Conference on Types for Proofs and Programs (TYPES 2022)*, volume 269 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 15:1–15:21, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.TYPES.2022.15.

A Formalization of the Lévy-Prokhorov Metric in Isabelle/HOL

Michikazu Hirata  

School of Computing, Tokyo Institute of Technology, Japan

Abstract

The Lévy-Prokhorov metric is a metric between finite measures on a metric space. The metric was introduced to analyze weak convergence of measures. We formalize the Lévy-Prokhorov metric and prove Prokhorov’s theorem in Isabelle/HOL. Prokhorov’s theorem provides a condition for the relative compactness of sets of finite measures and plays essential roles in proofs of the central limit theorem, Sanov’s theorem in large deviation theory, and the existence of optimal coupling in transportation theory. Our formalization includes important results in mathematics such as the Riesz representation theorem, which is a theorem in functional analysis and used to prove Prokhorov’s theorem. We also apply the Lévy-Prokhorov metric to show that the measurable space of finite measures on a standard Borel space is again a standard Borel space.

2012 ACM Subject Classification Mathematics of computing → Probability and statistics; Mathematics of computing → Mathematical analysis

Keywords and phrases formalization of mathematics, measure theory, metric spaces, topology, Lévy-Prokhorov metric, Prokhorov’s theorem, Isabelle/HOL

Digital Object Identifier 10.4230/LIPIcs.ITP.2024.21

Supplementary Material *Software (Formalization in the Archive of Formal Proofs)*: https://www.isa-afp.org/entries/Levy_Prokhorov_Metric.html [9]

Software (Formalization in the Archive of Formal Proofs): https://www.isa-afp.org/entries/Riesz_Representation.html [10]

Funding This work is supported by JSPS Research Fellowships for Young Scientists and JSPS KAKENHI Grant Number 23KJ0905.

Acknowledgements I would like to thank Yasuhiko Minamide and Tetsuya Sato for commenting on a draft of this paper. I also thank anonymous reviewers for their helpful comments and suggestions.

1 Introduction

The Lévy-Prokhorov metric is a mathematical tool to analyze asymptotic behaviors of distributions or measures in terms of weak convergence. Such analysis is one of the important aspects of probability theory and a foundation of statistics because the knowledge on asymptotic behaviors provides insights of what will be likely to happen when we collect large data.

Our motivation of formalizing the Lévy-Prokhorov metric is to prove that the measurable space of finite measures on a *standard Borel space* is again a standard Borel space, where a standard Borel space is a measurable space with certain good properties. Standard Borel spaces are often used in modern probability theory. The disintegration theorem, which guarantees the existence of *conditional probability kernels*, requires the underlying space to be a standard Borel space. Standard Borel spaces are also a theoretical basis for the theory of *quasi-Borel spaces*, a denotational model for higher-order probabilistic programs [8]. We formalize the Lévy-Prokhorov metric because we need to give a metric on finite measures in order to show that the space of finite measures is a standard Borel space. Another metric which metrizes weak convergence is the *Wasserstein metric*. The Wasserstein metric is applied



© Michikazu Hirata;

licensed under Creative Commons License CC-BY 4.0

15th International Conference on Interactive Theorem Proving (ITP 2024).

Editors: Yves Bertot, Temur Kutsia, and Michael Norrish; Article No. 21; pp. 21:1–21:18

Leibniz International Proceedings in Informatics



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

in transportation theory and machine learning. We chose to formalize the Lévy-Prokhorov metric because the Wasserstein metric may fail to be a metric in the mathematical sense when the underlying metric is not bounded.

During the proof of our goal, we also prove important mathematical theorems in related areas. Our work is divided into three parts.

Weak Convergence and the Lévy-Prokhorov Metric. We first formalize the notion of weak convergence including the Portmanteau theorem, equivalent conditions of weak convergence, and the topology of weak convergence. We define the notion of weak convergence using *filters* as convergence in Isabelle/HOL. We then formalize the Lévy-Prokhorov metric. We prove the equivalence of the topology of weak convergence and the topology induced by the Lévy-Prokhorov metric. The proof is different from the common textbook proofs (e.g. [3, 4]). We obtain a simpler proof thanks to the generalization of weak convergence by filters.

Prokhorov's Theorem. We show Prokhorov's theorem using the Lévy-Prokhorov metric. Prokhorov's theorem states that a set of (uniformly bounded) finite measures is relatively compact if and only if it is *tight*. Prokhorov's theorem plays essential roles in the proofs of the central limit theorem, Sanov's theorem, and the existence of the optimal coupling in transportation theory. In order to formalize Prokhorov's theorem, we also prove (a special case of) Alaoglu's theorem and the Riesz representation theorem. The Riesz representation theorem is an important result in functional analysis. While its proof, including related lemmas, consists of around nine pages in Rudin's book [22], our formalization takes more than 2,100 lines of proofs.

Measurable Spaces of Finite Measures. One often considers the measurable space of measures on some measurable space. Such spaces are used in stochastic processes and semantics of probabilistic programming. The measurable space of measures is defined independently from metrics or topologies. We prove that the measurable space of finite measures is generated from the Lévy-Prokhorov metric. As a consequence, we obtain that the measurable space of finite measures on a standard Borel space is a standard Borel space.

Our formalization is mainly based on the lecture notes by Gaans [27]. The lecture notes includes detailed proofs about the Lévy-Prokhorov metric on probability measures. We extend their definitions and proofs for finite measures.

Related Work

Avigad et al. formalized the notion of weak convergence of probability measures on \mathbb{R} and a special case of Prokhorov's theorem during the proof of the central limit theorem in Isabelle/HOL [1]. Compared to their work, our formalization of weak convergence treats finite measures on any metric spaces, and convergence is generalized by filters. While there is a simpler proof for the special case of Prokhorov's theorem that they formalized, Prokhorov's theorem that we formalize needs tools in functional analysis, such as the Riesz representation theorem, and thus requires more effort.

The Lean mathematical library, `mathlib` [25], includes ongoing formalization of the weak convergence and the Lévy-Prokhorov metric by Kytölä [15]. Their definition of the weak convergence is also generalized by filters and treats not only probability measures but also finite measures. They showed that the Lévy-Prokhorov metric on the set of finite measures on a pseudo metric space is a pseudo metric. They proved the equivalence of the topology of weak convergence and the topology induced by the Lévy-Prokhorov metric on the

space of probability measures. Our work contains more results than their work such as the equivalence of convergence with respect to the Lévy-Prokhorov metric and weak convergence (Theorem 10. 3) and Prokhorov's theorem (Theorem 20).

The Riesz representation theorem in its original form as given by Riesz is formalized by Narkawicz in PVS [19] and by Narita et al. in Mizar [18].

Paper Outline

In Section 2, we review the basic notions and theorems of topological spaces, metric spaces, and measurable spaces. In Section 3, we define the weak convergence of measures, the topology of weak convergence, and the Lévy-Prokhorov metric. We then show their properties. In Section 4, we explain Prokhorov's theorem and lemmas used in the proof of Prokhorov's theorem. In Section 5, we discuss the measurable space of finite measures.

We do not show Isabelle source code in this paper except for the definition of the topology of weak convergence and the Lévy-Prokhorov metric in Section 3.4. The definitions and statements in Isabelle/HOL are almost direct translations from the mathematical notation; therefore, printing them here would not provide any additional insights.

2 Preliminaries

In this section, we review basic definitions and theorems related to topology, metric spaces, and measure theory. Most of the results in this section are included in Isabelle/HOL's standard library.

2.1 Topology

Topology is a way of expressing *nearness* of points in a set. Let X be a set and \mathcal{O}_X a set of subsets of X . The pair (X, \mathcal{O}_X) is called a topological space when $\emptyset \in \mathcal{O}_X$, $X \in \mathcal{O}_X$, and \mathcal{O}_X is closed under finite intersections and arbitrary unions. We sometimes write only X for (X, \mathcal{O}_X) , when the structure is obvious from the context. We follow the standard definitions of topology, such as,

- $U \subseteq X$ is an open set of $X \stackrel{\text{def}}{\iff} U \in \mathcal{O}_X$
 - $C \subseteq X$ is a closed set of $X \stackrel{\text{def}}{\iff} X - C$ is open
 - $f : X \rightarrow Y$ is a continuous map $\stackrel{\text{def}}{\iff} \forall U \in \mathcal{O}_Y. f^{-1}(U) \in \mathcal{O}_X$
- for topological spaces X and Y .

2.2 Metric Spaces

While topological spaces express nearness in abstract way, metric spaces specify concrete distances. Let X be a set and $d : X \times X \rightarrow \mathbb{R}$. The pair (X, d) is called a *metric space* if the following holds.

- For all $x, y \in X$, $d(x, y) \geq 0$.
- For all $x, y \in X$, $d(x, y) = d(y, x)$.
- For all $x, y \in X$, $d(x, y) = 0 \iff x = y$.
- For all $x, y, z \in X$, $d(x, z) \leq d(x, y) + d(y, z)$.

We sometimes write only X for (X, d) . Let (X, d) be a metric space, $x \in X$ and $\varepsilon > 0$. The set $ball_X(x, \varepsilon) = \{y \in X \mid d(x, y) < \varepsilon\}$ is called an open ball with center x and radius ε . The set $cball_X(x, \varepsilon) = \{y \in X \mid d(x, y) \leq \varepsilon\}$ is called a closed ball with center x and radius ε . We assume that \mathbb{R} is equipped with the standard distance $d(x, y) = |x - y|$ in this presentation.

A metric space X induces the topological space (X, \mathcal{O}_d) , where \mathcal{O}_d consists of arbitrary unions of open balls. We call a topological space X *metrizable* if there exists a metric d on X , which induces X .

2.3 Filter and Convergence

In Isabelle/HOL's library, the notion of convergence is formalized in a general way using *filters*. A filter on I is a set of subsets of I satisfying certain conditions. Filters can, among other things, describe the set of all elements that are “sufficiently large” or “sufficiently close to a ”. We do not explain the detail of filters, which can be found in [13]. Let I be a set, F a filter on I , X a topological space, $\{x_i\}_{i \in I} \subseteq X$, and $x \in X$. The notion “ $\{x_i\}_{i \in I}$ converges to x in X with respect to F ”, denoted by $(x_i \longrightarrow x) F$ in X (*limitin* in Isabelle/HOL), is defined by

$$(x_i \longrightarrow x) F \text{ in } X \iff \text{For every open neighborhood } U \text{ of } x, \text{ eventually } x_i \in U \text{ w.r.t. } F.$$

Intuitively, $(x_i \longrightarrow x) F$ in X means that x_i is eventually close to x in X . We call x the limit if $(x_i \longrightarrow x) F$ in X . When the topology is obvious from the context, we omit the topology and write $(x_i \longrightarrow x) F$ for $(x_i \longrightarrow x) F$ in X . For instance, there are filters F_{seq} on \mathbb{N} and (at a) on \mathbb{R} corresponding to “for sufficiently large n ” and “for x sufficiently close to a ,” respectively. Convergences with respect to these filters have the same meaning as the usual definitions.

$$\begin{aligned} \lim_{n \rightarrow \infty} x_n = x &\iff (x_n \longrightarrow x) F_{\text{seq}} \text{ in } \mathbb{R} \\ &\iff \forall \varepsilon > 0. \exists N. \forall n \geq N. |x_n - x| < \varepsilon \\ \lim_{x \rightarrow a} f(x) = L &\iff (f \longrightarrow L) (\text{at } a) \text{ in } \mathbb{R} \\ &\iff (\forall \varepsilon > 0. \exists \delta > 0. \forall x. x \neq a \wedge |x - a| < \delta \implies |f(x) - L| < \varepsilon) \end{aligned}$$

In addition to limit, limit inferior and limit superior are also generalized by filter in Isabelle/HOL. Limit inferior and limit superior with respect to F are denoted by Liminf_F and Limsup_F , respectively.

A Characterization of Closed Sets by Limits. There is a characterization of closed sets using convergence with respect to *nets* (Exercise A.48 [7]). We formalize the following characterization of closed sets by limit with respect to filters because nets and filters are equally expressive in terms of convergence (Section 4 [23]).

- **Lemma 1.** *Let X be a topological space and $C \subseteq X$. Then, the following are equivalent.*
1. C is closed in X .
 2. For all sets I , filters F on I , $\{x_i\}_{i \in I} \subseteq C$, $x \in X$ such that $\emptyset \notin F$ and $(x_i \longrightarrow x) F$ in X , we have $x \in C$.

If X is first-countable, then these are also equivalent to the following.

3. For all $\{x_n\}_{n \in \mathbb{N}} \subseteq C$, $x \in X$ such that $(x_i \longrightarrow x) F_{\text{seq}}$ in X , we have $x \in C$.

The implication that 1 implies 2 (and 3) is already included in Isabelle/HOL's library. We prove the other implications. The last condition of the above equivalence has already been formalized for metric spaces. Since metric spaces are first-countable, our result is a relaxed version of the existing result. There is also a characterization of open sets by limit with respect to filters. The characterization is easily derived from that of closed sets.

From the characterization of closed sets, we obtain a condition to decide whether two topological spaces are equal using limit with respect to filters because topological spaces are determined by closed sets.

► **Corollary 2.** *Let (X, \mathcal{O}_X) and (X, \mathcal{O}'_X) be topological spaces.*

- *If $(x_i \rightarrow x) F$ in $(X, \mathcal{O}_X) \iff (x_i \rightarrow x) F$ in (X, \mathcal{O}'_X) for all $I, F, \{x_i\}_{i \in I}$, and x , then $\mathcal{O}_X = \mathcal{O}'_X$.*
- *If $(x_n \rightarrow x) F_{\text{seq}}$ in $(X, \mathcal{O}_X) \iff (x_n \rightarrow x) F_{\text{seq}}$ in (X, \mathcal{O}'_X) for all $\{x_n\}_{n \in \mathbb{N}}$ and x , and both of (X, \mathcal{O}_X) and (X, \mathcal{O}'_X) are first-countable, then $\mathcal{O}_X = \mathcal{O}'_X$.*

► **Remark 3.** In Isabelle/HOL, we cannot quantify filters as “for any filter F ” due to Isabelle/HOL’s type system. For instance, when showing $P \iff (\forall F :: \square \text{ filter. } Q F)$, we need to specify some type \square on which filters are defined. We state Lemma 1 and Corollary 2 by quantifying¹ filters as the type $F :: 'a \text{ set filter}$ when the topology is $X :: 'a \text{ topology}$ because we use a filter on $V(x)$ (the set of all open neighbourhoods of x) to prove the lemmas. Details of the filter are found in the lecture notes by Heil [7].

Finally, we define the Cauchy sequence and related notions.

- **Definition 4.** ■ *A sequence $\{x_n\}_{n \in \mathbb{N}}$ on a metric space X is called a Cauchy sequence if $\forall \varepsilon > 0. \exists N. \forall n, m \geq N. d(x_n, x_m) < \varepsilon$.*
- *A metric space is complete if every Cauchy sequence has a limit.*
- *A topological space X is called a completely metrizable space if there exists a complete metric on X , which induces X .*
- *A topological space X is called a Polish space if X is separable and completely metrizable.*

2.4 Measure Theory

The current measure theory library in Isabelle/HOL was first formalized by Hölzl and Heller [12] and has been extended by several other works [1, 5]. Let M be a set and Σ_M a set of subsets of M . A pair (M, Σ_M) is called a *measurable space* if Σ_M is non-empty and closed under complements and countable unions. We sometimes write M for a measurable space (M, Σ_M) . A member $A \in \Sigma_M$ is called a *measurable set*. A function f from a measurable space M to a measurable space N is *measurable* if $f^{-1}(A) \in \Sigma_M$ for all $A \in \Sigma_N$. Let M be a measurable space, $\mu : \Sigma_M \rightarrow [0, \infty]$ is a *measure* on M if $\mu(\emptyset) = 0$ and $\mu(\bigcup_{n \in \mathbb{N}} A_n) = \sum_{n=0}^{\infty} \mu(A_n)$ for any disjoint family $\{A_n\}_{n \in \mathbb{N}} \subseteq \Sigma_M$. A measure μ on M is called a *finite measure* if $\mu(M) < \infty$, a *sub-probability measure* if $\mu(M) \leq 1$, and a *probability measure* if $\mu(M) = 1$. For a measure μ on M and a measurable function $f : M \rightarrow \mathbb{R}$, $\int f d\mu$ denotes the Lebesgue integral of f with respect to μ .

A topological space (X, \mathcal{O}_X) induces the measurable space $(X, \sigma[\mathcal{O}_X])$, where $\sigma[\mathcal{O}_X]$ is the least σ -algebra including all open sets of X . The measurable space $(X, \sigma[\mathcal{O}_X])$ is called the Borel space. Notice that a metric space is also treated as a measurable space since it induces a topological space. The Borel space induced by a metric space (X, d) is denoted by (X, Σ_d) .

¹ The direction 1 implies 2 of Lemma 1, which is already included in Isabelle/HOL’s library, has been proved for all filters of any type using Isabelle’s polymorphism. That is, $P \implies Q (F :: 'b \text{ filter})$ can be stated in Isabelle.

3 The Lévy-Prokhorov Metric

Historically, Lévy first introduced a metric, known as the Lévy metric, between cumulative distribution functions [17]. Later, Prokhorov defined the Lévy-Prokhorov metric between finite measures analogous to the Lévy metric [21]. In this section, we review the notion of weak convergence and the Lévy-Prokhorov metric. At the end of this section, we discuss our formalization of the topology of weak convergence and the Lévy-Prokhorov metric. For a measurable space X , $\mathcal{P}(X)$ denotes the set of all finite measures on X . Note that X can be a metric space or a topological space since they both induce a measurable space.

3.1 Weak Convergence

The existing formalization of weak convergence in Isabelle/HOL's standard library is restricted to sequences on \mathbb{N} of probability measures on \mathbb{R} . We define the notion of weak convergence which treats finite measures on any topological spaces. The convergence in our formalization is generalized by filters.

► **Definition 5** (Weak Convergence). *Let X be a topological space, I a set, F a filter on I , $\{\mu_i\}_{i \in I} \subseteq \mathcal{P}(X)$, and $\mu \in \mathcal{P}(X)$. We say that $\{\mu_i\}_{i \in I}$ converges weakly to μ with respect to F , denoted by $(\mu_i \Rightarrow_{\text{wc}} \mu) F$, if $(\int f d\mu_i \longrightarrow \int f d\mu) F$ for all $f \in C_b(X)$, where $C_b(X)$ is the set of all bounded continuous functions from X to \mathbb{R} .*

The notion of weak convergence has several equivalent statements when X is a metric space.

► **Theorem 6** (The Portmanteau Theorem). *Let X be a metric space, I a set, F a filter on I , $\{\mu_i\}_{i \in I} \subseteq \mathcal{P}(X)$, and $\mu \in \mathcal{P}(X)$. Then, the following are equivalent.*

1. $(\mu_i \Rightarrow_{\text{wc}} \mu) F$.
2. For all $f \in \text{UC}_b(X)$, $(\int f d\mu_i \longrightarrow \int f d\mu) F$.
3. $(\mu_i(X) \longrightarrow \mu(X)) F$ and for every closed set C , $\text{Limsup}_F \{\mu_i(C)\}_{i \in I} \leq \mu(C)$.
4. $(\mu_i(X) \longrightarrow \mu(X)) F$ and for every open set U , $\text{Liminf}_F \{\mu_i(U)\}_{i \in I} \geq \mu(U)$.
5. For every measurable set $A \in \Sigma_X$ such that $\mu(\partial A) = 0$, $(\mu_i(A) \longrightarrow \mu(A)) F$.

The set $\text{UC}_b(X)$ denotes the set of all bounded uniform continuous functions $f : X \rightarrow \mathbb{R}$.

The Portmanteau theorem is commonly stated for probability measures rather than finite measures. Notice that we require the condition $(\mu_i(X) \longrightarrow \mu(X)) F$ in 3 and 4. This condition does not appear in the Portmanteau theorem for probability measures. In the proof for probability measures, we use $\mu_i(X) = \mu(X) = 1$. For finite measures, $\mu_i(X)$ is not equal to $\mu(X)$ in general. Hence, we use the condition $(\mu_i(X) \longrightarrow \mu(X)) F$ instead of $\mu_i(X) = \mu(X) = 1$ in order to approximate $\mu_i(X)$ to $\mu(X)$ during the proof.

3.2 Topology of Weak Convergence

Let X be a topological space. Topology of weak convergence on X , denoted by $\mathcal{O}_{\text{WC}_X}$, is the coarsest topology on $\mathcal{P}(X)$ which makes $(\lambda\mu. \int f d\mu) : \mathcal{P}(X) \rightarrow \mathbb{R}$ continuous for all $f \in C_b(X)$. As the name suggests, convergence in the topology of weak convergence is equal to weak convergence.

► **Lemma 7**. *Let X be a topological space, I a set, F a filter on I , $\{\mu_i\}_{i \in I} \subseteq \mathcal{P}(X)$, and $\mu \in \mathcal{P}(X)$. Then,*

$$(\mu_i \longrightarrow \mu) F \text{ in } (\mathcal{P}(X), \mathcal{O}_{\text{WC}_X}) \iff (\mu_i \Rightarrow_{\text{wc}} \mu) F.$$

3.3 The Lévy-Prokhorov Metric

In the lecture notes by Gaans, they only treat the case when $\mathcal{P}(X)$ is the set of all probability measures on X . We generalize their definitions and proofs to the set of all finite measures.

► **Definition 8** (Lévy-Prokhorov Metric). *For a metric space (X, d) , the Lévy-Prokhorov metric $d_{\mathcal{P}(X)}$ is a metric on $\mathcal{P}(X)$ defined by*

$$d_{\mathcal{P}(X)}(\mu, \nu) = \inf\{\alpha > 0 \mid \forall A \in \Sigma_X. \mu(A) \leq \nu(A^\alpha) + \alpha \wedge \nu(A) \leq \mu(A^\alpha) + \alpha\},$$

where $A^\alpha = \bigcup_{x \in A} \text{ball}_X(x, \alpha)$.

Note that $d_{\mathcal{P}(X)}(\mu, \nu) < \infty$ because $\infty \neq \max(\mu(X), \nu(X)) \in \{\alpha > 0 \mid \forall A \in \Sigma_X. \mu(A) \leq \nu(A^\alpha) + \alpha \wedge \nu(A) \leq \mu(A^\alpha) + \alpha\}$. The Lévy-Prokhorov metric is also expressed using open sets, closed sets, and compact sets.

► **Lemma 9.**

$$\begin{aligned} d_{\mathcal{P}(X)}(\mu, \nu) &= \inf\{\alpha > 0 \mid \forall U: \text{open. } \mu(U) \leq \nu(U^\alpha) + \alpha \wedge \nu(U) \leq \mu(U^\alpha) + \alpha\} \\ &= \inf\{\alpha > 0 \mid \forall C: \text{closed. } \mu(C) \leq \nu(C^\alpha) + \alpha \wedge \nu(C) \leq \mu(C^\alpha) + \alpha\}. \end{aligned}$$

If X is separable and complete, then

$$d_{\mathcal{P}(X)}(\mu, \nu) = \inf\{\alpha > 0 \mid \forall K: \text{compact. } \mu(K) \leq \nu(K^\alpha) + \alpha \wedge \nu(K) \leq \mu(K^\alpha) + \alpha\}.$$

The convergence with respect to the Lévy-Prokhorov metric is equivalent to the weak convergence when X is separable.

► **Theorem 10** (Theorem 4.1 and 4.2 [27]). *The following hold.*

1. $(\mathcal{P}(X), d_{\mathcal{P}(X)})$ is a metric space.
- Let I be a set, F a filter on I , $\{\mu_i\}_{i \in I} \subseteq \mathcal{P}(X)$ and $\mu \in \mathcal{P}(X)$.
2. $(\mu_i \rightarrow \mu)$ F in $(\mathcal{P}(X), \mathcal{O}_{d_{\mathcal{P}(X)}})$ implies $(\mu_i \Rightarrow_{\text{wc}} \mu)$ F .
3. If X is separable, $(\mu_i \rightarrow \mu)$ F in $(\mathcal{P}(X), \mathcal{O}_{d_{\mathcal{P}(X)}})$ if and only if $(\mu_i \Rightarrow_{\text{wc}} \mu)$ F .

The proofs are similar to the one when $\mathcal{P}(X)$ is the set of all probability measures and $F = F_{\text{seq}}$ (i.e., the convergence is not generalized by filters). The Lévy-Prokhorov metric metrizes the topology of weak convergence when X is separable.

► **Corollary 11.** *If X is separable, the Lévy-Prokhorov metric metrizes the topology of weak convergence, i.e., $\mathcal{O}_{\text{WC}_X} = \mathcal{O}_{d_{\mathcal{P}(X)}}$.*

The generalization by filters of weak convergence and Theorem 10 enables us to prove this lemma easily.

Proof. The metrizability is shown from the equivalence of convergences. From Lemma 7 and Theorem 10, convergences in $(\mathcal{P}(X), \mathcal{O}_{\text{WC}_X})$ and $(\mathcal{P}(X), \mathcal{O}_{d_{\mathcal{P}(X)}})$ are equivalent for all filters. Hence, we have $\mathcal{O}_{\text{WC}_X} = \mathcal{O}_{d_{\mathcal{P}(X)}}$ from Corollary 2. ◀

Even though Corollary 11 is a well-known result, only a few books include its proof. We found two books showing Corollary 11. In the book by Billingsley [3], they directly prove the equivalence by examining neighborhoods. In the book by Deuschel and Stroock [4], they prove the equivalence by using the equivalence of convergence with respect to the filter F_{seq} (i.e., sequences are defined on \mathbb{N} such as $\{\mu_n\}_{n \in \mathbb{N}}$). As we stated in Corollary 2, their proof requires the assumption that $(\mathcal{P}(X), \mathcal{O}_{\text{WC}_X})$ is first-countable. They use the fact that $(\mathcal{P}(X), \mathcal{O}_{\text{WC}_X})$ is second-countable (and thus also first-countable) without providing

any proof that it is second-countable. If we follow their proof, we will need additional efforts to show the first countability of $(\mathcal{P}(X), \mathcal{O}_{\text{WC}_X})$. In our proof, we do not need the first countability because we generalized the notion of weak convergence and equivalence of convergence by filters.

Thanks to Corollary 11, we identify $(\mathcal{P}(X), \mathcal{O}_{d_{\mathcal{P}(X)}})$ with $(\mathcal{P}(X), \mathcal{O}_{\text{WC}_X})$, when X is a separable metric space.

► **Proposition 12** (Proposition 4.4 [27]). *If X is a separable metric space, then $\mathcal{P}(X)$ is also a separable metric space.*

The proof is similar to the one when $\mathcal{P}(X)$ is the set of all probability measures on X . If $\{a_n\}_{n \in \mathbb{N}}$ is a dense subset of X , then

$$\bigcup_{k \in \mathbb{N}} \{r_0 \delta_{a_0} + \dots + r_k \delta_{a_k} \mid r_0, \dots, r_k \in \mathbb{Q} \cap [0, \infty)\}$$

is a countable dense subset of $\mathcal{P}(X)$, where δ_a denotes the Dirac measure centered at a .

3.4 Implementation in Isabelle/HOL

We explain our implementation of the topology of weak convergence and the Lévy-Prokhorov metric. We sometimes use usual mathematical symbols in source code for readability.

Topology of Weak Convergence

We define the topology of weak convergence by combining existing constants which generate topological spaces. Let f be a bounded continuous function on X and \mathcal{O}_f the least topology on $\mathcal{P}(X)$, which makes $(\lambda N. \int x. f x \partial N)$ continuous². Then, $(\mathcal{P}(X), \mathcal{O}_f)$ is written in Isabelle/HOL as follows:

$$(\mathcal{P}(X), \mathcal{O}_f) = \text{pullback-topology } \mathcal{P}(X) (\lambda N. \int x. f x \partial N) \mathbb{R},$$

where

$$\begin{aligned} \text{pullback-topology} &:: 'a \text{ set} \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'b \text{ topology} \Rightarrow 'a \text{ topology} \\ \text{pullback-topology } A \ f \ Y &= \text{The least topology on } A \text{ which makes } f : A \rightarrow Y \text{ continuous.} \end{aligned}$$

The set of all open sets \mathcal{O}_f is extracted as follows:

$$\mathcal{O}_f = \text{Collect } (\text{openin } (\text{pullback-topology } \mathcal{P}(X) (\lambda N. \int x. f x \partial N) \mathbb{R})),$$

where

$$\begin{aligned} \text{openin} &:: 'a \text{ topology} \Rightarrow 'a \text{ set} \Rightarrow \text{bool}, & \text{openin } X \ U &\iff U \text{ is an open set of } X. \\ \text{Collect} &:: ('a \Rightarrow \text{bool}) \Rightarrow 'a \text{ set}, & \text{Collect } P &= \{x. P \ x\}. \end{aligned}$$

Finally, we define the topology of weak convergence $(\mathcal{P}(X), \mathcal{O}[\bigcup_{f \in C_b(X)} \mathcal{O}_f])$.

definition *weak-conv-topology* :: *'a topology* \Rightarrow *'a measure topology* **where**

weak-conv-topology $X \equiv \text{topology-generated-by}$

$$\begin{aligned} &(\bigcup f \in \{f. \text{continuous-map } X \ \mathbb{R} \ f \wedge (\exists B. \forall x \in \text{topspace } X. |f \ x| \leq B)\} . \\ &\text{Collect } (\text{openin } (\text{pullback-topology } \mathcal{P}(X) (\lambda N. \int x. f x \partial N) \mathbb{R}))) \end{aligned}$$

² In Isabelle/HOL, the Lebesgue integral of f with respect to N is denoted by $\int x. f x \partial N$.

The term *continuous-map* $X \mathbb{R} f$ means that f is a continuous map from X to \mathbb{R} and *topology-generated-by* receives a set of sets and returns the least topology, including the received set. The topological space *weak-conv-topology* X meets the requirements of the topology of weak convergence.

lemma *continuous-map-weak-conv-topology*:

assumes *continuous-map* $X \mathbb{R} f$ **and** $\bigwedge x. x \in \text{topspace } X \implies |f\ x| \leq B$
shows *continuous-map* (*weak-conv-topology* X) $\mathbb{R} (\lambda N. \int x. f\ x\ \partial N)$

lemma *weak-conv-topology-minimal*:

assumes *topspace* $Y = \mathcal{P}(X)$
and $\bigwedge f B. \text{continuous-map } X \mathbb{R} f \implies (\bigwedge x. x \in \text{topspace } X \implies |f\ x| \leq B)$
 $\implies \text{continuous-map } Y \mathbb{R} (\lambda N. \int x. f\ x\ \partial N)$
shows *openin* (*weak-conv-topology* X) $U \implies \text{openin } Y\ U$

The first lemma guarantees that *weak-conv-topology* X makes $(\lambda N. \int x. f\ x\ \partial N)$ continuous and the second lemma states that *weak-conv-topology* X is the least topology in such topological spaces.

From Lemma 7, weak convergence and convergence in the topology of weak convergence are equivalent. Thus, we define the notion of weak convergence as an abbreviation of the convergence in the topology of weak convergence. Then, the usual definition of weak convergence (Definition 5) is shown as a lemma.

abbreviation *weak-conv-on* :: ('a \Rightarrow 'b measure) \Rightarrow 'b measure \Rightarrow 'a filter \Rightarrow 'b topology \Rightarrow bool
where *weak-conv-on* $Ni\ N\ F\ X \equiv \text{limitin } (\text{weak-conv-topology } X)\ Ni\ N\ F$

lemma *weak-conv-on-def'*:

assumes $\bigwedge i. Ni\ i \in \mathcal{P}(X)$ **and** $N \in \mathcal{P}(X)$
shows *weak-conv-on* $Ni\ N\ F\ X \longleftrightarrow$
 $(\forall f. \text{continuous-map } X \mathbb{R} f \longrightarrow (\exists B. \forall x \in \text{topspace } X. |f\ x| \leq B)$
 $\longrightarrow ((\lambda i. \int x. f\ x\ \partial Ni\ i) \longrightarrow (\int x. f\ x\ \partial N))\ F)$

The term *limitin* (*weak-conv-topology* X) $Ni\ N\ F$ denotes $(Ni \longrightarrow N)\ F$ in $(\mathcal{P}(X), \mathcal{O}_{WC_X})$ in our presentation.

The Lévy-Prokhorov Metric

To formalize the Lévy-Prokhorov metric in Isabelle/HOL, we use the set-based metric space library, which has recently appeared in the standard distribution since Isabelle 2023. The library was ported from HOL Light by Paulson [20]. Another metric space library that has been used is based on type classes [13]. While set-based metric space enable us to treat metric spaces with arbitrary carrier sets, type-based metric spaces only work for an entire type. For each type, there can only be one *metric-space* instance. This works well for situations where there is a “canonical” metric space for a type, but it lacks the flexibility to describe, for instance, the set of all metric spaces with a given carrier set. The library based on type classes is unsuitable for our use because we use the set of finite measures on a measurable space, which is not the universe of the type.

In Isabelle/HOL’s library, the set-based metric space is defined with the **locale** command.

locale *Metric-space* =

fixes $M :: 'a\ \text{set}$ **and** $d :: 'a \Rightarrow 'a \Rightarrow \text{real}$
assumes *nonneg*: $\bigwedge x\ y. 0 \leq d\ x\ y$
assumes *commute*: $\bigwedge x\ y. d\ x\ y = d\ y\ x$
assumes *zero*: $\bigwedge x\ y. \llbracket x \in M; y \in M \rrbracket \implies d\ x\ y = 0 \longleftrightarrow x=y$
assumes *triangle*: $\bigwedge x\ y\ z. \llbracket x \in M; y \in M; z \in M \rrbracket \implies d\ x\ z \leq d\ x\ y + d\ y\ z$

21:10 A Formalization of the Lévy-Prokhorov Metric in Isabelle/HOL

The **locale** command introduces a context. In this case, a set M and a function d are fixed and the four assumptions hold, i.e., (M, d) forms a metric space in the context of *Metric-space*. Notice that the non-negativity and commutativity must hold on not only M but the whole type universe. These assumptions make it easier to use non-negativity and commutativity in proofs, and do not change the essential structure of the metric space. Owing to these assumptions, we need to take care of non-negativity and commutativity even outside the carrier set when we define a metric space.

We introduced a new locale *Levy-Prokhorov* which is logically equivalent to *Metric-space*.

locale *Levy-Prokhorov* = *Metric-space*

Remember that the Lévy-Prokhorov metric is defined as follows.

$$d_{\mathcal{P}(X)}(\mu, \nu) = \inf\{\alpha > 0 \mid \forall A \in \Sigma_X. \mu(A) \leq \nu(A^\alpha) + \alpha \wedge \nu(A) \leq \mu(A^\alpha) + \alpha\},$$

$$\text{where } A^\alpha = \bigcup_{x \in A} \text{ball}_X(x, \alpha).$$

Hence, we define the Lévy-Prokhorov metric in the context of *Levy-Prokhorov* as follows:

definition $\mathcal{P} \equiv \{N. \text{sets } N = \text{sets (borel-of mtopology)} \wedge \text{finite-measure } N\}$

definition *LPm* :: 'a measure \Rightarrow 'a measure \Rightarrow real **where**

LPm N $L \equiv$

if $N \in \mathcal{P} \wedge L \in \mathcal{P}$ then

$$\left(\prod \{e. e > 0 \wedge (\forall A \in \text{sets (borel-of mtopology)}. \text{measure } N \ A \leq \text{measure } L \ (\bigcup_{a \in A}. \text{mball } a \ e) + e \wedge \text{measure } L \ A \leq \text{measure } N \ (\bigcup_{a \in A}. \text{mball } a \ e) + e)\} \right)$$

else 0

In the definition of \mathcal{P} , the projection function *sets* receives a measure and returns the σ -algebra on which the measure is defined. The constant *mtopology* denotes the topological space induced by (M, d) , and *borel-of mtopology* denotes the Borel space generated from *mtopology*. In the definition of *LPm*, *measure* N A corresponds to $N(A)$ in usual mathematics notation. Notice that *LPm* returns 0 when one of the arguments is not a member of \mathcal{P} because the set to which we apply infimum might be empty when *LPm* receives an infinite measure. In Isabelle/HOL, the infimum operator on real numbers does not return ∞ nor any specific value when applied to the empty set; i.e., the value of $\prod \emptyset$ is unknown. This is a problem because *LPm* needs to be a non-negative function on the whole type universe due to the definition of *Metric-space*.

We then prove that $(\mathcal{P}, \text{LPm})$ is a metric space in the context of *Levy-Prokhorov*.

sublocale *LPm*: *Metric-space* \mathcal{P} *LPm*

The reader might wonder why we define a new locale *Levy-Prokhorov*, which is logically equivalent to *Metric-space*, rather than using *Metric-space* directly. If we try to define the Lévy-Prokhorov metric in the context of *Metric-space* without introducing a new locale, it does not work.

context *Metric-space*

begin

definition $\mathcal{P} \equiv \{N. \text{sets } N = \text{sets (borel-of mtopology)} \wedge \text{finite-measure } N\}$

definition *LPm* $\equiv \dots$

sublocale *LPm*: *Metric-space* \mathcal{P} *LPm*

end

The problem is that we try to instantiate *Metric-space* inside the context of *Metric-space*. This causes Isabelle to build an infinite chain; thus, Isabelle does not terminate. This workaround is explained in the Isabelle tutorial on locales [2].

4 Prokhorov's Theorem

One of the important results related to the Lévy-Prokhorov metric is Prokhorov's theorem. In a typical situation in probability theory or statistics, one may want to know whether a sequence of measures has a limit or at least has a converging subsequence. Prokhorov's theorem is applied to prove the existence of a converging subsequence. The theorem is used in proofs for various important results such as the central limit theorem, Sanov's theorem, and the existence of *optimal coupling*. The central limit theorem and Sanov's theorem are key concepts in probability theory. The central limit theorem states that under appropriate conditions, the distribution of normalized sample means converges weakly to the standard normal distribution. Sanov's theorem is an important result in the large deviation theory (e.g. Section 3.2 [4]). The theorem describes the asymptotic behavior of atypical samples and gives evidence why we use the relative entropy (Kullback-Leibler divergence) to evaluate estimated distributions. Both the central limit theorem and Sanov's theorem use Prokhorov's theorem. In transportation theory, a *coupling* is a *plan* how to move resources from supply areas to demand areas. A coupling is represented as a measure satisfying certain conditions. An optimal coupling is a coupling that minimizes the total *cost* of transporting resources. In the proof of the existence of an optimal coupling, Prokhorov's theorem is essential [28, 29].

In this section, we discuss Prokhorov's theorem and related topics.

4.1 Regular Measures

We define the notion of regular measures and tightness of measures. The regularity of measures gives ways to approximate a measured value $\mu(A)$ by open sets, closed sets, and compact sets. The tightness of measures is used to express a condition in Prokhorov's theorem.

► **Definition 13.** *Let X be a topological space. A measure μ on X is called:*

1. *inner regular if $\mu(A) = \sup\{\mu(C) \mid C \subseteq A, C \text{ is closed}\}$ for all measurable sets A ,*
2. *outer regular if $\mu(A) = \inf\{\mu(U) \mid A \subseteq U, U \text{ is open}\}$ for all measurable sets A , and*
3. *regular if μ is inner regular and outer regular.*

► **Proposition 14.** *Let X be a metrizable space. Then, any finite measure on X is regular.*

► **Remark 15.** This definition of inner regular by Gaans is different from the standard definition. In general, a measure μ on X is called inner regular if

1'. $\mu(A) = \sup\{\mu(K) \mid K \subseteq A, K \text{ is compact}\}$ for all measurable sets A .

This definition is stronger than the condition 1 in Definition 13, when every compact set is closed (e.g. when X is metrizable). As we will see soon, Proposition 14 still holds even if we use the condition 1' as inner regularity when X is a Polish space (Corollary 19).

Proposition 14 has been already included in the standard Isabelle/HOL's library. They assume that X is a Polish space and use the condition 1' as the definition of inner regular. Their formalization is restricted to measures on the Borel space of topological space on type classes; thus, they treat only when X is the universal set such as \mathbb{R} . We formalize the general result when X is an arbitrary metrizable space or a Polish space.

Next, we define tightness.

► **Definition 16** (Tightness). *Let X be a topological space and $\Gamma \subseteq \mathcal{P}(X)$. We call Γ tight if for every $\varepsilon > 0$, there exists a compact set K of X such that $\mu(X - K) \leq \varepsilon$ for all $\mu \in \Gamma$. A measure μ on X is tight if $\{\mu\}$ is tight.*

The existing definition of tightness in Isabelle/HOL's library is restricted to when Γ is a sequence on \mathbb{N} of probability measures on \mathbb{R} .

► **Lemma 17.** *If X is metrizable and μ is a tight measure on X , then $\mu(A) = \sup\{\mu(K) \mid K \subseteq A, K \text{ is compact}\}$ for all measurable sets A .*

► **Theorem 18.** *If X is a Polish space, then any finite measure on X is tight.*

► **Corollary 19.** *If X is a Polish space and μ is a finite measure on X , then $\mu(A) = \sup\{\mu(K) \mid K \subseteq A, K \text{ is compact}\}$ for all measurable sets A .*

4.2 Prokhorov's Theorem

We formalize Prokhorov's theorem. Let $\mathcal{P}_r(X) = \mathcal{P}(X) \cap \{\mu \mid \mu(X) \leq r\}$ for $r \geq 0$.

► **Theorem 20** (Prokhorov's Theorem). *Let X be a Polish space and $\Gamma \subseteq \mathcal{P}_r(X)$ for some $r \geq 0$. Then, the following are equivalent.*

1. Γ is relatively compact.
2. Γ is tight.

► **Remark 21.** Actually, the assumption $\Gamma \subseteq \mathcal{P}_r(X)$ is relaxed to $\Gamma \subseteq \mathcal{P}(X)$ in the proof that 1 implies 2. The completeness assumption is not required in the proof that 2 implies 1.

The following corollary is applied to show the existence of a converging subsequence.

► **Corollary 22.** *Let X be a separable metrizable space and $\{\mu_n\}_{n \in \mathbb{N}} \subseteq \mathcal{P}_r(X)$ for some $r \geq 0$. If $\{\mu_n\}_{n \in \mathbb{N}}$ is tight, then there exists a subsequence $\{\mu_{n_k}\}_{k \in \mathbb{N}}$ and $\mu \in \mathcal{P}_r(X)$ such that $(\mu_{n_k} \Rightarrow_{\text{wc}} \mu) F_{\text{seq}}$.*

Avigad et al. formalized the above corollary when $\{\mu_n\}_{n \in \mathbb{N}}$ is a sequence of probability measures on \mathbb{R} and applied it to prove the central limit theorem [1]. In the case of probability measures on \mathbb{R} , there is a simpler proof using Helly's selection theorem. In general case, we need to prove in other way because the proof using Helly's selection theorem uses cumulative distribution function; i.e., X needs to be \mathbb{R} .

The proof that 1 implies 2 in Prokhorov's theorem is more straightforward. The proof that 2 implies 1 requires more effort to prove for us. We do not discuss the details of the proof. Instead, we explain a key lemma for the proof that 2 implies 1.

► **Lemma 23.** *If X is a compact metric space, then \mathcal{P}_r is compact.*

The proof relies on results from vector space theory such as Alaoglu's theorem and the Riesz representation theorem. Although these theorems need to be stated in set-based vector space in Isabelle/HOL for our use, most of Isabelle/HOL's vector space library is based on type classes. The set-based vector space library by Lee [16] includes only basic definitions. Thiemann and Yamada also formalized a set-based vector space [26]. However, their work treats only finite-dimensional spaces. Since we are interested in the Lévy-Prokhorov metric rather than vector space theory, we leave the development of the set-based vector space library for future work. Thus, we formalize positive linear functionals used in proofs and their properties without mentioning vector spaces. For Alaoglu's theorem, we prove a special case of the theorem.

Proof. The idea of the proof is to make a homeomorphism between \mathcal{P}_r and a compact space. Let Φ be

$$\Phi = \left(\mathbb{R}^{C(X)} \right) \cap \{ \varphi \mid \varphi \text{ is a positive linear functional} \wedge \varphi(1) \leq r \}. \quad (1)$$

Remember that an element φ of $\mathbb{R}^{C(X)}$ is a function $\varphi : C(X) \rightarrow \mathbb{R}$. We denote $\varphi(f)$ by φ_f . Then, the linearity of $\varphi \in \Phi$ means that for all $\alpha, \beta \in \mathbb{R}$ and $f, g \in C(X)$, $\varphi_{\alpha f + \beta g} = \alpha \varphi_f + \beta \varphi_g$. The positiveness means that for all $f \in C(X)$ such that $f \geq 0$, $\varphi_f \geq 0$.

We assume that Φ is equipped with the subspace topology of the product topology $\mathbb{R}^{C(X)}$ (subspace topology of the *weak* topology*). We define the function T from \mathcal{P}_r to Φ by $T(\mu)_f = \int f d\mu$. It is easy to check that $T(\mu) \in \Phi$, and T is a sequential homeomorphic map. For instance, the linearity of the integral implies the linearity of $T(\mu)$. The function T is bijective by the Riesz representation theorem (Corollary 31). As Gaans stated, Φ is metrizable³. Thus, T is a homeomorphism⁴. Furthermore, Φ is compact by the special case of Alaoglu's theorem (Theorem 28). Hence, \mathcal{P}_r is compact. ◀

► **Remark 24.** In the lecture notes, Gaans stated that the sequential compactness of a closed subset of Φ follows from its compactness. This statement is true because Φ is metrizable. However, they did not mention that in their proof.

Prokhorov's theorem is applied to prove the completeness of the Lévy-Prokhorov metric.

► **Corollary 25.** *If X is separable and complete, then $(\mathcal{P}(X), d_{\mathcal{P}(X)})$ is complete.*

When we prove the existence of a limit of a Cauchy sequence $\{\mu_n\}_{n \in \mathbb{N}} \subseteq \mathcal{P}(X)$, we use Prokhorov's theorem as $\Gamma = \{\mu_n\}_{n \in \mathbb{N}}$. Hence, we need to show that $\{\mu_n\}_{n \in \mathbb{N}} \subseteq \Gamma_r$ for some $r \geq 0$. This follows from the fact that $\{\mu_n\}_{n \in \mathbb{N}}$ is a Cauchy sequence.

As a consequence of Corollary 11, Proposition 12, and Corollary 25, we have the following.

► **Corollary 26.** *If X is a Polish space, then so is $\mathcal{P}(X)$.*

4.3 Alaoglu's Theorem

Alaoglu's theorem (sometimes called the Banach-Alaoglu theorem) is an important result in functional analysis. The theorem states that the closed unit ball of the dual space of a normed vector space is compact. Let Y be a vector space over \mathbb{R} and Y^* the dual space of Y . The *weak* topology* is a topology on Y^* , which is the coarsest topology that makes every $(\lambda f, f(y)) : Y^* \rightarrow \mathbb{R}$ continuous. The original statement of the Alaoglu's theorem is the following.

► **Theorem 27 (Alaoglu's Theorem).** *Let Y be a normed vector space and $B^* = \{ \varphi \in Y^* \mid \|\varphi\| \leq r \}$. Then, B^* is compact in Y^* with respect to the weak* topology.*

We do not prove the above form of the theorem due to the lack of set-based vector space library in Isabelle/HOL. Instead, we prove a special case of Alaoglu's theorem for our use.

³ Since X is compact, $C(X)$ along with the topology of uniform convergence is separable (Theorem 2.4.3 [24]). Let $\{g_n\}_{n \in \mathbb{N}}$ be a dense subset of $C(X)$. Then, the metric on Φ is, for instance, given by

$$d(\varphi, \psi) = \sum_{n=0}^{\infty} \frac{1}{2^{n+1}} \min(1, |\varphi(g_n) - \psi(g_n)|).$$

⁴ A function f from a first-countable space is continuous iff it is sequentially continuous.

► **Theorem 28.** *If a topological space X is compact, then Φ defined by (1) in the proof of Lemma 23 is compact.*

► **Remark 29.** While the Alaoglu's theorem says that $\{\varphi \in C(X)^* \mid \|\varphi\| \leq r\}$ is compact, Theorem 28 states that $\Phi = \{\varphi \in C(X)^* \mid \|\varphi\| \leq r \wedge \varphi \text{ is positive}\}$ is compact. Note that $\|\varphi\| = \varphi(1)$ when $\varphi \in C(X)^*$ is positive.

Proof Outline. We formalize the theorem following the proof in the lecture notes by Heil [6]. The proof is simple. We first observe that $\prod_{f \in C(X)} [-r\|f\|, r\|f\|]$ is compact in $\mathbb{R}^{C(X)}$ by Tychonoff's theorem. Note that every $f \in C(X)$ is bounded because X is compact. We then show that $\Phi \subseteq \prod_{f \in C(X)} [-r\|f\|, r\|f\|]$ and Φ is closed. The fact that Φ is closed is shown by the characterization of closed sets by limit (Lemma 1). ◀

4.4 The Riesz Representation Theorem

The Riesz representation theorem (sometimes called the Riesz-Markov representation theorem or Riesz-Markov-Kakutani representation theorem) states that a real-valued (or complex-valued) positive linear functional is represented by the Lebesgue integration with respect to a unique measure. We prove the Riesz representation theorem following the book by Rudin [22].

► **Theorem 30** (The Riesz representation theorem). *Let X be a locally compact Hausdorff space and φ a real-valued positive linear functional on $C_c(X)$, where $C_c(X)$ is the set of all continuous functions on X whose closed support is compact. Then, there exists a σ -algebra \mathcal{M} in X and a unique measure μ on (X, \mathcal{M}) such that:*

- $\varphi(f) = \int f d\mu$ for all $f \in C_c(X)$,
- $\Sigma_X \subseteq \mathcal{M}$,
- $\mu(K) < \infty$ for all compact sets K ,
- $\mu(A) = \inf\{\mu(U) \mid A \subseteq U, U \text{ is open}\}$ for all $A \in \mathcal{M}$,
- $\mu(A) = \sup\{\mu(K) \mid K \subseteq A, K \text{ is compact}\}$ for all open sets A and for all $A \in \mathcal{M}$ such that $\mu(A) < \infty$, and
- μ is a complete measure, i.e., if $E \in \mathcal{M}$, $A \subseteq E$, and $\mu(E) = 0$, then $A \in \mathcal{M}$.

In the book, the proof of the Riesz representation theorem is divided into ten steps and uses two lemmas. Their proofs consist of around nine pages, whereas we spent more than 2,100 lines for their proofs. The proof requires Urysohn's lemma on locally compact Hausdorff space. Although Isabelle/HOL's library has several forms of Urysohn's lemmas and lemmas related to locally compact spaces, the library does not include Urysohn's lemma on locally compact Hausdorff space. Hence, we formalized the lemma by ourselves.

We use the following corollary in the proof of Prokhorov's theorem.

► **Corollary 31.** *Let X be a compact metric space and φ be a real-valued positive linear functional on $C(X)$. Then, there exists a unique measure μ on X such that for all $f \in C(X)$,*

$$\varphi(f) = \int f d\mu.$$

5 Measurable Spaces of Finite Measures

In this section, we discuss the measurable space of all finite measures. Measurable spaces on a set of measures are used in stochastic processes and semantics of probabilistic programs. In stochastic processes, measures are usually indexed by time or states. A stochastic process

is interpreted as a measurable function from its index set to the space of measures. In the semantics of probabilistic programs, the Giry monad G (or sub-Giry monad) gives a standard semantics of probabilistic programs where $G(M)$ is the measurable space of all probability measures on M defined independently from metric or topology.

We will show that this type of measurable space of all finite measures is generated from the topology of weak convergence when the underlying topological space is a Polish space.

► **Definition 32.** *Let M be a measurable space. The space of finite measures on M is denoted by $(\mathcal{P}(M), \Sigma_{\mathcal{P}(M)})$, where $\Sigma_{\mathcal{P}(M)}$ is the least σ -algebra that makes $(\lambda\mu. \mu(A))$ measurable for all $A \in \Sigma_M$.*

Note that this definition does not use any metric or topology. In Isabelle/HOL's library, the space of all sub-probability measures $\mathcal{P}_{\text{sprob}}(M)$ and the space of all probability measures $\mathcal{P}_{\text{prob}}(M)$ are already formalized by Eberl et al. [5] (*subprob-algebra M* and *prob-algebra M* , respectively). We have formalized the space of all finite measures in the same way as *subprob-algebra*. Subsequently, we have shown that $\mathcal{P}_{\text{sprob}}(M)$ and $\mathcal{P}_{\text{prob}}(M)$ are subspaces of $\mathcal{P}(M)$.

The following lemma follows immediately from the Portmanteau theorem⁵.

► **Lemma 33** (Corollary 17.21 [14]). *For open $U \subseteq X$, $(\lambda\mu. \mu(U)) : (\mathcal{P}(X), \mathcal{O}_{d_{\mathcal{P}(X)}}) \rightarrow \mathbb{R}$ is lower semi-continuous. For closed $C \subseteq X$, $(\lambda\mu. \mu(C)) : (\mathcal{P}(X), \mathcal{O}_{d_{\mathcal{P}(X)}}) \rightarrow \mathbb{R}$ is upper semi-continuous.*

► **Corollary 34.** $\Sigma_{\mathcal{P}(X)} \subseteq \Sigma_{d_{\mathcal{P}(X)}}$.

Proof. From the definition of $\Sigma_{\mathcal{P}(X)}$, it is sufficient to show that for all $A \in \Sigma_X$, $(\lambda\mu. \mu(A))$ is a measurable function from $(\mathcal{P}(X), \Sigma_{d_{\mathcal{P}(X)}})$ to \mathbb{R} . It is easy to check the measurability because by Lemma 33, $(\lambda\mu. \mu(U)) : (\mathcal{P}(X), d_{\mathcal{P}(X)}) \rightarrow \mathbb{R}$ is lower semi-continuous for all open sets $U \subseteq X$, hence measurable. ◀

The inverse inclusion holds when X is separable and complete.

► **Theorem 35.** *If a metric space X is separable and complete, then $\Sigma_{\mathcal{P}(X)} = \Sigma_{d_{\mathcal{P}(X)}}$.*

► **Corollary 36.** *If X is a Polish space, then $\Sigma_{\mathcal{P}(X)} = \Sigma_{(\mathcal{P}(X), \mathcal{O}_{\text{wC}_X})}$.*

We constructed the proof of Theorem 35 by ourselves because we could not find any proof for the statement. We provide an informal proof here.

Proof of Theorem 35. Since $\Sigma_{d_{\mathcal{P}(X)}}$ is generated from closed balls, it is sufficient to prove that every closed ball is a member of $\Sigma_{\mathcal{P}(X)}$. Let μ be a finite measure on X and $\varepsilon \geq 0$. Our goal is to show that $cBall_{\mathcal{P}(X)}(\mu, \varepsilon) \in \Sigma_{\mathcal{P}(X)}$. Let \mathcal{O}_b be a countable base of X and \mathcal{O}_{bfU} the set of all finite unions of elements of \mathcal{O}_b . Then, \mathcal{O}_{bfU} is also countable.

▷ **Claim 37.**

$$cBall_{\mathcal{P}(X)}(\mu, \varepsilon) = \bigcap_{U \in \mathcal{O}_{\text{bfU}}} \left(\bigcap_{n \in \mathbb{N}} (\lambda\nu. \nu(U))^{-1} \left(-\infty, \mu \left(U^{(\varepsilon + \frac{1}{1+n})} \right) + \varepsilon + \frac{1}{1+n} \right) \cap \right. \\ \left. (\lambda\nu. \nu \left(U^{(\varepsilon + \frac{1}{1+n})} \right))^{-1} \left[\mu(U) - \left(\varepsilon + \frac{1}{1+n} \right), \infty \right) \right) \quad (2)$$

⁵ Remember that for a first-countable space X ,

- $f : X \rightarrow \mathbb{R}$ is lower semi-continuous iff $(x_n \rightarrow x) F_{\text{seq}}$ in X implies $f(x) \leq \text{Liminf}_{F_{\text{seq}}} \{f(x_n)\}_{n \in \mathbb{N}}$.
- $f : X \rightarrow \mathbb{R}$ is upper semi-continuous iff $(x_n \rightarrow x) F_{\text{seq}}$ in X implies $f(x) \geq \text{Limsup}_{F_{\text{seq}}} \{f(x_n)\}_{n \in \mathbb{N}}$.

If the above claim is shown, $cBall(\mu, \varepsilon) \in \Sigma_{\mathcal{P}(X)}$ follows from the definition of $\Sigma_{\mathcal{P}(X)}$.

The inclusion \subseteq in equation (2) is directly proven by unfolding the definition of the Lévy-Prokhorov metric. Hence, we show \supseteq of (2). Let us assume that ν is a member of the right hand side of (2). Then, for all $U \in \mathcal{O}_{bFU}$ and $n \in \mathbb{N}$, we have

$$\nu(U) \leq \mu\left(U^{(\varepsilon + \frac{1}{1+n})}\right) + \varepsilon + \frac{1}{1+n}, \quad \mu(U) \leq \nu\left(U^{(\varepsilon + \frac{1}{1+n})}\right) + \varepsilon + \frac{1}{1+n}. \quad (3)$$

We show $\nu \in cBall_{\mathcal{P}(X)}(\mu, \varepsilon)$ by proving that $d_{\mathcal{P}(X)}(\mu, \nu) < \varepsilon'$ for all $\varepsilon' > \varepsilon$. Let $\varepsilon' > \varepsilon$, then there exists $n \in \mathbb{N}$ such that $\varepsilon + \frac{1}{1+n} < \varepsilon'$. For an open set $A \subseteq X$, we have

$$\begin{aligned} \mu(A) &= \sup\{\mu(K) \mid K \subseteq A, K \text{ is compact}\} \quad (\text{Corollary 19}) \\ &\leq \sup\{\mu(U) \mid U \subseteq A, U \in \mathcal{O}_{bFU}\} \\ &\leq \sup\left\{\nu\left(U^{(\varepsilon + \frac{1}{1+n})}\right) + \varepsilon + \frac{1}{1+n} \mid U \subseteq A, U \in \mathcal{O}_{bFU}\right\} \quad (\text{by (3)}) \\ &\leq \nu\left(A^{(\varepsilon + \frac{1}{1+n})}\right) + \varepsilon + \frac{1}{1+n}. \end{aligned} \quad (4)$$

The inequality (4) above is shown as follows: Since \mathcal{O}_b is a base of X , there exists $\mathcal{O}' \subseteq \mathcal{O}_b$ such that $A = \bigcup_{U \in \mathcal{O}'} U$. If $K \subseteq A$ is compact, there exists a finite subset $\mathcal{O}'_{\text{fin}} \subseteq \mathcal{O}'$ such that $K \subseteq \bigcup_{U \in \mathcal{O}'_{\text{fin}}} U$. By the definition of \mathcal{O}_{bFU} , we have $\bigcup_{U \in \mathcal{O}'_{\text{fin}}} U \in \mathcal{O}_{bFU}$. Thus, (4) holds.

Similarly, we have $\nu(A) \leq \mu\left(A^{(\varepsilon + \frac{1}{1+n})}\right) + \varepsilon + \frac{1}{1+n}$ for all open sets $A \subseteq X$. Hence,

$$\begin{aligned} d_{\mathcal{P}(X)}(\mu, \nu) &= \inf\{\alpha > 0 \mid \forall A: \text{open. } \mu(A) \leq \nu(A^\alpha) + \alpha \wedge \nu(A) \leq \mu(A^\alpha) + \alpha\} \\ &\leq \varepsilon + \frac{1}{1+n} < \varepsilon'. \end{aligned} \quad \blacktriangleleft$$

Corollary 36 is applied to prove that the space of finite measures is a standard Borel space, which is a measurable space generated from a Polish space. Many practical spaces (e.g. \mathbb{R} , \mathbb{N} , and countable product spaces of standard Borel spaces) are standard Borel spaces. Standard Borel spaces have good properties such as Kuratowski's theorem stating that any standard Borel space is either a countable discrete space or isomorphic to \mathbb{R} . In our previous work [11], we formalized the notion of standard Borel space. As a consequence of Corollary 26 and Corollary 36, we obtain the following.

► **Corollary 38.** *If M is a standard Borel space, then so is $\mathcal{P}(M)$.*

► **Corollary 39.** *If M is a standard Borel space, then $\mathcal{P}_{\text{sprob}}(M)$ and $\mathcal{P}_{\text{prob}}(M)$ are also standard Borel spaces.*

6 Conclusion

We formalized the Lévy-Prokhorov metric and related notions to show that the measurable space of finite measures on a standard Borel space is a standard Borel space. We also showed important mathematical theorems such as the Riesz representation theorem and Prokhorov's theorem. Our formalization consists of around 11,000 lines (4,400 lines for the Riesz representation theorem and 6,600 lines for the Lévy-Prokhorov metric) including comments and blank lines.


Formalization of the large deviation theory and transportation theory could be interesting future works. Both of these theories depend on Prokhorov's theorem for the proof of important theorems in their fields, namely, Sanov's theorem and the existence of an optimal coupling.

References

- 1 Jeremy Avigad, Johannes Hölzl, and Luke Serafin. A formally verified proof of the central limit theorem. *Journal of Automated Reasoning*, 59(4):389–423, 2017. doi:10.1007/s10817-017-9404-x.
- 2 Clemens Ballarin. Tutorial to locales and locale interpretation. *Contribuciones científicas en honor de Mirian Andrés Gómez*, pages 123–140, 2010.
- 3 Patrick Billingsley. *Convergence of Probability Measures*. Wiley series in probability and mathematical statistics, Tracts on probability and statistics. Wiley, New York, United States, 1968.
- 4 Jean-Dominique Deuschel and Daniel W. Stroock. *Large Deviations*, volume 137 of *Pure and Applied Mathematics*. Elsevier Science, 1989.
- 5 Manuel Eberl, Johannes Hölzl, and Tobias Nipkow. A verified compiler for probability density functions. In *European Symposium on Programming (ESOP 2015)*, volume 9032 of *LNCS*, pages 80–104. Springer, 2015. doi:10.1007/978-3-662-46669-8_4.
- 6 Christopher E. Heil. Alaoglu’s theorem. <https://heil.math.gatech.edu/6338/summer08/section9f.pdf>, 2008. Lecture notes on MATH 6338 (Real Analysis II) at Georgia Insisute of Technology, Accessed: January 5, 2024.
- 7 Christopher E. Heil. Nets, directed sets, and convergence. <https://heil.math.gatech.edu/6338/summer08/section9b.pdf>, 2008. Lecture notes on MATH 6338 (Real Analysis II) at Georgia Insisute of Technology, Accessed: January 5, 2024.
- 8 Chris Heunen, Ohad Kammar, Sam Staton, and Hongseok Yang. A convenient category for higher-order probability theory. In *Proceedings of the 32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2017*. IEEE Press, 2017. doi:10.1109/lics.2017.8005137.
- 9 Michikazu Hirata. The Lévy-Prokhorov metric. *Archive of Formal Proofs*, June 2024. , Formal proof development. URL: https://isa-afp.org/entries/Levy_Prokhorov_Metric.html.
- 10 Michikazu Hirata. The Riesz representation theorem. *Archive of Formal Proofs*, June 2024. , Formal proof development. URL: https://isa-afp.org/entries/Riesz_Representation.html.
- 11 Michikazu Hirata, Yasuhiko Minamide, and Tetsuya Sato. Semantic foundations of higher-order probabilistic programs in Isabelle/HOL. In Adam Naumowicz and René Thiemann, editors, *14th International Conference on Interactive Theorem Proving (ITP 2023)*, volume 268 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 18:1–18:18, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ITP.2023.18.
- 12 Johannes Hölzl and Armin Heller. Three chapters of measure theory in Isabelle/HOL. In Marko van Eekelen, Herman Geuvers, Julien Schmaltz, and Freek Wiedijk, editors, *Interactive Theorem Proving*, pages 135–151, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.
- 13 Johannes Hölzl, Fabian Immler, and Brian Huffman. Type classes and filters for mathematical analysis in Isabelle/HOL. In *Proceedings of the 4th International Conference on Interactive Theorem Proving*, ITP 2013, pages 279–294, Berlin, Heidelberg, 2013. Springer-Verlag. doi:10.1007/978-3-642-39634-2_21.
- 14 Alexander S. Kechris. *Classical Descriptive Set Theory*. Graduate Texts in Mathematics. Springer New York, 1995. doi:10.1007/978-1-4612-4190-4.
- 15 Kalle Kytölä. `LevyProkhorovMetric.lean`. <https://github.com/leanprover-community/mathlib4/blob/master/Mathlib/MeasureTheory/Measure/LevyProkhorovMetric.lean>. Accessed May 27th 2024.
- 16 Holden Lee. Vector spaces. *Archive of Formal Proofs*, August 2014. , Formal proof development. URL: <https://isa-afp.org/entries/VectorSpace.html>.
- 17 Paul Lévy. *Théorie de l’addition des variables aléatoires*. Gauthier-Villars, Paris, 1937.
- 18 Keiko Narita, Kazuhisa Nakasho, and Yasunari Shidama. F. Riesz theorem. *Formalized Mathematics*, 25(3):179–184, 2017. doi:10.1515/forma-2017-0017.
- 19 Anthony Narkawicz. A formal proof of the Riesz representation theorem. *Journal of Formalized Reasoning*, 4(1):1–24, January 2011. doi:10.6092/issn.1972-5787/1952.

- 20 Lawrence C. Paulson. Porting the HOL Light metric space library. https://lawrencecpaulson.github.io/2023/07/12/Metric_spaces.html. Accessed: December 31, 2023.
- 21 Yuri Vasilyevich Prokhorov. Convergence of random processes and limit theorems in probability theory. *Theory of Probability and Its Applications*, 1(2):157–214, 1956. doi:10.1137/1101016.
- 22 Walter Rudin. *Real and Complex Analysis, 3rd Ed.* McGraw-Hill, Inc., USA, 1987.
- 23 Mirah Shi. Nets and filters. <https://www.uvm.edu/~smillere/TProjects/MShi20s.pdf>, 2020. Accessed November 17th 2023.
- 24 Shashi Mohan Srivastava. *A Course on Borel Sets.* Springer, 1998. doi:10.1007/b98956.
- 25 The mathlib Community. The Lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020*, pages 367–381. Association for Computing Machinery, 2020. doi:10.1145/3372885.3373824.
- 26 René Thiemann and Akihisa Yamada. Matrices, jordan normal forms, and spectral radius theory. *Archive of Formal Proofs*, August 2015. , Formal proof development. URL: https://isa-afp.org/entries/Jordan_Normal_Form.html.
- 27 Onno van Gaans. Probability measures on metric spaces. <https://www.math.leidenuniv.nl/~vangaans/janco11.pdf>. Accessed: February 29, 2024.
- 28 Cédric Villani. *Optimal Transport: Old and New.* Grundlehren der mathematischen Wissenschaften. Springer Berlin Heidelberg, 2008. doi:10.1007/978-3-540-71050-9.
- 29 Shaoyi Zhang. Existence and application of optimal markovian coupling with respect to non-negative lower semi-continuous functions. *Acta Mathematica Sinica*, 16(2):261–270, 2000. doi:10.1007/s101140000049.

Distributed Parallel Build for the Isabelle Archive of Formal Proofs

Fabian Huch 

Technische Universität München, Garching, Germany

Makarius Wenzel 

Augsburg, Germany

Abstract

Motivated by the continuously growing performance demands for the Isabelle Archive of Formal Proofs (AFP), we introduce distributed cluster computing to the Isabelle platform. Parallel build time on a single node has approached 4 h–8 h in recent years: by supporting multiple nodes, without shared memory nor shared file-systems, we target at a substantial speedup factor to get below the critical limit of 45 min total elapsed time. Our distributed build tool is part of the regular Isabelle distribution, but specifically adapted to cope with the structure of projects seen in the AFP.

In this work, we address two main challenges: (1) the distributed system architecture that has been implemented in Isabelle/Scala, and (2) the build schedule optimization problem for multi-threaded tasks on multiple compute nodes. We introduce a heuristic tuned to the typical AFP session structure, which can generate good schedules in a few seconds. We reached a total speedup factor of over 100, which is a milestone never before reached. Using this approach, we could build the Isabelle distribution in 8 min 10 s elapsed time, and the AFP in 35 min 40 s, or 1 h 59 min 13 s including very slow sessions.

2012 ACM Subject Classification Computer systems organization → Distributed architectures; Software and its engineering → Distributed systems organizing principles; Mathematics of computing → Discrete optimization

Keywords and phrases Interactive theorem proving, Isabelle, Archive of Formal Proofs, Theorem prover technology, Distributed computing, Schedule optimization

Digital Object Identifier 10.4230/LIPIcs.ITP.2024.22

Supplementary Material

Software (Experiments and Data): <https://github.com/Dacit/isabelle-distributed-build>
archived at `swh:1:dir:748bbe787e4431fca08e42ab4d8fd64a4a3e6970`

Funding *Fabian Huch*: Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under the National Research Data Infrastructure – NFDI 52/1 – 501930651.

Acknowledgements We are immensely grateful to Rainer Kolisch, who shaped the research on project scheduling in operations research with his seminal PSPLIB library, and to Maximilian Kolter, for contributing their invaluable expertise to this project.

1 Introduction

Motivation. The Isabelle proof assistant is accompanied by the Archive of Formal Proofs (AFP) [1], an online journal of formal content produced with Isabelle. The AFP is continuously checked against the underlying Isabelle version (official release or development snapshots).

The AFP was founded 20 years ago: the first entry is from 19-Mar-2004 [21], and in June 2024 there are ≈ 830 entries. The steady growth of AFP content challenges the underlying Isabelle platform, since proposed changes to Isabelle need to be pushed through all entries of the AFP. Such changes could affect interfaces of Isabelle/ML, fundamental syntax of the Isabelle/Pure framework, library content in Isabelle/HOL (e.g. term notation, default proof



© Fabian Huch and Makarius Wenzel;

licensed under Creative Commons License CC-BY 4.0

15th International Conference on Interactive Theorem Proving (ITP 2024).

Editors: Yves Bertot, Temur Kutsia, and Michael Norrish; Article No. 22; pp. 22:1–22:19

Leibniz International Proceedings in Informatics



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

rules). Often, the initial change idea fails: it requires iterated refinement, until it becomes acceptable to the AFP – not causing too many follow-up changes there.

To retain the very notion of *interactive* proof development and maintenance, AFP test runs need to be reasonably fast. The folklore of AFP maintenance tells us that 45 min is a critical limit of “waiting online” for results, e.g. while doing other maintenance work elsewhere. Most of its lifetime, though, AFP tests have required much longer than that: often a rather painful 2 h–4 h. To help a bit, there is a classification of some AFP entries as `slow` (sessions that may take ≈ 1 h CPU time on their own account), or `very_slow` (sessions that require many hours). These tagged groups are tested less frequently than the main bulk.

In recent years, Isabelle users have been so successful in producing AFP content that we are now at 4.4 million source lines,¹ and the build time has degraded to 4 h–8 h (on a high-end server machine with many cores). Thus it has become increasingly difficult to change Isabelle libraries significantly: known problems often remain open for a long time.

That means it is high time to catch up with Isabelle prover technology. This has already happened several times in the past, e.g. with the introduction of shared-memory parallelism in Isabelle/ML from 2009 to 2013 [29, 19, 30], and replacement of GNU `make` by the custom-made `isabelle build` tool (after 2012). The next stage is to support *distributed parallel build* directly in Isabelle, to regain the 45 min limit for AFP built time.

Problem. The overall task is to organize the Isabelle build process better, to achieve a significant speedup factor that accommodates interactive development and maintenance: there needs to be enough headroom for the coming years of AFP growth. One side-condition is that Isabelle/AFP developments can be very large and thus produce large build results that need to be handled efficiently. Other side-conditions are given by the underlying ML system of Isabelle, Poly/ML² by David Matthews, and its traditional concept of *persistent heap image* from old LISP times. That is a convenient programming model from the past, which allows a growing context of ML code and values produced incrementally, but there are built-in limitations. ML heaps can only be stacked-up linearly – there can be forks, but no joins (so the dependency graph of Isabelle sessions always is a tree). Thus, large developments may often contain redundantly processed theories, to simulate session merges.

Solutions. There are various axes to tackle the problem, and several solutions may be combined eventually. At first we distinguish *internal organization* about how Isabelle/ML works vs. *external organization* of running ML sessions; the latter is done in Isabelle/Scala.

The past 15 years have seen many internal improvements of Isabelle/ML, notably shared-memory multiprocessing (e.g. yielding a solid factor of ≈ 5 –6 on 8 cores), but also runtime sharing of ML values. Those have already provided significant improvements to scalability, but also cause quite some complexity in the Poly/ML platform. Without exceedingly difficult engineering, significant gains cannot to be expected in this area, according to collected timings [16].

An external approach is to reorganize Isabelle session structure on the spot. The static description (via `ROOT` files) consists of session name, parent, used theories, and optionally re-used theories from other sessions. There is no need to build ML heaps precisely around this structure. The Isabelle/jEdit Prover IDE already allows to build a custom heap image, for the true requirements of the edited session (via `isabelle jedit -A -R`). The build tool

¹ <https://www.isa-afp.org/statistics>

² <https://www.polyml.org>

could do likewise, and load common library theories into one big session used as prerequisite for AFP, e.g. the union of `HOL-Library`, `HOL-Analysis`, and `HOL-Algebra`.

A less conservative variation is to impose the `skip_proofs` option on theories that are re-used from other sessions. This would imitate the theory (object) files known from the HOL4 prover, which is also based on Poly/ML. In any case, the expected speedup factor is limited to the currently observed waste factor of ≈ 1.75 in the AFP.

A more promising approach is to allow multiple build hosts, to open the game of distributed parallel computing: compute clusters may easily consist of hundreds or thousands of nodes, so a large speedup factor can be expected for the future. This merely requires to manage distributed builds properly, which is the plan of this work: for small clusters of 10–20 nodes.

Contribution. We introduce Isabelle technology for distributed parallel build of the AFP, with only minimal requirements on system infrastructure, and support for heterogeneous hardware that is often found in scientific environments, consisting of fast and slow machines. We provide build schedule optimizations based on an initial set of empirical data, and evaluate the resulting schedules and system performance with two more data sets

2 Distributed Build Architecture

Our overall approach is that “*Isabelle is the build system of Isabelle*”. Regular users often identify “Isabelle” with the Isabelle/HOL object-logic, but the reality is more complex: Isabelle is a platform of different languages with different purposes (although with a common emphasis on λ -calculus and functional programming). Two Isabelle languages are particularly important: (1) Isabelle/ML as *mathematical programming language* to implement formal logic, specification mechanisms, proof tools etc. – that provides only minimal system operations, e.g. to access external files. (2) Isabelle/Scala as *functional language for systems programming*, with full-scale access to file systems, database engines (SQLite or PostgreSQL), external processes (GNU Bash), operating system tools (rsync, SSH, XZ, Zstd), TCP/IP services etc. Scala runs on the Java/VM and acts like an abstract operating-system for organization of Isabelle applications: it works uniformly on Linux, macOS, or Windows (with Cygwin).

So we shall rather identify “Isabelle” with the Isabelle/Scala environment for systems programming: this is where our main implementation happens.³ Thus we follow the tradition of the first (more modest) `isabelle build` tool from September 2012.⁴

2.1 Re-use the Slurm Cluster Workload Manager?

Quite a lot of systems for compute cluster management have appeared in the past 10–20 years. In the world of scientific computing, the most popular one is now Slurm [17]. Major centers for high-performance computing (HPC) often use Slurm as a “meta-operating system”, to manage queues of user jobs. A running job appears on several compute nodes at the same time. The user program sees a conventional Linux system with network access; usually there are additional means to communicate with sibling nodes via message passing (e.g. OpenMPI).

The HPC Linux Cluster of the Leibniz Supercomputing Centre (LRZ) is next door to TU München in Garching. Thus we were motivated to explore this infrastructure carefully,

³ <https://isabelle-dev.sketis.net/source/isabelle/browse/default/src/Pure/Build/;29f2b8ff84f3>

⁴ <https://isabelle-dev.sketis.net/source/isabelle/browse/default/src/Pure/System/build.scala;a1e2ba3c697>

with the help of a student project that produced an Isabelle workload manager based on Slurm [20], but running on our own Linux computers or virtual machines from the LRZ. The main lessons from this experiment are:

1. Slurm is primarily for Linux, and unavailable on macOS or Windows.⁵ This is bad, because macOS is important in the ITP community, and we intend to test AFP on it eventually.
2. Slurm set-up is rather complicated, requires special Unix permissions, and is not easy to extend. In particular, Slurm needs a custom authentication service (system daemon), demands uniform user and group IDs, and has static cluster configuration that makes it difficult to add new machines. This is in conflict with typical compute resources found in academic environments: often there is only non-root access and standard system software.
3. Slurm does provide mechanisms for planning and scheduling jobs, but these are insufficient for our purpose, as they cannot use as much of our domain-specific information: For instance, Isabelle sessions can be built in different configurations (regarding ML threads and heap size). With a prediction model for the required computation time and resources, a build schedule could be planned in advance and optimized accordingly.
4. Although Slurm is very popular, its feature set turned out insufficient to handle high-end workstations with different kinds of CPU cores (hot “P-cores” versus cool “E-cores”).

Hence, we decided not to use an off-the-shelf component for cluster management, and instead do it ourselves in Isabelle/Scala, with minimal requirements and better results.

2.2 System Design based on Isabelle/Scala + SSH + PostgreSQL

Our design makes only minimal assumptions about the system environment:

1. Uniform platform: All build machines (master and worker nodes) operate on the same platform, after an initial decision if that is Linux, macOS, or Windows – and Intel or ARM.
2. Repository clones: official Mercurial repositories of Isabelle⁶ and AFP⁷ are required at the root of the overall build process, but regular build hosts get sources via `rsync`.
3. Local file systems: Working directories reside on fast solid-state storage. Network shares are neither required, nor desired: too slow and difficult to use concurrently.
4. SSH: The master node can reach all worker nodes via `ssh`, to access files and to run processes. SSH users and home directories may vary, but the shell needs to be GNU `bash`.
5. PostgreSQL: There is a dedicated PostgreSQL database server⁸, which is accessible via LAN/WAN from all build nodes (using TCP/IP directly or tunneled through SSH).

SSH clients and servers are readily available on all platforms, usually from the single source of the OpenSSH project, which works reliably. In Isabelle/Scala, `ssh` works as stay-resident program: a single connection is multiplexed for many requests. The PostgreSQL database server is available in many forms on many systems, e.g. as a system package on Ubuntu Linux that requires approx. 10 min to set-up – we are using version 14, e.g. on Ubuntu 22.04

⁵ <https://slurm.schedmd.com/platforms.html>

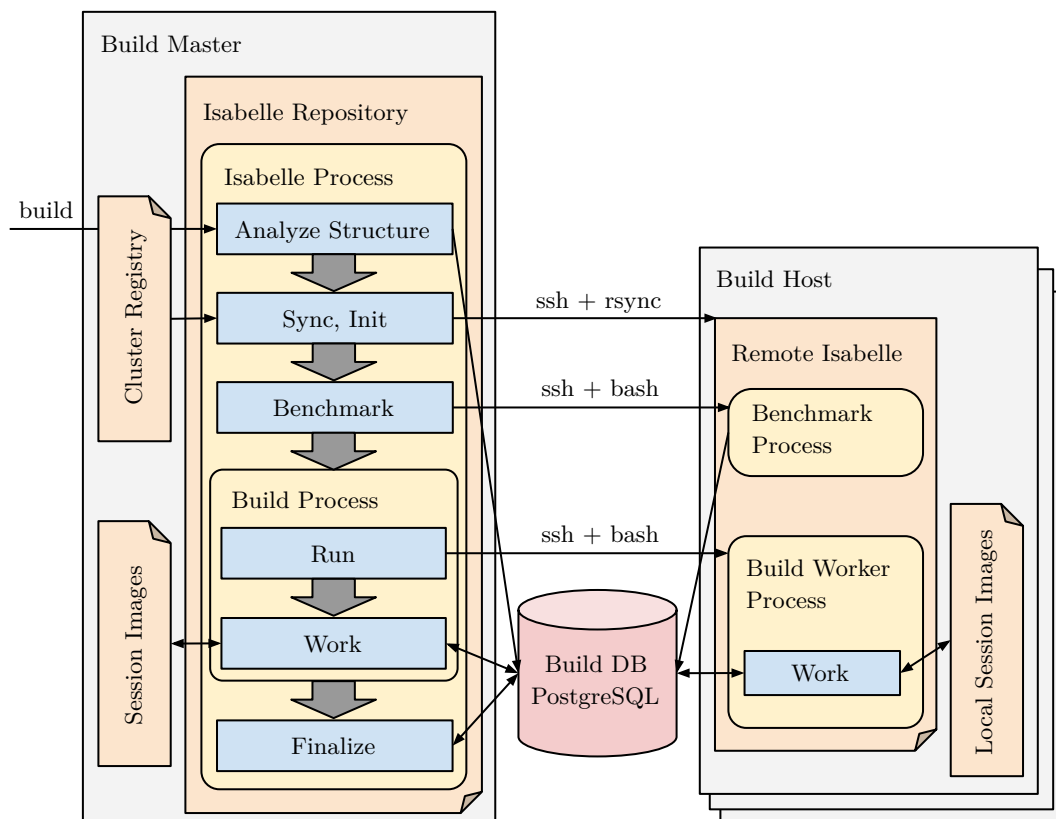
⁶ <https://isabelle.in.tum.de/repos/isabelle>

⁷ <https://foss.heptapod.net/isa-afp/afp-devel>

⁸ <https://www.postgresql.org>

LTS. Accessing the database server through SSH tunneling (via Isabelle/Scala) avoids the need to configure extra SSL certificates. The `SQL` module in Isabelle/Scala provides a simple programming interface, based on PostgreSQL JDBC for Java. All other system dependencies are bundled with the Isabelle distribution, notably Scala 3 and OpenJDK 21. This now also includes the `rsync` tool, in just one version for all platforms, so that it can speak to itself reliably over a cluster of dissimilar machines.

These parts are fit together as the distributed parallel build infrastructure for Isabelle as shown in Figure 1. The cluster topology is given on the command-line via “`isabelle build -H hosts:options ...`”, based on predefined hosts and host groups from a global registry file (in TOML format). The master oversees the build process, using database tables that are shared with all workers. Workers may in principle join and leave the build process freely, but presently we use a fixed arrangement determined at the build start. The master and workers all run the same Isabelle/Scala program, with a conventional programming model of *critical regions* and *functional update of shared state* (similar to *synchronized variables* in Isabelle/ML and Isabelle/Scala). The distribution of state information happens via the database server, using our own implementation of transactions with table locking, and efficient propagation of incremental updates (that works like a version control system, using a relational `OUTER JOIN` of recent updates against latest data). Since full transaction locking is rather costly (minimum ≈ 100 ms, median ≈ 1 s, sometimes much longer), we use the special PostgreSQL commands `LISTEN` and `NOTIFY` for asynchronous IPC messages. This reduces the frequency of state synchronization for critical regions: the master signals when relevant state updates have happened, so workers run silently most of the time without polling the database.



■ **Figure 1** Overview of the distributed Isabelle build system.

Seen from the master build host, the main build phases of Figure 1 are as follows:

1. *Analyze* the source structure of selected sessions (as defined in ROOT files), including full theory dependencies (given by theory headers). The resulting data is transferred to the database and can later be retrieved by workers. File names are stored with symbolic paths, relative to the root directories of Isabelle or the AFP.
2. *Synchronize* the master Isabelle distribution to all workers via `ssh` and `rsync`. This includes Scala/Java build artifacts and thus saves 1.5 min to initialize remote clones. The command-line `isabelle build -A`: treats the AFP directory structure analogously, but other project directories must already be present on each worker node.
3. *Initialize* the worker Isabelle distributions, by letting each node download required system components from the Isabelle components store, and finally ensure that all Scala/Java modules are up-to-date. Afterwards, workers are ready to run.
4. Run a short *benchmark* session by each worker, if explicit schedules should be used. This helps to predict overall run-times as discussed below.
5. *Run* all workers and let them *work* on the build autonomously, using queue and schedule information from the database. The master could participate as another worker, but is disabled by default, as it often resides on a slow workstation.
6. *Finalize* the build when all pending sessions are finished. This includes HTML and \LaTeX presentation for all sessions, based on markup information found in session databases.

For the AFP, being more a scientific journal than a library of code, the presentation is the main outcome of a successful build. In contrast, the result of running an Isabelle/ML session is an opaque *heap image*, which is potentially large. So we compress heap images with `Zstd` (portably in Isabelle/Scala) and store them in separate tables of the database as raw blobs (avoiding extra compression by PostgreSQL). Remote workers check and restore required heaps from the shared store: local file systems act like a cache for heaps served by the database.

Care is required to manage ML heaps in a scalable manner. For example, the HOL image is at ≈ 200 MiB uncompressed, ≈ 50 MiB compressed, with 0.9s/0.3s compression/uncompression time (on a fast machine). To give a sense of scale: Committing the compressed blob requires roughly 1.5s. Most Isabelle/AFP session images are much smaller, but can sometimes approach 1 GB. To allow very large applications in the future and circumvent the PostgreSQL limit of 2 GB for blobs, we store heap images as slices of ≈ 50 MiB.

After several rounds of performance tuning of our Isabelle/Scala implementation, the PostgreSQL engine works fairly well, both for small-scale synchronized program state, and large-scale storage of session images. Thus we can work without network shares, and really treat worker hosts as independent machines (apart from user-level SSH connections).

2.3 Prediction of Run-time per Configuration

A critical component for the schedule generation is an accurate estimation of run-time for a given configuration. Generally, there is little hope to estimate that without actually running the session: For instance, the AFP theory `Lorenz_C1` contains a single innocent-looking locale interpretation (three lines), whose run-time is about 50 h CPU-time on average test hardware. Still, once the run-time is known for some configuration, we can predict others. We use the `build_log` database of Isabelle to get data from historic builds. Our estimation is based on two main principles:

1. Given a job and number of threads, we estimate how the time changes across different build hosts via a factor derived from an initial benchmark. The benchmark session runs in single-threaded mode, it should be neither too short nor too long, its content should

rarely change, and it must be representative. Figure 2 shows the mean square error (MSE) of all sessions when used as individual benchmarks to scale the run-time of other sessions across hosts. There is a distinct separation in the build time since most sessions require HOL, which accumulates to 155 s already. FOLP-ex stands out as the session with lowest MSE amongst the faster sessions, and only takes about 15 s. All sessions with lower MSE are much slower. Further experiments with the initial fragment of HOL as benchmark all exhibited a far higher MSE, and the sessions that stick out have in common that they manipulate large terms. A likely explanation is that the typical workload seen in most sessions consists of such term manipulations, and the Isabelle/ML loading in Pure and bootstrapping of HOL is not representative.

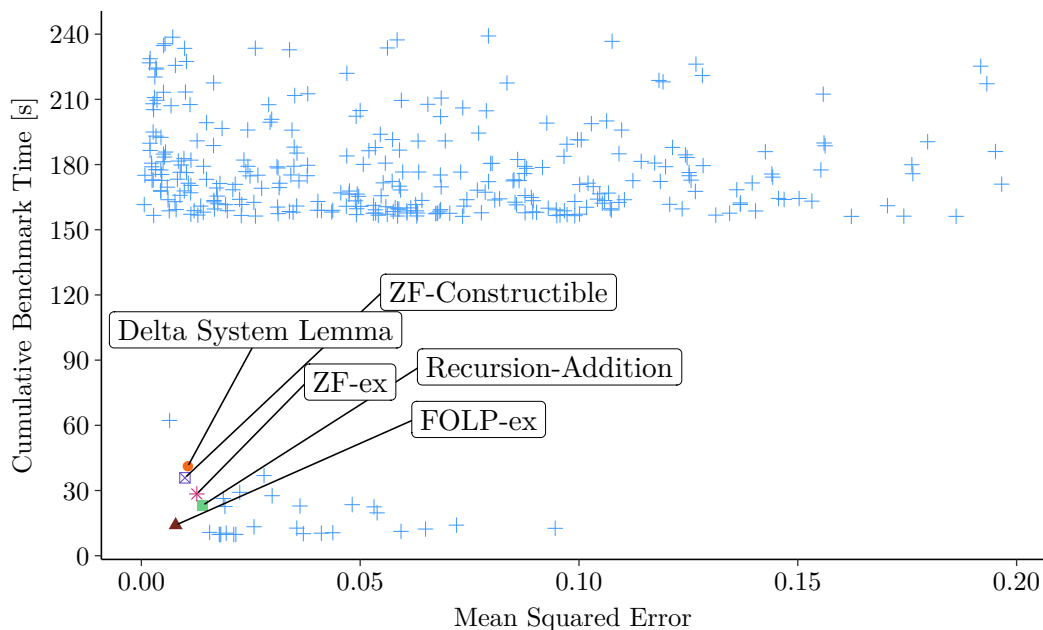
2. If a job has been run with a different number of threads on the same host, we can interpolate the curve to the unknown point. However, the speedup has lots of variation, as illustrated by the speedup curve for HOL-Analysis shown in Figure 3. Even though the speedup factor varies, it always increases up to an optimum at 8–16 threads and then decreases again due to multithreading inefficiencies. Although one could try to train a sophisticated prediction model for the speedup, in practice only very few configurations are known most of the time and robustness is more important than accuracy. Hence we make use of the only underlying mechanic that we can safely assume: According to Amdahl’s law [14], in any task only a fraction p of the total work T is parallelizable, so the total time for n threads is $\frac{p}{n}T + (1-p)T$. Thus, we select the prefix for which the speedup is monotone, estimate p by $pT = \frac{nm(T_n - T_m)}{m - n}$ for any two present timings T_n and T_m (for n and m threads, respectively), and interpolate according to that. If there is no such prefix, we use a simple rectified linear interpolation. This approach decreases the mean relative error (MRE) from 0.52 (linear only) to 0.42.

Our estimation then works as follows: If there is no historic build of a given job, we either use the session time-out if it is defined, or we take the average time of all jobs. If there are no other jobs to approximate from, we use dummy data to bootstrap the process. Otherwise, if there is data for the right job and threads, we just scale it to the right host. Should there be no data for the given number of threads, we try to interpolate. If there are not enough data points on the given host to do that, we interpolate on all other hosts individually and scale the results up to the current host. In case there is no host with enough data points, we unify the data and scale the individual points up to the current host, and then interpolate. Finally, if there is only one data point, we use the global speedup curve for the interpolation. Figure 4 shows boxplots of the relative error in the different scenarios outlined above. Unsurprisingly, the error is lower in the scenarios where less prediction is necessary.

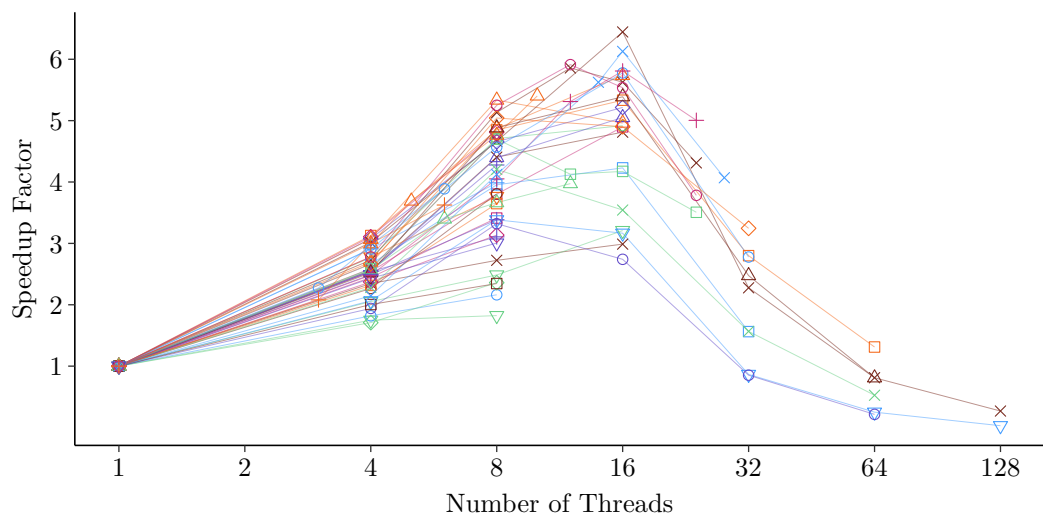
In the final estimation, we are interested in producing a result that does not lead to underestimations which could be exploited by the schedule optimization – especially since a delayed task can cause problems for the whole schedule, but faster completion is not a problem. Hence, we increase the estimations slightly depending on the scenario.

3 The Scheduling Problem

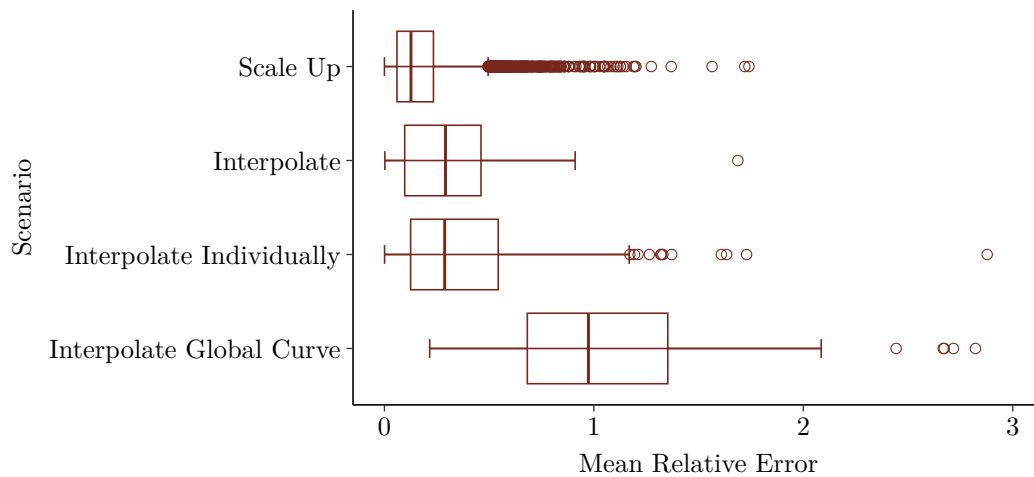
The classic `isabelle build` tool immediately starts new session jobs whenever there is free capacity. If there are multiple sessions to select from, the session with the longest build time in its path of successors is chosen. To estimate the build time, the timing of the previous build is used, if available. Otherwise the static session timeout serves as a rough approximation of actual runtime. The configuration of the number of threads for the Isabelle/ML process is a static parameter, usually given on the command-line.



■ **Figure 2** MSE vs. cumulative benchmark time (single-threaded build of benchmark session, after parallel build of its requirements), cut off at 240s or MSE of 0.20. The MSE refers to the estimation error when scaling build times across different hosts, using on relative benchmark time as factor. Only sessions from the preliminary data set with at least 10 different data points are shown.



■ **Figure 3** Number of threads vs. speedup factor for **HOL-Analysis** on various CPUs, using data from [16]. The speedup is the elapsed time, divided by single-threaded run-time on the same machine.



■ **Figure 4** Box plots for MRE of interpolation in different scenarios on preliminary data. Whiskers mark 1.5 inter-quartile range.

This now works differently: the command-line option “`-o build_engine=...`” tells which implementation of build scheduling should be used. The default is merely an upgrade of the old approach, to work with multiple build hosts. In addition, there is now the “`build_schedule`” engine that implements advanced scheduling as described in the following.

While the old strategy was sufficient for a low degree of session-parallelism, it is far from optimal for parallel builds in a large heterogeneous cluster, where the compute power may vary greatly across machines. The two different levels of parallelism, as well as non-uniform memory access and different kinds of processor cores found in current CPUs, make for a very uneven scheduling problem. Hence, basic approaches to multiprocessor scheduling are not applicable. Instead, the scheduling problem at hand has been studied as the multi-mode resource-constrained project scheduling problem (MRCPSP) since the late 1960s [7]. Some variations of the exact formulation of this problem exist in the literature. Typically, it consists of finding a schedule for activities \mathcal{A} in different modes \mathcal{M} , where each configuration (pair of activity and mode) takes $T: \mathcal{A} \times \mathcal{M} \rightarrow \mathbb{N}$ time, such that the overall *makespan* (i.e., the time it takes to process all activities once in any mode) is minimized. The problem is subject to precedence constraints according to some acyclic precedence relation $\prec: \mathcal{A} \times \mathcal{A} \rightarrow \text{bool}$, and there are constrained (renewable and non-renewable) resources \mathcal{R} . Each resource has an associated capacity $C: \mathcal{R} \rightarrow \mathbb{N}$, and a certain amount is required to run a configuration according to some cost function $R: \mathcal{A} \times \mathcal{M} \rightarrow \mathcal{R} \rightarrow \mathbb{N}$.

In our case, we need to schedule the build of a selection of sessions as activities, adhering to the dependency graph that induces the precedence relation. Each session can be run with different arguments such as the number of ML threads or the maximum ML heap size, though we only consider threads in this work. The build hosts (machines) define our resources: Each host has a number of CPU cores, which is a capacity that limits the sum of threads used. A second capacity is given by the maximum number of sessions that may run in parallel on each host. Finally, the run-time is determined by the session, build host, and number of threads as discussed in Section 2.3. Since in the MRCPSP formulation the run-time is independent of the resource used, we employ one mode per host and possible number of threads.

As we may not know the exact run-time and have to estimate it from historical data, we do introduce some uncertainty. Although stochastic approaches to the problem exist [26, 18], we work with the deterministic variant since it is simpler and more widely studied, and because the worst-case time is less of a concern – we are more interested in obtaining a solution quickly with good average time (which is a situation where this approach is appropriate [2]). Moreover, there is no need that the schedule stays fixed throughout the build: Should the actual times deviate (or a better solution be found due to the decreasing search space), the new schedule can be dynamically adopted. Hence, we re-compute the schedule periodically during the build.

3.1 Computational Complexity

Already the much simpler scheduling problem (considering identical processors instead of resources) is NP-complete [27]. Moreover, the MRCPSP (with resources and processors) remains strongly NP-hard even when restricting the problem simultaneously to one resource, uniform activity durations, a dependency graph that is a tree, and two processors (or three when the graph is omitted) [10]. It can also not be approximated by a constant factor in polynomial time even for a less general variant [9]. In our scheduling problem, in addition to the dependency graph being a tree, we can assume that T typically exhibits a monotone behavior: For all sessions, faster CPUs and more Isabelle/ML threads (up to a limit) usually correspond to less runtime. However, this does not improve complexity as the problem remains strongly NP-hard even with only two machines, a chain of dependencies, one single resource, as well as uniform mode, capacity, cost, time, and speed [4].

3.2 Schedule Generation Methods

Due to the practical importance in many fields and industries, a vast number of approaches exist to the problem (cf. [25] for a recent comprehensive resource). Exact solutions are often branch-and-cut methods which iteratively constrain the LP-relaxation of the mixed integer program for the optimal solution [33]. However, these integer programs explode in size: Due to the linear nature, one variable is required for every possible configuration of every activity at every time step. In another approach, Bofill et al. [5] use iterative SAT solving to find the lower bound for which an encoding of the problem is still satisfiable such that the satisfying assignments correspond to a schedule. Both this implementation as well as a recent implementation of the linear programming based-approach (as well as [12] and our own implementation of another mixed integer approach) could find the optimal solution for a small problem with 28 activities⁹, but found no reasonable bounds for larger problems.

Meta-heuristic optimization approaches such as genetic algorithms, local search, or simulated annealing (where neighbor solutions are constructed for an initial feasible solution to find optima) have become quite popular in the field [23]; these can find approximative solutions for larger problems. We observed that a local search implementation [11] and one of the original genetic algorithm approaches [13] found solutions for the small problem very fast, but struggled for our medium-sized problem (132 activities)¹⁰ due to technical limitations. Using OptaPlanner¹¹ (which combines several meta-heuristics), we could generate a good solution for this problem in short time, but did not find a solution for our large problem

⁹ Building all sessions in the Isabelle distribution excluding HOL on one machine, with 1 or 8 threads.

¹⁰ Building the whole Isabelle distribution of a cluster of 4 fast machines, with 1 or 8 threads.

¹¹ <https://www.optaplanner.org>

(923 activities)¹². We found that only some constraint programming solvers were able to tackle problems of this size. While the open-source Google OR-Tools¹³ could solve some of the large problem instances given enough time, IBM ILOG CPLEX¹⁴ excelled for this task, finding a reasonable solution for the large problem in a few minutes.

However, since CPLEX is proprietary software that cannot be bundled with Isabelle, we focus on heuristic-based *schedule generation schemes* (SGS) in the present work. These can also provide an initial solution to aid further optimization, e.g. by providing an upper bound on the makespan for the constraint programming formulation. SGS yield very fast solutions at the expense of solution quality by iteratively building a schedule, maintaining a decision set of activities whose precedence constraints are fulfilled. In the *serial* SGS, at every step a configuration is selected according to some *priority rule* (e.g., selecting the activity with the most successors in the fastest mode) and scheduled in the earliest possible time. In contrast, in every time step of the *parallel* SGS, a priority rule is employed to select from the set of currently feasible configurations until no more can be scheduled, at which point the scheme proceeds to the next time step. This latter approach is greedy since activities are scheduled immediately when they are feasible, which limits the search space, but also means that the optimal solution may not be found (independently of the priority rule).

3.3 Priority Rules for Isabelle builds

In the literature, a vast amount of priority rules have been explored, and ultimately it depends on the problem structure which rules perform best. Our exact problem structure can vary depending on which sessions need to run, and which hardware is available. Still, we can make some general observations that hold for AFP builds on a typical cluster:

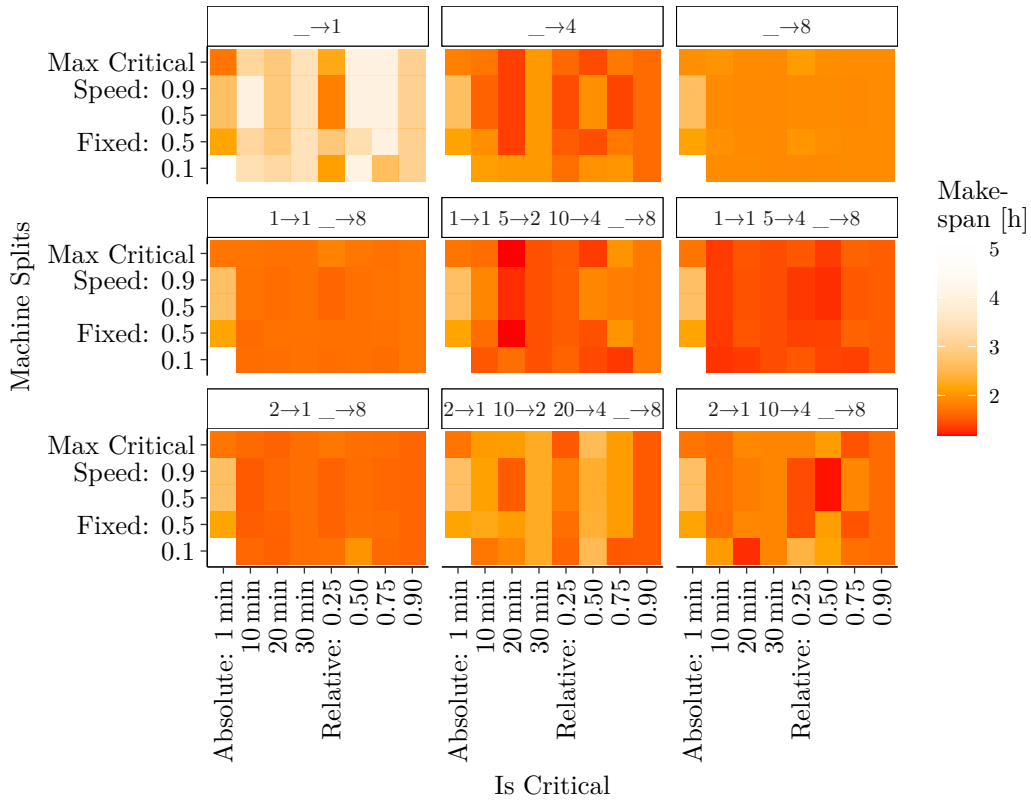
1. Session sizes and in-degrees in the dependency graph (the number of other uses of a session) weakly follow a scale-free distribution [15], i.e. there are few long-running or frequently-used sessions, and many sessions which are fast or without dependencies.
2. As machines in research environments often accumulate over the years, there will be relatively few of the fastest machine, but plenty of slower ones. Moreover, machines with many cores (server CPUs) typically have far lower clock speed than high-end consumer CPUs with fewer cores, so the more cores a CPU has the slower they are.
3. As the number of threads for a single session increases, the elapsed time decreases (up to some limit, typically 16 threads), but the parallel efficiency also degrades.

Hence, it is desirable to build sessions that greatly influence the overall makespan with the optimal amount of threads on the hardware with fastest individual cores, and sessions that have little to no impact in single-threaded mode on slow hardware. Due to the scale-free nature, we can hope that constructing a greedy solution that takes this into account yields a near-optimal solution. Our heuristics hence employ the notion of *critical* sessions, which are sessions that are on a path in the dependency graph whose accumulated time (for optimal amount of threads) is longer than some cut-off. The cut-off is taken either at some absolute value, or as a factor of the critical path through the dependency graph. A number of the fastest hosts is reserved for these critical sessions: Either one for every critical session that needs to be built in parallel, a fixed fraction of the fastest machines, or all machines faster than some cut-off. Finally, the uncritical sessions are run with a number of threads that increases for slower sessions (or is always constant).

¹² Building the AFP (without slow sessions) on an experimental cluster of 14 machines, with 1 or 8 threads.

¹³ <https://developers.google.com/optimization>

¹⁴ <https://www.ibm.com/products/ilog-cplex-optimization-studio>



■ **Figure 5** Solution quality (on preliminary data) for the three different parameters of our heuristic. The x -axis shows the criterion for a critical session, the y -axis how the available machines are split for critical/uncritical sessions, and the group label shows the threads parameter: The arrow notation maps run times less than the specified number of minutes on the left hand side to the number of threads on the right hand side. Darker is better (shorter makespan).

Figure 5 shows a comparison of the resulting makespan for an AFP build, depending on concrete values for the parameters explained above. The thread distribution is clearly the most important parameter for this build, and best results are generally achieved with 1 thread up to 1 min, then 4 threads up to 5 min, and 8 threads above that. Which sessions should be considered critical depends on the other parameters, but an absolute path length of 10 min to 20 min works well. The machine split only plays a minor role most of the time.

For the final heuristic, we additionally exploit that the number of sessions that may be scheduled on each host is typically much smaller than the number of CPU cores available (due to memory requirements), so we construct a parallel SGS where sessions are scheduled with a larger amount of threads than the minimum if there are unused cores. Additionally, if there is more capacity than sessions can use in parallel, we immediately schedule all sessions in their optimal mode. Finally, we sweep over the parameter space to find the best schedule, which can be done within a few seconds even for our largest problems.

4 Experimental Results

To empirically evaluate our schedules and strategies, we used three data sets with different hardware and configuration, as shown in Table 1: A preliminary data set used during development (cf. Section 3), one set collected on a heterogeneous cluster, and a final data set where we used the Isabelle2024 release on our current hardware.

■ **Table 1** Experimental setup for data sets used in evaluation. CPU slices have access to 10 cores.

Data Set	Preliminary	Heterogeneous	Release
Measure Period	Nov/Dec 2023	Mar 2024	Jun 2024
Isabelle Revision	various	08b83f91a1b2	Isabelle2024
Builds/Data Points (Sessions)	38/19 263	41/30 510	36/30 203
Cluster Hardware:			
Xeon Silver 4316 Slice, 44 GB DDR4	10	10	–
Intel Core i9-12900K, 64 GB DDR5	4	4	4
Intel Core i9-13900K, 128 GB DDR5	–	–	4

We compare four scenarios in the following: Building the Isabelle distribution (medium-sized problem in Section 3.2), the AFP (without `slow`, corresponds to the large problem), the slow AFP (including `slow`, but not `very_slow`), or the full AFP. In doing so, we want to answer the following research questions empirically:

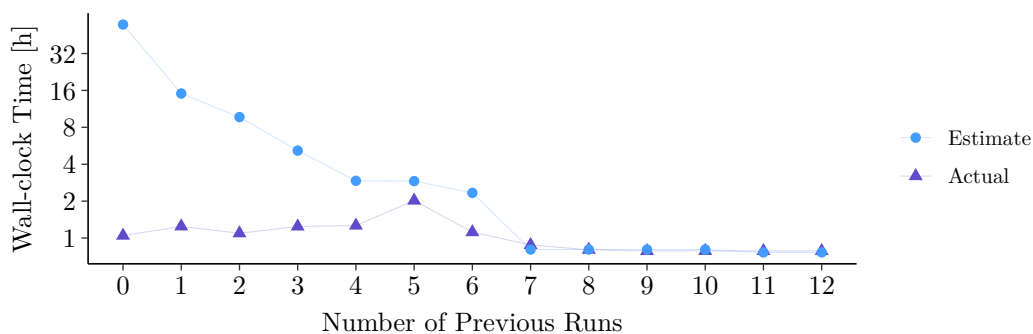
- RQ1: Which timings do we achieve, and when should one use explicit schedules?
- RQ2: How good is our heuristic compared to the best solver-generated solution?
- RQ3: Do the actual builds follow the generated schedule closely?

RQ1. To answer our first question, we compare the resulting build timings for different scheduling modes of the heterogeneous cluster in Table 2a, and the final results for the release version on our current hardware in Table 2b. On our current (homogeneous) cluster, the

■ **Table 2** Final build timings from heterogeneous and release data sets, with different build engines. The solver engine refers to a scheduled build with an initial schedule generated by CPLEX in 5 min.

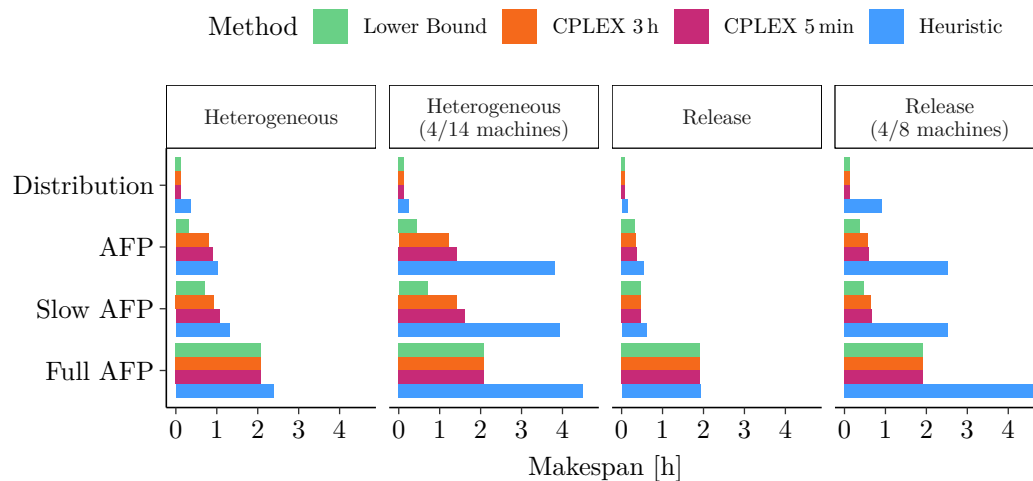
	(a) Elapsed time for heterogeneous data set, without benchmarks.			(b) Elapsed and CPU time for release data set, including benchmarks.			
	Classic Elapsed	Heuristic Elapsed	Solver Elapsed	Classic Elapsed CPU		Heuristic Elapsed CPU	
Distribution	0:16:23	0:09:16	0:09:29	0:08:10	2:49:51	0:09:12	3:58:49
AFP	1:00:25	0:43:11	0:56:12	0:42:16	37:22:37	0:35:40	64:57:11
Slow AFP	0:58:16	0:48:03	0:50:13	0:49:09	45:52:17	0:40:50	73:49:22
Full AFP	5:04:32	2:07:56	2:12:01	2:07:10	50:53:02	1:59:13	77:01:21

classic build mode performs best for shorter builds, finishing the distribution in 8 min 10 s. For longer builds, using the schedule generated by our heuristic is 7 min to 8 min faster than the classic build, finishing the AFP in 35 min 40 s, and 40 min 50 s including slow sessions. This corresponds to a factor of over 100 between CPU time and elapsed time, though the CPU time does increase with the amount of parallelism (since the parallelism consumes additional CPU time). These timings are all within our critical limits of 10 min for the distribution and 45 min for the AFP, which is a huge improvement. The difference between the strategies is much more pronounced on the large heterogeneous cluster, as important jobs can end up on a slow host if assigned blindly. Since this is the case for the classic build, finishing the full AFP (with very slow sessions) took more than 5 h, compared to about 2 h otherwise. Finally, starting with a solver-generated schedule (CPLEX in 5 min) resulted in slightly worse times compared to the heuristic, although the solver always generated a better initial schedule.



■ **Figure 6** Estimate and actual build time (log-scaled) for consecutive builds of the slow AFP, using the release version and our current hardware (starting with an initially empty build log database).

Even though the heuristic was faster than the classic build for the AFP, it requires bootstrapping by collecting data from several builds until the estimation can be accurate. Figure 6 shows how much the actual time deviates from the initially predicated time when starting with an empty database. After 7–8 builds, the estimate was very close to the actual build time, and both remained stable consequently. While the bootstrapping could still be improved by generating schedules that explore more modes, it is sufficient in automated build environments – for one-shot builds, the classic build engine is better suited anyway.

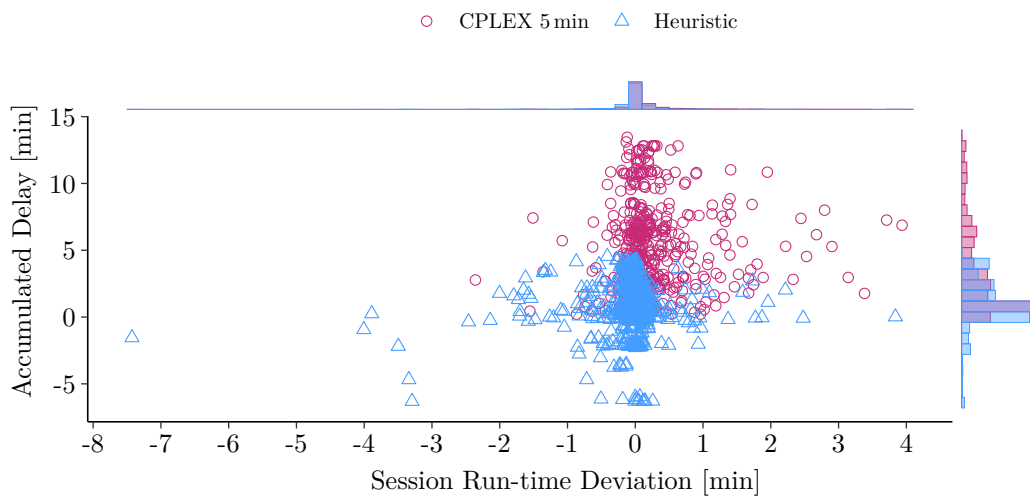


■ **Figure 7** Comparison of estimated makespan (and best computed lower bound) in different scenarios on heterogeneous and release data sets, using the full cluster or only the four fastest hosts.

RQ2. Although the true build duration is often better than initially estimated due to the optimization every 5 min (by default), computing a good schedule is still critical. Hence, for our second research question we compare the makespan of the initially generated schedules. We consider the schedule found by our heuristic as well as the best schedule and lower bound we could generate via CPLEX (on a slow machine) in 5 min, and 3 h, respectively. Figure 7 shows the results. For both data sets, the schedules are reasonably close to each other on

the full cluster (in the expected ordering). There is still a significant gap to the best lower bound for the harder scheduling problems (AFP and slow AFP). The solution quality of the heuristic drops off dramatically when scheduling only for the four fastest machines, where our assumption is violated that plenty of slower hardware would be available.

RQ3. For our last research question, we want to assess whether generated schedules can be followed closely in the actual build. To that end, we analyze deviation of session run-time and session delay (i.e., the difference between planned and actual completion) for builds of the AFP with pre-generated schedules. All optimizations that would change the schedule during the build are disabled for this experiment. Figure 8 shows the result. Overall, the individual run-time deviations were quite low. However, with the solver-generated schedule a few sessions took several minutes longer, delaying a significant chunk of sessions such that the schedule finished 13 min late. The heuristically generated schedule had less (and mostly negative) deviations and accumulated not much delay, finishing within a minute of the estimated time. While the solver-generated schedule might have accumulated more delay as it is more tightly packed, the mean deviation was also three times as high compared to the heuristic, indicating that the run-time estimation was worse. A likely reason for this is that the data was collected by running builds with the heuristic strategy, so there is more accurate data for its frequently selected configurations. This also explains why using the solver resulted in worse timings even though the initially estimated makespan was shorter.



■ **Figure 8** Accumulated delay and session run-time deviation for AFP builds with fixated schedules (release data), with histograms.

5 Conclusions

In this work, we introduced a distributed build system for Isabelle tailored to the requirements of the AFP. Based on the Isabelle/Scala framework in a standard repository clone, the system requires only SSH access, a fast local file system, and a single PostgreSQL database server – fully platform-independent. To utilize large heterogeneous clusters as well, we developed a sophisticated scheduling approach that incorporates build time estimation based

on previous timings. This involves finding a good solution to the computationally difficult multi-mode resource-constrained project scheduling problem in a few seconds, which remains NP-complete under major relaxations [10] (and does not admit constant-factor approximation in polynomial time [9]). After several rounds of tuning, the scheduling now works exceedingly well, and with our small clusters of 8–14 machines, we could achieve a speedup factor of over 100 vs. CPU time. This means that the AFP, which (through sheer size) had accumulated a multithreaded build time of 4 h–8 h on a single high-end server machine, can now be run in under 45 min again including slow sessions. Even with the `very_slow` group, the AFP now takes less than 2 h.

5.1 Related work

Any successful ITP system will eventually struggle with build times, as users are pushing more and more material into the libraries and archives. Coping with that requires technical efforts that usually remain unnoticed and unpublished. Subsequently we can only sketch important work for other proof assistants, with the focus on three kinds of parallelism: (1) multithreading within a single process, (2) multiprocessing on a single machine, (3) cluster computing on multiple machines.

The **Mizar** system and **Mizar Mathematical Library** have many similarities to Isabelle and AFP, concerning sizes of content, and the idea of a formally checked journal. The software technology is quite different, though: Mizar is implemented in Pascal and works like a multi-pass compiler on intermediate files. This does open possibilities for parallel processing: Urban [28] uses Perl and GNU `make -j` to manage the Mizar process for multiprocessing on a single machine. It might be worth revisiting that approach 10 years later, e.g. with the help of a distributed `make` tool. The resulting architecture would be unlike ours, where build data is stored in the database instead of a (shared) file-system.

Coq lacks a curated collection of applications, and it lacks a uniform build tool. Instead, every project needs to stand on its own for hosting and tooling. Reichel et al. [24] have scrutinized Coq build processes of projects found on public repositories: the motivation was to replay builds on intermediate versions, in order to collect data for automatic proof repair via machine-learning. As preliminary stage, huge efforts were required to “capture” Coq builds (they report that 68% of commits in open-source Coq projects could not be built without problems). This lack of uniform tooling for Coq makes difficult to foresee eventual moves towards cluster computing.

The OCaml runtime (on which Coq runs) did not support parallel execution of concurrent code until recently, so Coq parallelization had to rely on parallel processes without shared-memory, e.g. via parallel `make` for independent files. More involved efforts use local multiprocessing with message passing for proofs that are irrelevant (opaque), notably Barras et al. [3]: an explicit directed acyclic graph of Coq proofs tells which parts can be independently checked in parallel forks of `coqc`. Coq proof state information is passed along with the opaque proof task. A distributed version of that will probably require homogeneous software setup, but this has not been pursued so far. Instead, *regression proof selection* was introduced by Celik et al. [6] and later refined and parallelized by Palmkog et al. [22], where the fine-grained dependency graph of theorems and definitions is analyzed so that only proofs affected by changes need to be re-checked. However, they do not consider changes in the logic, which would be very hard to track via dependency graph. For instance, if any tactic changes, only checking the selected proofs would not be sufficient. Despite those technological efforts, changes are often not checked well enough.

The **Lean** prover [8] and its companion library **Mathlib** are closer to Isabelle and AFP than Coq, but Lean makes every effort to do everything in just one language. Consequently, the “Lean Make” or **Lake** tool has been implemented in the monadic functional programming language of Lean. Lake is a sophisticated build and package manager for Lean projects, and is part of the main code base of Lean 4.¹⁵ The corresponding Mathlib 4¹⁶ is organized as a Lake project: its current source size is 1.4 million lines or 62 MiB (the AFP has 4.4 million lines or 216 MiB). Proof-level parallelism is part of regular Lean, with built-in multithreading. Further scaling in Lake is unclear from skimming through the documentation: we did not find dedicated papers to explain its concepts.

In practice, users of Lean and Lake usually rely on cached object files for theories rather than building them afresh, and the build time of the continuous integration for isolated changes in Mathlib is usually less than 1 h. We could run the whole of Mathlib 4 in 28 min on one of our fastest machines, plus another 7 min for add-on tests (which in Isabelle/AFP would be part of regular sessions). As Mathlib grows further towards the current size of AFP, it will be interesting to see which improvements on Lean build management will emerge.

5.2 Future Work

We can imagine many directions for further improvement of the Isabelle build system. Some alternative solutions have already been sketched in the introduction: notably ad hoc **reorganization of session structure**, similar to existing options `isabelle jedit -A -R`. In fact, the Prover IDE opens many further questions of working with **quasi-interactive builds**: the user could repair failures incrementally in the IDE, while the build keeps running, and changed sources would be uploaded to the build database without stopping sessions that are unaffected. This ultimately means combining ideas from the PIDE editor model [31] with the build tool, to scale-up editing to the AFP, as was postulated naïvely in 2014 [32].

Revisiting the build schedule problem, we could support **near-optimal schedules** as follows. The heuristic developed above can find appropriate schedules in a very short time, but they are usually not optimal. In contrast, a state-of-the-art CP-solver can yield schedules with tangibly shorter makespan, but finding optimal solutions often takes much longer than our target of 45 min for the whole build. We could do the scheduling occasionally (e.g. once per week), and re-use the result for subsequent versions of the sources. With an optimal solution generated offline, and only small ongoing changes of the AFP, we can expect even better quantitative results.

Looking further at the big picture, an important question is how to use compute cluster resources efficiently within a typical research group, with several **independent builds** potentially by **multiple users**. This could mean to have different categories and priorities of builds, e.g. *responsive* ones for quasi-interactive maintenance via the Prover IDE front-end vs. more traditional *batch builds* for continuous integration scenarios (occasional updates by AFP authors). Our cluster resources could be maintained in a global database, and Isabelle/Scala build process would work with this information to re-arrange schedules on the spot. Ultimately, we would like to regain good reactivity for interactive work, and avoid disappointing “CI-clouds” where the queue-time is often longer than the actual build time.

¹⁵ <https://github.com/leanprover/lean4/blob/master/src/lake/README.md>

¹⁶ <https://github.com/leanprover-community/mathlib4>

References

- 1 Archive of Formal Proofs (AFP). ISSN 2150-914x. URL: <https://www.isa-afp.org/about>.
- 2 Francisco Ballestín. When it is worthwhile to work with the stochastic RCPSP?, 2007. doi:10.1007/s10951-007-0012-1.
- 3 Bruno Barras, Carst Tankink, and Enrico Tassi. Asynchronous processing of Coq documents: From the kernel up to the user interface. In *Interactive Theorem Proving (ITP 2015)*, volume 9236 of *LNCS*. Springer, 2015. doi:10.1007/978-3-319-22102-1_4.
- 4 J. Blazewicz, J. K. Lenstra, A. Kan, and H. G. Rinnooy. Scheduling subject to resource constraints: classification and complexity. *Discrete Applied Mathematics*, 5, 1983. doi:10.1016/0166-218X(83)90012-4.
- 5 Miquel Bofill, Jordi Coll, Josep Suy, and Mateu Villaret. Solving the Multi-Mode Resource-Constrained Project Scheduling Problem with SMT. In *International Conference on Tools with Artificial Intelligence (ICTAI 2016)*. IEEE, 2016. doi:10.1109/ICTAI.2016.0045.
- 6 Ahmet Celik, Karl Palmskog, and Milos Gligoric. ICoq: Regression proof selection for large-scale verification projects. In *Automated Software Engineering (ASE 2017)*. IEEE, 2017. doi:10.1109/ASE.2017.8115630.
- 7 Edward Wilson Davis. *An exact algorithm for the multiple constrained-resource project scheduling problem*. Yale University, 1968.
- 8 Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The Lean theorem prover (system description). In *Conference on Automated Deduction (CADE 2015)*, volume 9195 of *LNCS*. Springer, 2015. doi:10.1007/978-3-319-21401-6_26.
- 9 Evgeny R. Gafarov, Alexander A. Lazarev, and Frank Werner. Approximability results for the resource-constrained project scheduling problem with a single type of resources. *Annals of Operations Research*, 213, 2014. doi:10.1007/s10479-012-1106-5.
- 10 M. R. Garey and D. S. Johnson. Complexity Results for Multiprocessor Scheduling under Resource Constraints. *SIAM Journal on Computing*, 4, 1975. doi:10.1137/0204035.
- 11 Martin Josef Geiger. A multi-threaded local search algorithm and computer implementation for the multi-mode, resource-constrained multi-project scheduling problem. *European Journal of Operational Research*, 256, 2017. doi:10.1016/j.ejor.2016.07.024.
- 12 Daniel Geiss. Optimal Build Strategies for Isabelle. Technical report, Technische Universität München, 2023.
- 13 Sönke Hartmann. Project Scheduling with Multiple Modes: A Genetic Algorithm. *Annals of Operations Research*, 102, 2001. doi:10.1023/A:1010902015091.
- 14 Mark D. Hill and Michael R. Marty. Amdahl's Law in the Multicore Era. *Computer*, 41, 2008. doi:10.1109/MC.2008.209.
- 15 Fabian Huch. Formal Entity Graphs as Complex Networks: Assessing Centrality Metrics of the Archive of Formal Proofs. In *Conference on Intelligent Computer Mathematics (CICM 2022)*. Springer, 2022. doi:10.1007/978-3-031-16681-5_10.
- 16 Fabian Huch and Vincent Bode. The Isabelle Community Benchmark. In *Practical Aspects of Automated Reasoning (PAAR 2022)*, volume Vol-3201. CEUR-WS, 2022. doi:10.48550/arXiv.2209.13894.
- 17 Morris A. Jette and Tim Wickberg. Architecture of the Slurm Workload Manager. In *Job Scheduling Strategies for Parallel Processing (JSSPP 2023)*, volume 14283 of *LNCS*. Springer, 2023. doi:10.1007/978-3-031-43943-8_1.
- 18 Brian Daniel Keller. *Models and methods for multiple resource constrained job scheduling under uncertainty*. PhD thesis, University of Arizona, 2009.
- 19 David Matthews and Makarius Wenzel. Efficient parallel programming in Poly/ML and Isabelle/ML. In *Declarative Aspects of Multicore Programming (DAMP 2010)*. ACM, 2010. doi:10.1145/1708046.1708058.
- 20 Mokhtar Riahi Mohamed. Development of a Distributed Build Platform for Isabelle. Technical report, Technische Universität München, 2022.

- 21 Tobias Nipkow and Cornelia Pusch. AVL Trees. *Archive of Formal Proofs*, 2004. , Formal proof development. URL: <https://isa-afp.org/entries/AVL-Trees.html>.
- 22 Karl Palmkog, Ahmet Celik, and Milos Gligoric. PiCoq: Parallel regression proving for large-scale verification projects. In *International Symposium on Software Testing and Analysis (ISSTA 2018)*, volume 12. ACM, 2018. doi:10.1145/3213846.3213877.
- 23 Robert Pellerin, Nathalie Perrier, and François Berthaut. A survey of hybrid metaheuristics for the resource-constrained project scheduling problem, 2020. doi:10.1016/j.ejor.2019.01.063.
- 24 Tom Reichel, R. Wesley Henderson, Andrew Touchet, Andrew Gardner, and Talia Ringer. Proof Repair Infrastructure for Supervised Models: Building a Large Proof Repair Dataset. In *Interactive Theorem Proving (ITP 2023)*, volume 268 of *LIPICs*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPICs.ITP.2023.26.
- 25 Christoph Schwindt and Jürgen Zimmermann. *Handbook on project management and scheduling*, volume 1. Springer, 2015. doi:10.1007/978-3-319-05443-8.
- 26 Frederik Stork. *Stochastic Resource-Constrained Project Scheduling*. PhD thesis, Technische Universität Berlin, 2001.
- 27 Jeffery David Ullman. Polynomial complete scheduling problems. In *Proceedings of the Fourth ACM Symposium on Operating System Principles*, volume 7 of *SIGOPS*. ACM, 1973. doi:10.1145/800009.808055.
- 28 Josef Urban. Parallelizing Mizar. In *Trends in Contemporary Computer Science*. Bialystok University of Technology Publishing Office, 2014. doi:arXiv:1206.0141v2.
- 29 Makarius Wenzel. Parallel Proof Checking in Isabelle/Isar. In *Programming Languages for Mechanized Mathematics Systems (PLMMS 2009)*, 2009. URL: <https://files.sketis.net/papers/parallel-isabelle.pdf>.
- 30 Makarius Wenzel. Shared-Memory Multiprocessing for Interactive Theorem Proving. In *Interactive Theorem Proving (ITP 2013)*, LNCS. Springer, 2013. doi:10.1007/978-3-642-39634-2_30.
- 31 Makarius Wenzel. Asynchronous User Interaction and Tool Integration in Isabelle/PIDE. In *Interactive Theorem Proving (ITP 2014)*, LNCS. Springer, 2014. doi:10.1007/978-3-319-08970-6_33.
- 32 Makarius Wenzel. System description: Isabelle/jEdit in 2014. In *User Interfaces for Theorem Provers (UITP 2014)*, volume 167 of *EPTCS*, 2014. doi:10.4204/EPTCS.167.10.
- 33 Guidong Zhu, Jonathan F. Bard, and Gang Yu. A branch-and-cut procedure for the multimode resource-constrained project-scheduling problem. *Journal on Computing*, 18, 2006. doi:10.1287/ijoc.1040.0121.

A Generalised Union of Rely–Guarantee and Separation Logic Using Permission Algebras

Vincent Jackson  

The University of Melbourne, Australia

Toby Murray  

The University of Melbourne, Australia

Christine Rizkallah  

The University of Melbourne, Australia

Abstract

This paper describes GenRGSep, an Isabelle/HOL library for the development of RGSep logics using a general algebraic state model. In particular, we develop an algebraic state models based on resource algebras that assume neither the presence of unit resources or the cancellativity law. If a new resource model is required, its components need only be proven an instance of a permission algebra, and then they can be composed together using tuples and functions.

The proof of soundness is performed by Vafeiadis’ operational soundness method. This method was originally formulated with respect to a concrete heap model. This paper adapts it to account for the absence of both units as well as the cancellativity law.

2012 ACM Subject Classification Theory of computation → Logic and verification; Theory of computation → Concurrency; Theory of computation → Separation logic

Keywords and phrases verification, concurrency, rely-guarantee, separation logic, resource algebras

Digital Object Identifier 10.4230/LIPIcs.ITP.2024.23

Supplementary Material

Software (Mechanised Proof): <https://github.com/vjackson725/GeneralRGSep/tree/itp24> [22]
archived at [swh:1:dir:e759950d2ebd7571c86913f8296dfb29aa24a108](https://swh.1:dir:e759950d2ebd7571c86913f8296dfb29aa24a108)

1 Introduction

This paper describes GenRGSep, an Isabelle/HOL [35] library for the development of RGSep logics [39, 37], using a general algebraic state model. This library bases its state model on permission algebras, the most generic form of resource algebra [8]. Permission, multi-unit and single-unit separation algebras [12, 3] are developed as a type-class hierarchy that enables the integration of permissions and values into a common algebraic language. This allows for useful resource models to be developed from simple components and then automatically applied to RGSep. The soundness of GenRGSep has been formally verified by an operational soundness proof that generalises a method of Vafeiadis’ [38] to work without the cancellativity law.

This project is motivated, in part, by a desire to have a general separation logic framework for verifying concurrent code in Isabelle/HOL. There are several very general frameworks for the development of separation logics for the verification of concurrent programs in other theorem provers: for example, VST [2] and Iris [25]. There is, as yet, none in Isabelle/HOL. While this work is not yet as comprehensive as these projects, we hope it will provide a good foundation for the future development of such projects in Isabelle/HOL.

In order to achieve generality, we develop separation logic from resource algebras, an abstract algebraic model of resources [8]. A resource algebra is a specific sort of partial semigroup or monoid, which defines a model of separated resources. There are many variations,



© Vincent Jackson, Toby Murray, and Christine Rizkallah;
licensed under Creative Commons License CC-BY 4.0

15th International Conference on Interactive Theorem Proving (ITP 2024).

Editors: Yves Bertot, Temur Kutsia, and Michael Norrish; Article No. 23; pp. 23:1–23:16



Leibniz International Proceedings in Informatics

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

usually on which unit elements the algebra is guaranteed to have (no guarantee, some unit for every resource, or a universal unit), and whether the logic admits the cancellativity law. The resource algebra approach has been used in many projects [12, 26, 3, 28, 25, 29].

All Isabelle/HOL separation algebras up to this point assume the existence of resources that act like a unit for resource addition. Similar to VST [3], GenRGSep is based on a type-class hierarchy of resource algebras which includes permission algebras, that do not require units to exist. This approach treats permissions as first-class citizens that compositionally integrate with other resources and into larger structures.

The RGSep family of logics [39, 37] combines the rely–guarantee method [24, 23] and concurrent separation logic [31, 7]. The rely–guarantee method is a method of concurrent program verification that requires rely and guarantee relations for each proces. The rely relation establishes how the shared state can be changed by the environment (other processes), and the guarantee relation establishes how the current process can change the shared state. Concurrent separation logic is a method of concurrent program verification that requires all state to either be local: in which case it can be separated into pieces which are acted upon by parallel processes separately, or shared, in which case, it must obey a resource invariant. RGSep combines the benefits of both methods: the separate reasoning about local state of concurrent separation logic and the fine-grained concurrency of rely–guarantee.

This paper introduces GenRGSep, a generalisation of RGSep to non-cancellative resource algebras without units, and prove its soundness. Our paper is structured as follows: in Section 2, we review the construction of resource algebras and describe the particular issues we encountered in translating them to Isabelle/HOL. In Section 3, we describe our GenRGSep language, which in addition to standard programming constructs, includes external nondeterminism and non-deterministic **do** statements [19, 20], and the RGSep logic for it. We also review explicit stabilisation [37, 41], which simplifies reasoning about stability. In Section 4.2, we discuss the soundness proof for GenRGSep, using an extension of a method by Vafeiadis [38]. This method encounters some problems with the combination of non-cancellative resource models and external-nondeterminism, which we demonstrate how to address.

Contributions

The paper presents the following contributions:

1. an encoding of an Isabelle/HOL type-class hierarchy for permission and separation algebras that allows for the compositional construction of resource models;
2. the formalisation of the soundness of RGSep over general permission algebras in Isabelle/HOL; and
3. a re-examination of Vafeiadis’ operational soundness method, showing how to extend it to non-cancellative resource algebras.

2 Formalising the Foundations

We construct separation logic from the foundations of an algebraic model of separated resources; these are often called separation algebras or resource algebras [8, 3]. In particular, we define three structures as type-classes in Isabelle/HOL: permission algebras, multi-unit separation algebras, and (single-unit) separation algebras. We will refer to these collectively as *resource algebras*, and to the elements of these algebras as *resources*. The axioms for these structures are listed in Figure 1.

```

class perm-alg(#, +) =
  partial-add-assoc:      a # b → b # c → a # c → (a + b) + c = a + (b + c)
  partial-add-commute:    a # b → a + b = b + a
  disjoint-sym:           a # b → b # a
  disjoint-add-rightL:    b # c → a # b + c → a # b
  disjoint-add-right-commute: b # c → a # b + c → b # a + c
  positivity:             a # a' → b # b' → a + a' = b → b + b' = a → a = b

class multi-sep-alg(#, +, unitof) =
  perm-alg(#, +) +
  unitof-disjoint:      (unitof a) # a
  unitof-add:           (unitof a) # b → (unitof a) + b = b

class sep-alg(#, +, unitof, 0) =
  multi-sep-alg(#, +, unitof) +
  zero-disjoint :      0 # x
  zero-unit :         0 + x = x

class cancel-perm-alg(#, +) =
  perm-alg(#, +) +
  cancel-right:       a # c → b # c → a + c = b + c → a = b

```

■ **Figure 1** Resource algebra axioms.

2.1 Resource Algebras

A *permission algebra* (**perm-alg**), is a partial commutative semigroup, where $+$ is the semigroup operator and $\#$ (disjoint) specifies when $+$ is defined. The $+$ operator and $\#$ obey a number of laws, namely: the disjointness relation is commutative, the disjoint parts of a resource remain disjoint to (other) resources disjoint to the whole (that is, $y \# z$ and $x \# y + z$ implies $x \# y$), and the disjoint parts of a resource remain disjoint to the other parts of that resource when added to (another) resource disjoint from the whole (that is, $y \# z$ and $x \# y + z$ implies $x + y \# z$). In addition, resources that contain each other as parts are equal (positivity).

A *multi-unit separation algebra* (**multi-sep-alg**) is a permission algebra with an additional operation $\text{unitof} : \alpha \Rightarrow \alpha$, which produces the unit of the given resource of the algebra. A *separation algebra* (**sep-alg**) is a multi-unit separation algebra with the single unit, 0.

Resources form an order: a resource is strictly less than another (\prec) if they are not equal and there is some resource that adds to the first to make the second ($a \neq b \wedge (\exists c. a + c = b)$). A resource is less than or equal to another (\preceq) if they are equal or there is some third resource that adds to the first to make the second ($a = b \vee (\exists c. a + c = b)$). These definitions form an order, but this order is not necessarily Isabelle/HOL's standard order instance. Note that the order is anti-symmetric by virtue of the law of positivity. Note also that, in a multi-unit separation algebra, we have that $a \preceq b \iff (\exists c. a \# c \wedge a + c = b)$, because units are guaranteed to exist.

Permission algebras are useful for representing values with constraints on how those values may be used. The classic model is fractional permissions [6, 5] ($\mathbf{P}_{\mathbb{Q}}$ in Figure 2), where 1 represents the ability to change the value, and fractional quantities ($0 < x < 1$) represent only the ability to read the value. By placing this permission in a tuple with the discrete permission algebra ($\alpha \text{ discr}$, Figure 2), we obtain a model of these read-write values.

```

Fractional Permissions
typedef  $\mathbf{P}_{\mathbb{Q}}$  := { $x \in \mathbb{Q}. 0 < x \leq 1$ }
instance  $\mathbf{P}_{\mathbb{Q}}$  : perm-alg
   $a \# b$  :=  $a + b \leq 1$ 
   $a + b$  :=  $\min(a + b) 1$ 

Multiplicative Unit
datatype  $\mathbf{1} = \mathbf{1}$ 
instance  $\mathbf{1}$  : perm-alg
   $a \# b$  :=  $\perp$ 
   $a + b$  := undefined

Discrete Type
typedef  $\alpha$  discr := (UNIV :  $\alpha$  set)
instance  $\alpha$  discr : multi-sep-alg
   $a \# b$  :=  $a = b$ 
   $a + b$  :=  $a$ 
  unitof  $a$  :=  $a$ 

Functions
instance ( $\alpha \Rightarrow (\beta : \text{perm-alg})$ ) : perm-alg
   $f \# g$  :=  $\forall x. (f x) \# (g x)$ 
   $a + b$  :=  $\lambda x. (f x) + (g x)$ 
instance ( $\alpha \Rightarrow (\beta : \text{multi-sep-alg})$ ) : multi-sep-alg
  unitof  $f$  :=  $\lambda x. \text{unitof } (f x)$ 
instance ( $\alpha \Rightarrow (\beta : \text{sep-alg})$ ) : sep-alg
   $0$  :=  $\lambda x. 0$ 

Tuples
instance (( $\alpha : \text{perm-alg}$ )  $\times$  ( $\beta : \text{perm-alg}$ )) : perm-alg
   $(a_1, a_2) \# (b_1, b_2)$  :=  $(a_1 \# b_1) \wedge (a_2 \# b_2)$ 
   $(a_1, a_2) + (b_1, b_2)$  :=  $(a_1 + b_1, a_2 + b_2)$ 
instance (( $\alpha : \text{multi-sep-alg}$ )  $\times$  ( $\beta : \text{multi-sep-alg}$ )) : multi-sep-alg
  unitof  $(a_1, a_2)$  := (unitof  $a_1$ , unitof  $a_2$ )
instance (( $\alpha : \text{sep-alg}$ )  $\times$  ( $\beta : \text{sep-alg}$ )) : sep-alg
   $0$  :=  $(0, 0)$ 

Option Type
datatype  $\alpha$  option = Some  $\alpha$  | None
instance ( $\alpha : \text{perm-alg}$ ) option : sep-alg
   $a \# b$  := case  $(a, b)$  of
    (None,  $b$ )  $\Rightarrow$  True
  | ( $a$ , None)  $\Rightarrow$  True
  | (Some  $x$ , Some  $y$ )  $\Rightarrow$  ( $x \# y$ )
   $a + b$  := case  $(a, b)$  of
    (None,  $b$ )  $\Rightarrow$   $b$ 
  | ( $a$ , None)  $\Rightarrow$   $a$ 
  | (Some  $x$ , Some  $y$ )  $\Rightarrow$  Some  $(x + y)$ 
  unitof  $a$  := None
   $0$  := None

```

■ **Figure 2** Resource algebras instances for basic types.

The multiplicative unit ($\mathbb{1}$, Figure 2) is another permission algebra; it acts as an indivisible permission. By placing this permission in a tuple with the discrete permission algebra ($\alpha \text{ discr}$, Figure 2), we obtain a model of non-duplicable values.

Using these type-classes, we can develop compositional instances for standard data-types, such as sums (+), tuples (\times), functions (\Rightarrow), and options ($\alpha \text{ option}$). Note that such instances have already been described in previous literature [12]. For this paper, it is sufficient to note the following: tuples inherit the (least specific) class of their components, options transform permission algebras to separation algebras, and functions inherit the class of their co-domain.

Given these instantiations, it becomes simple to create various complex separation algebras built from these simple ones. For example, the standard heap model is encoded as

$$(\alpha, \beta) \text{ heap} := \alpha \multimap (\beta \text{ discr} \times \mathbb{1}),$$

where $\alpha \multimap \beta := \alpha \Rightarrow \beta \text{ option}$. One key point to structuring our type-class hierarchy in this manner, distinguishing permission algebras from multi-unit algebras from separation algebras, is to allow *flexibility* for the proof engineer. For example: to change the previous heap instance to use fractional permissions [6, 5], one only needs to swap the $\mathbb{1}$ for $\mathbf{P}_{\mathbb{Q}}$.

3 The GenRGSep Logic

Using these resource algebras, we can construct a generic RGSep [39, 37], a combination of separation logic and rely-guarantee, to reason over programs in resource models other than the standard heap model.

3.1 Language

The language (Figure 3) includes skip statements (**skip**), sequencing ($c_1; c_2$), parallel ($c_1 \parallel c_2$), and do-loops (**do** c **od**). Atomic statements ($\langle b \rangle$) are specified by a relation between states (b), and execution is *blocked* when the state is not in the domain of the relation. Inspired by CSP [20], we also distinguish between internal ($c_1 + c_2$) and external ($c_1 \square c_2$) non-determinism. We have chosen to include both internal and external non-determinism, and also relational atomic actions, because they provide a generic foundation upon which to build more concrete languages. The standard while-loop and if-then-else constructs can be encoded using external non-determinism, blocking guards, and do loops.

The state model for this language is composed of two parts: local and shared state. Thus we represent our state as a tuple, the left representing the local state and the right representing the shared state. Local state splits among the processes on parallel composition, whereas shared state is shared identically between the processes. Note that we choose *not* to explicitly model a store, because such an abstraction is not present in low-level state models.

The relational atomic statement, in particular, allows the definition of the specific atomic actions appropriate for whichever resource model the logic is instantiated with. For the same reason, the relation acts over a pair of local and shared state, which allows the resource models for local and shared state to differ. Moreover, this removes the requirement from standard RGSep that the shared part of the pre- and postconditions must pick out the shared state precisely.

Logical Variables	
r, g, b	(state relations)
p, q	(state predicates)
Commands	
$c ::= \mathbf{skip}$	(skip)
$c_1; c_2$	(sequence)
$c_1 + c_2$	(internal non-det.)
$c_1 \square c_2$	(external non-det.)
$c_1 \parallel c_2$	(parallel)
$\langle b \rangle$	(relational atomic action)
$\mathbf{do} c \mathbf{od}$	(do loop)
Abbreviations	
$[p] := \langle \lambda x y. p \ x \wedge x = y \rangle$	(guard)
$\mathbf{while} p \mathbf{do} c \mathbf{done} := \mathbf{do} ([p]; c) \square [\neg p] \mathbf{od}$	(while loop)
$\mathbf{if} p \mathbf{then} c_1 \mathbf{else} c_2 \mathbf{fi} := ([p]; c_1) \square ([\neg p]; c_2)$	(if-then-else)

Small-Step Semantics

$$\begin{array}{c}
 \frac{(h, c_1) \sim \alpha \rightsquigarrow (h', c'_1)}{(h, c_1; c_2) \sim \alpha \rightsquigarrow (h', c'_1; c_2)} \text{Seq}_L \quad \frac{}{(h, \mathbf{skip}; c_2) \sim \tau \rightsquigarrow (h, c_2)} \text{Seq}_R \\
 \frac{(h, c_1) \sim \alpha \rightsquigarrow (h', c'_1)}{(h, c_1 + c_2) \sim \alpha \rightsquigarrow (h', c'_1)} \text{INDet}_L \quad \frac{(h, c_2) \sim \alpha \rightsquigarrow (h', c'_2)}{(h, c_2 + c_2) \sim \alpha \rightsquigarrow (h', c'_2)} \text{INDet}_R \\
 \frac{}{(h, \mathbf{skip} \square c_2) \sim \tau \rightsquigarrow (h, c_2)} \text{ENDetSkip}_L \quad \frac{}{(h, c_1 \square \mathbf{skip}) \sim \tau \rightsquigarrow (h, c_1)} \text{ENDetSkip}_R \\
 \frac{(h, c_1) \sim \tau \rightsquigarrow (h', c'_1)}{(h, c_1 \square c_2) \sim \tau \rightsquigarrow (h', c'_1 \square c_2)} \text{ENDetTau}_L \quad \frac{(h, c_2) \sim \tau \rightsquigarrow (h', c'_2)}{(h, c_2 \square c_2) \sim \tau \rightsquigarrow (h', c_1 \square c'_2)} \text{ENDetTau}_R \\
 \frac{(h, c_1) \sim \alpha \rightsquigarrow (h', c'_1)}{(h, c_1 \square c_2) \sim \alpha \rightsquigarrow (h', c'_1)} \text{ENDet}_L \quad \frac{(h, c_2) \sim \alpha \rightsquigarrow (h', c'_2)}{(h, c_2 \square c_2) \sim \alpha \rightsquigarrow (h', c'_2)} \text{ENDet}_R \\
 \frac{}{(h, \mathbf{skip} \parallel \mathbf{skip}) \sim \tau \rightsquigarrow (h, \mathbf{skip})} \text{ParSkip} \\
 \frac{(h, c_1) \sim \alpha \rightsquigarrow (h', c'_1)}{(h, c_1 \parallel c_2) \sim \alpha \rightsquigarrow (h', c'_1 \parallel c_2)} \text{Par}_L \quad \frac{(h, c_2) \sim \alpha \rightsquigarrow (h', c'_2)}{(h, c_2 \parallel c_2) \sim \alpha \rightsquigarrow (h', c_1 \parallel c'_2)} \text{Par}_R \\
 \frac{(h, c) \sim \alpha \rightsquigarrow (h', c')}{(h, \mathbf{do} c \mathbf{od}) \sim \alpha \rightsquigarrow (h', c'; \mathbf{do} c \mathbf{od})} \text{DoStep} \quad \frac{\neg \text{enabled } c \ h}{(h, \mathbf{do} c \mathbf{od}) \sim \tau \rightsquigarrow (h, \mathbf{skip})} \text{DoEnd} \\
 \frac{b \ h \ h'}{(h, \langle b \rangle) \sim \text{Upd} \rightsquigarrow (h', \mathbf{skip})} \text{Atomic}
 \end{array}$$

where $\text{enabled } c \ h$ holds when there is some head atomic command $\langle b \rangle$ in c where h is in the domain of b .

■ **Figure 3** Language syntax and small-step semantics.

3.2 Semantics

We give the language a small step semantics (Figure 3) with the relation $(h, c) \sim_{\alpha} \rightsquigarrow (h', c')$. This should be interpreted as: starting with state h and program c , an α -step can be taken to state h' and program c' .

Steps are divided into two sorts of actions: τ -actions that represent internal decisions a process makes that are not directly observable by other processes and observable actions that are visible to other processes. Examples of τ -actions include the outcome of a non-deterministic choice and the end of a while loop. An example of an observable action is heap updates. This distinction is reflected by the fact that, when we connect these semantics to RGSep, it will be the observable actions that generate the guarantee. As is traditional, we will use the variable α to stand for any action and the variable a to stand for any observable action.

We only have one observable action: **Upd**, for atomic update actions. The distinction between internal and update commands is all that is necessary to prove soundness with respect to the operational semantics.

3.3 Separation Logic

We shallowly embed the predicates in Isabelle/HOL, rather than constructing a deeply embedded predicate language. The definitions of separating conjunction, separating implication, and the empty predicate are standard.

$$\begin{aligned}
 (*) & : ((\alpha : \text{perm-alg}) \Rightarrow \text{bool}) \Rightarrow (\alpha \Rightarrow \text{bool}) \Rightarrow (\alpha \Rightarrow \text{bool}) \\
 p * q & := \lambda x. \exists x_1 x_2. x_1 \# x_2 \wedge x = x_1 + x_2 \wedge p x_1 \wedge q x_2 \\
 (-*) & : ((\alpha : \text{perm-alg}) \Rightarrow \text{bool}) \Rightarrow (\alpha \Rightarrow \text{bool}) \Rightarrow (\alpha \Rightarrow \text{bool}) \\
 p -* q & := \lambda h. \forall h_1. h \# h_1 \longrightarrow p h_1 \longrightarrow q (h + h_1) \\
 \text{emp} & : ((\alpha : \text{perm-alg}) \Rightarrow \text{bool}) \\
 \text{emp} & := \lambda x. x \# x \wedge (\forall a. a \# x \longrightarrow a + x = a)
 \end{aligned}$$

Slightly less standard (notationally) is the connective $(*\wedge)$. This is defined as

$$\begin{aligned}
 (*\wedge) & : ((\alpha : \text{perm-alg}) \times (\beta : \text{perm-alg}) \Rightarrow \text{bool}) \Rightarrow (\alpha \times \beta \Rightarrow \text{bool}) \Rightarrow (\alpha \times \beta \Rightarrow \text{bool}) \\
 p * \wedge q & := \lambda(x, y). \exists x_1 x_2. x_1 \# x_2 \wedge x = x_1 + x_2 \wedge p(x_1, y) \wedge q(x_2, y),
 \end{aligned}$$

and plays the role of the RGSep separating conjunction. We define this connective separately, as the standard permission algebra instance for tuples splits both the left and right parts of the tuple, not only the left part (the local resources), which is what RGSep requires.

Note also that, as we wish to formalise RGSep shallowly, we do not have Vafeiadis' boxed-predicates, which are a syntactic construct which demarcates predicates on the shared state. To regain the ease of reasoning that predicates acting on just the local or shared state provide, we define two liftings \mathcal{L} and \mathcal{S} from predicates on local and shared states, respectively, to predicates on the overall state

$$\begin{aligned}
 \mathcal{L} & : (\alpha \Rightarrow \text{bool}) \Rightarrow (\alpha \times \beta \Rightarrow \text{bool}) & \mathcal{S} & : (\beta \Rightarrow \text{bool}) \Rightarrow (\alpha \times \beta \Rightarrow \text{bool}) \\
 \mathcal{L} p & := p \circ \text{fst} & \mathcal{S} p & := p \circ \text{snd}
 \end{aligned}$$

Unlike Vafeiadis, our \mathcal{S} does not enforce that the local state is empty, as units are not guaranteed to exist in a permission algebra.

Using these, we can prove that $*\wedge$ is indeed the standard RGSep separating conjunction, by showing that the connective separates over local state, $\mathcal{L} p * \wedge \mathcal{L} q = \mathcal{L}(p * q)$, and is additive over shared state, $\mathcal{S} p * \wedge \mathcal{S} q = \mathcal{S}(p \wedge q)$.

3.4 Stabilisation Predicate Transformers

In our formalisation of RGSep, instead of adding side-conditions to the reasoning rules asserting that our pre- and postconditions are stable (invariant under the action of the rely, guarantee, or both), we instead use stabilisation predicate transformers [37, 41]. These ease reasoning about stability in RGSep, because they semi-distribute over $*\wedge$. This means that the stability of a predicate can be proven from the stability of its parts, unlike stability side-conditions, which do not distribute at all with $*\wedge$. They are defined using relational weakest precondition (**wlp**) and strongest postcondition (**sp**) predicate transformers [11], defined as follows: **wlp** $r q := (\lambda x. \forall y. r x y \longrightarrow q y)$ and **sp** $r p := (\lambda y. \exists x. r x y \wedge p x)$.

If we know we have a state that meets the predicate q , and we wish to know what the state could have been *before* the interference of the environment, we calculate the *weakest* assertion *stronger* than q and *stable* under r (the weakest stronger stable assertion, **wssa**). If we know we have a state that meets the predicate p , and we wish to know what the state might be *after* the interference of the environment, we calculate the *strongest* assertion *weaker* than p and *stable* under r (the strongest stable weaker assertion, **sswa**). These are defined as follows:

$$\mathbf{wssa} \ r \ q := \mathbf{wlp} \ ((=) \times_{\mathcal{R}} r^*) \ q \qquad \mathbf{sswa} \ r \ p := \mathbf{sp} \ ((=) \times_{\mathcal{R}} r^*) \ p.$$

where $r_1 \times_{\mathcal{R}} r_2 := \lambda(x_1, x_2) (y_1, y_2). r_1 x_1 y_1 \wedge r_2 x_2 y_2$, and thus $((=) \times_{\mathcal{R}} r^*)$ is the relation that leaves the local state the same, and changes the shared state by the reflexive transitive closure of r .

Useful facts are that **wssa** is an interior operator and **sswa** is a closure operator,

$$\begin{array}{ll} \mathbf{wssa} \ r \ p \longrightarrow p & p \longrightarrow \mathbf{sswa} \ r \ p \\ \mathbf{wssa} \ r \ (\mathbf{wssa} \ r \ p) \longleftarrow \mathbf{wssa} \ r \ p & \mathbf{sswa} \ r \ (\mathbf{sswa} \ r \ p) \longleftarrow \mathbf{sswa} \ r \ p \\ (p \longrightarrow q) \wedge \mathbf{wssa} \ r \ p \longrightarrow \mathbf{wssa} \ r \ q & (p \longrightarrow q) \wedge \mathbf{sswa} \ r \ p \longrightarrow \mathbf{sswa} \ r \ q; \end{array}$$

they distribute or semi-distribute over the logical connectives

$$\begin{array}{ll} \mathbf{wssa} \ r \ (p \wedge q) \longleftarrow \mathbf{wssa} \ r \ p \wedge \mathbf{wssa} \ r \ q & \mathbf{sswa} \ r \ (p \wedge q) \longrightarrow \mathbf{sswa} \ r \ p \wedge \mathbf{sswa} \ r \ q \\ \mathbf{wssa} \ r \ p \vee \mathbf{wssa} \ r \ q \longrightarrow \mathbf{wssa} \ r \ (p \vee q) & \mathbf{sswa} \ r \ (p \vee q) \longleftarrow \mathbf{sswa} \ r \ p \vee \mathbf{sswa} \ r \ q \\ \mathbf{wssa} \ r \ p * \wedge \mathbf{wssa} \ r \ q \longrightarrow \mathbf{wssa} \ r \ (p * \wedge q) & \mathbf{sswa} \ r \ (p * \wedge q) \longrightarrow \mathbf{sswa} \ r \ p * \wedge \mathbf{sswa} \ r \ q; \end{array}$$

and they do not interact with local state

$$\mathbf{wssa} \ r \ (\mathcal{L} \ p) \longleftarrow \mathcal{L} \ p \qquad \mathbf{sswa} \ r \ (\mathcal{L} \ p) \longleftarrow \mathcal{L} \ p;$$

and this is the case even under a $*\wedge$ for **sswa**

$$\mathbf{sswa} \ r \ (\mathcal{L} \ p * \wedge q) \longleftarrow \mathcal{L} \ p * \wedge \mathbf{sswa} \ r \ q.$$

$$\begin{array}{c}
\frac{}{r, g \vdash \{p\} \text{ skip } \{ \text{sswa } r p \}} \text{Skip} \quad \frac{r, g \vdash \{p_1\} c_1 \{p_2\} \quad r, g \vdash \{p_2\} c_2 \{p_3\}}{r, g \vdash \{p_1\} c_1; c_2 \{p_3\}} \text{Seq} \\
\frac{\text{sp } b (\text{wssa } r p) \subseteq \text{sswa } r q \quad \forall f. \text{sp } b (\text{wssa } r (p * \wedge f)) \subseteq \text{sswa } r (q * \wedge f) \quad \top \times_{\mathcal{R}} g \subseteq b}{r, g \vdash \{ \text{wssa } r p \} \langle b \rangle \{ \text{sswa } r q \}} \text{Atomic} \\
\frac{r, g \vdash \{p\} c_1 \{q_1\} \quad r, g \vdash \{p\} c_2 \{q_2\}}{r, g \vdash \{p\} c_1 + c_2 \{q_1 \vee q_2\}} \text{INDet} \quad \frac{r, g \vdash \{p\} c_1 \{q_1\} \quad r, g \vdash \{p\} c_2 \{q_2\}}{r, g \vdash \{p\} c_1 \square c_2 \{q_1 \vee q_2\}} \text{ENDet} \\
\frac{(r \cup g_2), g_1 \vdash \{p_1\} c_1 \{q_1\} \quad (r \cup g_1), g_2 \vdash \{p_2\} c_2 \{q_2\} \quad \text{sswa } (r \cup g_2) q_1 \subseteq q'_1 \quad \text{sswa } (r \cup g_1) q_2 \subseteq q'_2}{r, (g_1 \cup g_2) \vdash \{p_1 * \wedge p_2\} c_1 \parallel c_2 \{q'_1 * \wedge q'_2\}} \text{Par} \\
\frac{r, g \vdash \{ \text{sswa } r i \} c \{ \text{sswa } r i \}}{r, g \vdash \{i\} \text{ do } c \text{ od } \{ \text{sswa } r i \}} \text{Do} \quad \frac{r, g \vdash \{p\} c \{q\} \quad \text{sswa } (r \cup g) f \subseteq f'}{r, g \vdash \{p * \wedge f\} c \{q * \wedge f'\}} \text{Frame} \\
\frac{r, g \vdash \{p_1\} c \{q_1\} \quad r, g \vdash \{p_2\} c \{q_2\}}{r, g \vdash \{p_1 \vee p_2\} c \{q_1 \vee q_2\}} \text{Disj} \\
\frac{r, g \vdash \{p_1\} c \{q_1\} \quad r, g \vdash \{p_2\} c \{q_2\} \quad \text{for all local states } h_1, \text{cancellative } h_1}{r, g \vdash \{p_1 \wedge p_2\} c \{q_1 \wedge q_2\}} \text{Conj} \\
\frac{r', g' \vdash \{p'\} c \{q'\} \quad p \subseteq p' \quad q' \subseteq q \quad r \subseteq r' \quad g' \subseteq g}{r, g \vdash \{p\} c \{q\}} \text{Weaken}
\end{array}$$

where

$\text{cancellative} : (\alpha : \text{perm-alg}) \Rightarrow \text{bool}$

$\text{cancellative } z := \forall x y. x \# z \wedge y \# z \wedge x + z = y + z \longrightarrow x = y.$

■ **Figure 4** The GenRGSep Logic.

3.5 RGSep Reasoning

The RGSep judgement, $r, g \vdash \{p\} c \{q\}$, should be interpreted as follows: if we can rely on the environment changing the shared state according to r , and we start in a state that satisfies the precondition p , then successful execution of the program c will result in a state that satisfies the postcondition q , only changing the shared state according to g . The rules for this judgement can be found in Figure 4.

4 Soundness

To prove soundness, we must extend the individual small-step rules above to a semantics of the execution of the entire program. We apply Vafeiadis' operational soundness approach [38], where the program execution not only integrates the transitive closure of small steps, but requires that each small step be closed under framing by a local state. We generalise this approach to permission algebras, which means that we do not assume either the presence of units or the cancellativity law (Figure 1).

4.1 Safety

The inductive judgement *safe* establishes that a program c can: take n steps from the state (h_l, h_s) , where h_l is the local state and h_s is the shared state; under interference from rely relation r ; while ensuring the guarantee g for each Upd step; and that the state satisfies the postcondition q if c has terminated. (Note also that rely steps are counted as steps.) The formal definition of *safe* is as follows:

► **Definition 1** (Safety).

Inductively:

1. 0: $\text{safe } 0 \ c \ h_l \ h_s \ r \ g \ q$ always holds;
2. $\text{Suc } n$: $\text{safe } (\text{Suc } n) \ c \ h_l \ h_s \ r \ g \ q$ holds if

- a. *Post-condition Safety:*

$$c = \mathbf{skip} \longrightarrow q \ (h_l, h_s),$$

- b. *Rely Safety:*

$$\forall h'_s. \ r \ h_s \ h'_s \longrightarrow \text{safe } n \ c \ h_l \ h'_s \ r \ g \ q,$$

- c. *Guarantee Safety:*

$$\forall \alpha \ h'_{lx} \ h'_{lx} \ h'_s \ c'. \ \alpha \neq \tau \wedge h_l \preceq h'_{lx} \wedge ((h'_{lx}, h_s), c) \sim \alpha \rightsquigarrow ((h'_{lx}, h'_s), c') \longrightarrow g \ h_s \ h'_s$$

- d. *Opstep Safety*

$$\forall \alpha \ h'_l \ h'_s \ c'. \ ((h_l, h_s), c) \sim \alpha \rightsquigarrow ((h'_l, h'_s), c') \longrightarrow \text{safe } n \ c \ h'_l \ h'_s \ r \ g \ q, \text{ and}$$

- e. *Frame Safety*

$$\begin{aligned} \forall \alpha \ h' \ c' \ h_{lf}. \ ((h_l + h_{lf}, h_s), c) \sim \alpha \rightsquigarrow (h', c') \longrightarrow \\ (\exists h'_l. \ h'_l \# h_{lf} \wedge h' = h'_l + h_{lf} \wedge (\alpha = \tau \longrightarrow h'_l = h_l) \wedge \text{safe } n \ c \ h'_l \ h'_s \ r \ g \ q). \end{aligned}$$

The function of each clause is as follows: taking zero steps is always safe, and when a step is taken; if execution has terminated ($c = \mathbf{skip}$) the postcondition is established, taking a rely step is safe, taking a local step under any expanded state ensures the guarantee, taking a local step is safe, and finally taking a local step under a frame is also safe and a framed local tau step steps to the same (unframed) local state.

We make a number of changes to Vafeiadis' original definition. By adding actions, we can distinguish between τ actions, that do not induce a guarantee step, and observable actions, that do. This also means that g is not forced to be reflexive by internal actions. Moreover, it allows us to combine non-cancellative models with operations such as external non-determinism, which have τ actions that do not collapse part of the program. (Compare sequencing and internal non-determinism, which destroy their connectives upon the τ move. See Paragraph 4.2.1.1 for more discussion of this.)

As we only have a single atomic statement, we do not need abort conditions to prevent multiple acquisitions of the same lock. If multiple locks are desired, these can be added either by the extension of the proof, as Vafeiadis does, or by instantiation with the appropriate resource model.

4.2 Soundness

For each language construct, a theorem is proven that the safety of the sub-commands shows the safety of the overall command. In addition, it is shown that framing by $*\wedge$ and weakening the precondition preserves safety. This then allows us to show the soundness of the RGSep proof system.

► **Theorem 2** (Soundness).

$$r, g \vdash \{p\} c \{q\} \longrightarrow p \ (h_l, h_s) \longrightarrow \text{safe } n \ c \ h_l \ h_s \ r \ g \ q$$

Proof Sketch. The proof is by induction over the RGSep rules [37]. Each safe-preservation rule discussed above corresponds to an RGSep proof rule, and proves it essentially directly, with occasional weakening of the postcondition. ◀

4.2.1 Proving Operational Soundness Without Cancellativity

Perhaps surprisingly, Vafeiadis' approach to soundness *almost* generalises to non-cancellative models without any amendment. That is, the respective safety preservation rule for each command can be proven without issue, except for external non-determinism and the conjunction rule. The reason for this is that, while the frame safety condition appears to require that we cancel a non-cancellative resource, it does not actually make the true claim of cancellativity: that the resources are *equal*. It only requires that we can safely *continue* from some unframed resource.

4.2.1.1 External Non-determinism

One place where the original proof breaks is in the τ -substep rules for external non-determinism (Figure 3), ENDetTau_L and ENDetTau_R. Here, we do find that, using the original definition of safe, which does not distinguish between actions, we need to appeal to cancellativity. External non-determinism, uniquely, has a rule which executes a τ -step, but keeps the primary operation (\square) over that executed sub-command after execution. This creates issues with the inductive proof of safety, as τ -steps always produce *equal* heaps, but Vafeiadis' original frame safety condition only required that we find *some* smaller heap. Thus, in the soundness proof of \square , in, for example, the left-step case, we would have that safe n h_1 h_s r g q and

$$((h_1 + h_{1f}, h_s), c_1) \sim \tau \rightsquigarrow ((h'_1 + h_{1f}, h_s), c'_1),$$

(from the inductive frame safety hypothesis), but be required to prove safe n h'_1 h_s r g q . This problem is resolved by strengthening the existential heap condition in frame safety, to require that $h'_1 = h_1$ in the case of a τ move.

4.2.1.2 Cancel and The Conjunction Rule

A more fundamental appeal to cancellativity appears in the safety proof of the conjunction rule. When proving the frame safety condition, as there are *two* safe assumptions, we obtain, by reduction of the hypotheses, two safe assumptions

$$\text{safe } n \ c' \ h'_1 \ h'_s \ r \ g \ q_1 \wedge \text{safe } n \ c' \ h''_1 \ h'_s \ r \ g \ q_2$$

and the relation

$$h_1 + h_{1f} = h'_1 + h_{1f},$$

but are required to find a single h_1^* such that

$$h_1^* + h_{1f} = h'_1 + h_{1f} \wedge \text{safe } n \ c' \ h_1^* \ h'_s \ r \ g \ (q_1 \wedge q_2).$$

There is no way to satisfy the inductive step, because the two safe assumptions disagree on their local states, but the inductive step requires them to be equal.

This is another appearance of the well-studied *precision* side-condition for the conjunction rule [16], as cancellativity is an instance of the precision law:

$$((=) a * \wedge (=) c) \wedge ((=) a * \wedge (=) c) \longrightarrow ((=) a \wedge (=) b) * \wedge (=) c.$$

Thus we make the pessimistic assumption that, when applying conj, every possible local state is cancellative.

4.2.1.3 Atomic

Lastly, care must be taken with atomic, as the natural framing condition to apply to the relation is the frame property [42],

$$p(x, z) \wedge x \# f \wedge b(x + f, z) \text{ } x f z' \longrightarrow \\ \exists x' z'. x' \# f \wedge x f z' = (x' + f, z') \wedge b(x, z) (x', z') \wedge q(x', z'),$$

but this is stronger than necessary to prove safety, and rules out useful atomic commands.

We only require that

$$p(x, z) \wedge x \# f \wedge b(x + f, z) \text{ } x f z' \longrightarrow \exists x' z'. x' \# f \wedge x f z' = (x' + f, y') \wedge q(x', z'),$$

which does not require that b also admits the unframed step. Note that this condition can be written more neatly as $\forall f. \mathbf{sp} b (p * \wedge f) \subseteq (q * \wedge f)$.

5 Related Work

5.1 Resource Algebras

The resource algebra approach to building separation logic was introduced by Calcagno et al. [8], although similar ideas had been applied much earlier to relevant logic by Routley and Meyer [32, 4]. There are two main styles to formalising these algebras either represent the partial plus operation with a ternary relation or have a total plus operation and a binary disjointness relation that marks when the monoid/semigroup laws actually hold. Iris [25] takes yet another approach, and has a total plus operation and total laws, but has a validity predicate which marks when the *output* of plus is not a meaningful resource.

Calcagno et al. introduce both separation algebras and permission algebras, but assume only a single unit (for separation algebras) and the cancellativity property (for both). Separation algebras were revisited by Dockins et al. [12], who formalised them in ternary style in Coq [34], noted that the algebraic structure could be weakened to include multiple units, and distilled many useful laws that extend the basic resource algebra laws. Klein et al. [26] implemented separation algebra and separation logic as an Isabelle/HOL type-class, in disjoint-plus style, which pairs well with Isabelle/HOL's simplifier. Appel et al. [3] constructed a permission–separation algebra type-class hierarchy in ternary style in Coq for VST. This implementation weakens the positivity axiom from Dockins et al. to account for the lack of the cancellativity law. Krebbers [28] formalised separation algebras in disjoint-plus style in Coq, and built a C memory model on top of them. Lastly, Iris [25] develops a very powerful concurrent separation logic in Coq, based on a generalisation of resource algebras called a Camera, that allow for the approximation of impredicative invariants using step-indexing.

5.2 RGSep

Vafeiadis' original soundness proof for RGSep was proven using cancellative separation algebra, by a pen-and-paper proof [37]. Vafeiadis later proved the soundness of RGSep for the heap model, using a much simpler proof method [38]; this proof was mechanically formalised in Coq and Isabelle/HOL.

5.3 Explicit Stabilisation

Explicit stabilisation, or, the connectives **wssa** and **sswa**, were originally defined by Vafeiadis [37] to analyse where stabilisation needed to occur in an RGSep proof. However, they were defined impredicatively. Wickerson et al. [41, 40] noted that they could be defined predicatively: respectively, as the weakest precondition and strongest postcondition of the transitive closure of the destabilising relation (e.g. the rely). They applied them to rely-guarantee, RGSep, and GSep, a proof system for reasoning about sequential programs with modules. They were applied to the verification of barriers by Dodds et al. [14], where they were noted to improve the ease of reasoning about stability, because they could be distributed through the separating conjunction.

5.4 Separation Logic Frameworks

There are many frameworks for the verification of programs using separation logic. RGSep was integrated into the automated verification tool SmallfootRG [9]. It employs symbolic execution to automatically prove the correctness of program assertion. It is specific to the abstract heap model. SmallfootRG was formalised [36] in the HOL4 theorem prover [33], again for the heap model. The Verified Software Toolchain (VST) [2] is a toolchain and framework for the verification of C code. Its foundations are built on permission algebras in Coq. Iris [25] is a particularly powerful concurrent separation logic framework, that provides an algebraic model of ghost state for the verification of concurrent code and protocols. However, the Iris logic cannot simply be embedded into Isabelle/HOL, as the later modality is incompatible with the law of excluded middle, and thus incompatible with standard Isabelle/HOL predicates.

In Isabelle/HOL, Dodds et al. [13] implement deny-guarantee, a close relative of RGSep; they use a separation algebra approach, but assume a singular unit and cancellativity. Separation Algebras have been formalised by Klein et al. [26, 27], but they assume a single unit, which prevents them from developing permissions separately, and also prevents the development of the **multi-sep-alg** instance for **discr** and **sums**. Lammich and Meis [30] develop imperative separation logic specifically for heaps. Lammich [29] develops a Concurrent Separation Logic in Isabelle/HOL based on Klein et al.'s Isabelle/HOL library, which, as noted earlier assumes a single unit. Lastly, Eilers et al. [15, 10] develop a Relational Information Flow Concurrent Separation Logic, which is specific to a combination of a fractional heap, guard state, and guard condition heap.

6 Conclusion and Future Work

In this paper, we have introduced a new Isabelle/HOL library for the development of RGSep logics. It provides a foundation for future verification of concurrent code in Isabelle/HOL.

In the future, we would like to generalise the semantics of **safe** to a proper failure trace semantics, where update actions record the state update that occurs. We believe Vafeiadis' soundness method [38] should generalise quite nicely to this, as it resembles the method of Aczel traces [1], except that extra traces are added to allow for intermittent framing.

Moreover, we would like to replace **do-od** with μ -recursion, as it appears in later CSP languages [20]. This would allow for a simple implementation of general recursion, and remove the notion of *enabled* from our semantics. This is frustrated by the fact that the standard Hoare rule for recursion [18, 21] requires non-well-founded induction on the triple. This could be solved by adding concurrent specification statements [17] to our language.

References

- 1 P Aczel. On an inference rule for parallel composition. Private communication to Cliff Jones, February 1983. URL: <https://homepages.cs.ncl.ac.uk/cliff.jones/publications/MSs/PHGA-traces.pdf>.
- 2 Andrew W. Appel. Verified software toolchain. In Gilles Barthe, editor, *Programming Languages and Systems*, pages 1–17. Springer, Berlin, Heidelberg, 2011. doi:10.1007/978-3-642-19718-5_1.
- 3 Andrew W. Appel, Robert Dockins, Aquinas Hobor, Lennart Beringer, Josiah Dodds, Gordon Stewart, Sandrine Blazy, and Xavier Leroy. Chapter 6 - Separation algebras. In *Program Logics for Certified Compilers*. Cambridge University Press, 1 edition, April 2014. doi:10.1017/CB09781107256552.
- 4 Katalin Bimbó, Jon Michael Dunn, and Nicholas Ferenz. Two manuscripts, one by Routley, one by Meyer: The origins of the Routley–Meyer semantics for relevance logics. *The Australasian Journal of Logic*, 15(2), 2018. doi:10.26686/ajl.v15i2.4066.
- 5 Richard Bornat, Cristiano Calcagno, Peter O’Hearn, and Matthew Parkinson. Permission accounting in separation logic. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL ’05, page 259–270, New York, NY, USA, 2005. Association for Computing Machinery. doi:10.1145/1040305.1040327.
- 6 John Boyland. Checking interference with fractional permissions. In Radhia Cousot, editor, *Static Analysis*, pages 55–72. Springer, Berlin, Heidelberg, 2003. doi:10.1007/3-540-44898-5_4.
- 7 Stephen Brookes and Peter W. O’Hearn. Concurrent separation logic. *ACM SIGLOG News*, 3(3):47–65, aug 2016. doi:10.1145/2984450.2984457.
- 8 Cristiano Calcagno, Peter W. O’Hearn, and Hongseok Yang. Local action and abstract separation logic. In *22nd Annual IEEE Symposium on Logic in Computer Science (LICS 2007)*, pages 366–378, 2007. doi:10.1109/LICS.2007.30.
- 9 Cristiano Calcagno, Matthew Parkinson, and Viktor Vafeiadis. Modular safety checking for fine-grained concurrency. In Hanne Riis Nielson and Gilberto Filé, editors, *Static Analysis*, pages 233–248. Springer, Berlin, Heidelberg, 2007. doi:10.1007/978-3-540-74061-2_15.
- 10 Thibault Dardinier. Formalization of commcsL: A relational concurrent separation logic for proving information flow security in concurrent programs. *Archive of Formal Proofs*, March 2023. <https://isa-afp.org/entries/CommCSL.html>, Formal proof development.
- 11 Edsger W. Dijkstra and Carel S. Scholten. *Predicate calculus and program semantics*. Springer-Verlag, Berlin, Heidelberg, 1990. doi:10.1007/978-1-4612-3228-5.
- 12 Robert Dockins, Aquinas Hobor, and Andrew W. Appel. A fresh look at separation algebras and share accounting. In Zhenjiang Hu, editor, *Programming Languages and Systems*, pages 161–177. Springer, Berlin, Heidelberg, 2009. doi:10.1007/978-3-642-10672-9_13.
- 13 Mike Dodds, Xinyu Feng, Matthew Parkinson, and Viktor Vafeiadis. Deny-guarantee reasoning. Technical Report UCAM-CL-TR-736, University of Cambridge, Computer Laboratory, January 2009. doi:10.48456/tr-736.
- 14 Mike Dodds, Suresh Jagannathan, Matthew J. Parkinson, Kasper Svendsen, and Lars Birke-dal. Verifying custom synchronization constructs using higher-order separation logic. *ACM Transactions on Programming Languages and Systems*, 38(2), jan 2016. doi:10.1145/2818638.
- 15 Marco Eilers, Thibault Dardinier, and Peter Müller. CommCSL: Proving information flow security for concurrent programs using abstract commutativity. *Proceedings of the ACM on Programming Languages*, 7(PLDI), jun 2023. doi:10.1145/3591289.
- 16 Alexey Gotsman, Josh Berdine, and Byron Cook. Precision and the conjunction rule in concurrent separation logic. *Electronic Notes in Theoretical Computer Science*, 276:171–190, September 2011. doi:10.1016/j.entcs.2011.09.021.
- 17 Ian J. Hayes. Generalised rely-guarantee concurrency: an algebraic foundation. *Formal Aspects Computing*, 28(6):1057–1078, 2016. doi:10.1007/S00165-016-0384-0.

- 18 C. A. R. Hoare. Procedures and parameters: an axiomatic approach. In E. Engeler, editor, *Symposium on Semantics of Algorithmic Languages*, pages 102–116. Springer, Berlin, Heidelberg, 1971. doi:10.1007/BFb0059696.
- 19 C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, aug 1978. doi:10.1145/359576.359585.
- 20 C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall International, 1985. URL: <http://www.usingcsp.com/cspbook.pdf>.
- 21 C. A. R. Hoare. Procedures and parameters: an axiomatic approach. In *Essays in Computing Science*, chapter 6. Prentice-Hall, Inc., USA, 1989. doi:10.5555/63445.C1104361.
- 22 Vincent Jackson. General RGSep. Software, swhId: [swh:1:dir:e759950d2ebd7571c86913f8296dfb29aa24a108](https://doi.org/10.1145/359576.359585) (visited on 2024-08-22). URL: <https://github.com/vjackson725/GeneralRGSep/tree/itp24>.
- 23 C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Transactions on Programming Languages and Systems*, 5(4):596–619, oct 1983. doi:10.1145/69575.69577.
- 24 Cliff B. Jones. *Development Methods for Computer Programs including a Notion of Interference*. PhD thesis, Oxford University, June 1981. Printed as: Programming Research Group, Technical Monograph 25. URL: <https://www.cs.ox.ac.uk/publications/publication3768-abstract.html>.
- 25 Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming*, 28:e20, 2018. doi:10.1017/S0956796818000151.
- 26 Gerwin Klein, Rafal Kolanski, and Andrew Boyton. Mechanised separation algebra. In Lennart Beringer and Amy Felty, editors, *Interactive Theorem Proving*, pages 332–337. Springer, Berlin, Heidelberg, 2012. doi:10.1007/978-3-642-32347-8_22.
- 27 Gerwin Klein, Rafal Kolanski, and Andrew Boyton. Separation algebra. *Archive of Formal Proofs*, May 2012. https://isa-afp.org/entries/Separation_Algebra.html, Formal proof development.
- 28 Robbert Krebbers. Separation algebras for C verification in Coq. In Dimitra Giannakopoulou and Daniel Kroening, editors, *Verified Software: Theories, Tools and Experiments*, pages 150–166. Springer, Cham, 2014. doi:10.1007/978-3-319-12154-3_10.
- 29 Peter Lammich. Refinement of parallel algorithms down to LLVM. In June Andronick and Leonardo de Moura, editors, *13th International Conference on Interactive Theorem Proving (ITP 2022)*, volume 237 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 24:1–24:18, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ITP.2022.24.
- 30 Peter Lammich and Rene Meis. A separation logic framework for imperative HOL. *Archive of Formal Proofs*, November 2012. https://isa-afp.org/entries/Separation_Logic_Imperative_HOL.html, Formal proof development.
- 31 Peter W. O’Hearn. Resources, concurrency, and local reasoning. *Theoretical Computer Science*, 375(1):271–307, 2007. Festschrift for John C. Reynolds’s 70th birthday. doi:10.1016/j.tcs.2006.12.035.
- 32 Richard Routley and Robert K. Meyer. The semantics of entailment. In Hugues Leblanc, editor, *Truth, Syntax and Modality*, volume 68 of *Studies in Logic and the Foundations of Mathematics*, pages 199–243. Elsevier, 1973. doi:10.1016/S0049-237X(08)71541-6.
- 33 Konrad Slind and Michael Norrish. A brief overview of HOL4. In Otmane Aït Mohamed, César A. Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings*, volume 5170 of *Lecture Notes in Computer Science*, pages 28–32. Springer, Berlin, Heidelberg, 2008. doi:10.1007/978-3-540-71067-7_6.
- 34 The Coq Development Team. The Coq reference manual – release 8.19.1. <https://coq.inria.fr/doc/V8.19.1/refman>, 2024.

- 35 Lawrence C. Paulson Tobias Nipkow, Markus Wenzel, editor. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, 2002. doi:10.1007/3-540-45949-9.
- 36 Thomas Tuerk. A formalisation of smallfoot in HOL. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics*, pages 469–484. Springer, Berlin, Heidelberg, 2009. doi:10.1007/978-3-642-03359-9_32.
- 37 Viktor Vafeiadis. Modular fine-grained concurrency verification. Technical Report UCAM-CL-TR-726, University of Cambridge, Computer Laboratory, July 2008. doi:10.48456/tr-726.
- 38 Viktor Vafeiadis. Concurrent separation logic and operational semantics. *Electronic Notes in Theoretical Computer Science*, 276:335–351, 2011. Twenty-seventh Conference on the Mathematical Foundations of Programming Semantics (MFPS XXVII). doi:10.1016/j.entcs.2011.09.029.
- 39 Viktor Vafeiadis and Matthew Parkinson. A marriage of rely/guarantee and separation logic. In Luís Caires and Vasco T. Vasconcelos, editors, *CONCUR 2007 – Concurrency Theory*, pages 256–271. Springer, Berlin, Heidelberg, 2007. doi:10.1007/978-3-540-74407-8_18.
- 40 John Wickerson. Concurrent verification for sequential programs. Technical Report UCAM-CL-TR-834, University of Cambridge, Computer Laboratory, May 2013. doi:10.48456/tr-834.
- 41 John Wickerson, Mike Dodds, and Matthew Parkinson. Explicit stabilisation for modular rely-guarantee reasoning. In Andrew D. Gordon, editor, *Programming Languages and Systems*, pages 610–629. Springer, Berlin, Heidelberg, 2010. doi:10.1007/978-3-642-11957-6_32.
- 42 Hongseok Yang and Peter O’Hearn. A semantic basis for local reasoning. In Mogens Nielsen and Uffe Engberg, editors, *Foundations of Software Science and Computation Structures*, pages 402–416. Springer, Berlin, Heidelberg, 2002. doi:10.1007/3-540-45931-6_28.

An Isabelle/HOL Formalization of Narrowing and Multiset Narrowing for E -Unifiability, Reachability and Infeasibility

Dohan Kim  

Department of Computer Science, University of Innsbruck, Austria

Abstract

We present an Isabelle/HOL formalization of narrowing for E -unifiability, reachability, and infeasibility. Given a semi-complete rewrite system \mathcal{R} and two terms s and t , we show a formalized proof that if narrowing terminates, then it provides a decision procedure for \mathcal{R} -unifiability for s and t , where \mathcal{R} is viewed as a set of equations. Furthermore, we present multiset narrowing and its formalization for multiset reachability and reachability analysis, providing decision procedures using certain restricted conditions on multiset reachability and reachability problems. Our multiset narrowing also provides a complete method for E -unifiability problems consisting of multiple goals if E can be represented by a complete rewrite system.

2012 ACM Subject Classification Theory of computation \rightarrow Logic and verification; Theory of computation \rightarrow Equational logic and rewriting

Keywords and phrases Narrowing, Multiset Narrowing, Unifiability, Reachability

Digital Object Identifier 10.4230/LIPIcs.ITP.2024.24

Funding Austrian Science Fund (FWF) project I5943.

Acknowledgements The author would like to thank René Thiemann for his valuable comments and discussion on narrowing with special encoding and multiset narrowing.

1 Introduction

Narrowing [13,18,23] generalizes rewriting in the sense that matching is replaced by unification. Narrowing is a widely used technique for solving E -unification problems using term rewriting systems, where equational unification (or E -unification) is concerned with making terms equivalent w.r.t. an equational theory E [4]. For example, consider $E = \{f(x, 0) \approx x\}$. Then, two terms $f(y, z)$ and 0 are not syntactically unifiable, but they are E -unifiable using the substitution $\theta := \{y \mapsto 0, z \mapsto 0\}$ because $f(y, z)\theta = f(0, 0) \approx_E 0$. Given a complete rewrite system \mathcal{R} representing E , narrowing is known to be *complete* for E -unification in the sense that for every solution of a given E -unification problem for s and t , a more general solution can be found by narrowing [18]. It is also known that the semi-completeness of \mathcal{R} suffices for the completeness of narrowing w.r.t. E -unification [23,30].

In logic programming [20] and constraint based theorem proving [19,25], it is often sufficient to decide the solvability of E -unification problems, called *E -unifiability* [29]. Given a set of equations E and two terms s and t , it is generally undecidable whether there exists a substitution σ such that $s\sigma \approx_E t\sigma$ holds or not [4]. It is a natural question to ask when this E -unifiability problem is decidable. E -unifiability using narrowing was considered in [29] using a complete rewrite system \mathcal{R} . However, it focuses on the complexity result of narrowing w.r.t. E -unifiability, where narrowing is used as a complete semi-decision procedure for E -unifiability.

Given a semi-complete rewrite system \mathcal{R} corresponding to E , we present a new formalized proof that (ordinary) narrowing may provide a decision procedure for E -unifiability if it terminates. Roughly speaking, if the narrowing procedure terminates, then it either reaches



© Dohan Kim;

licensed under Creative Commons License CC-BY 4.0

15th International Conference on Interactive Theorem Proving (ITP 2024).

Editors: Yves Bertot, Temur Kutsia, and Michael Norrish; Article No. 24; pp. 24:1–24:19

Leibniz International Proceedings in Informatics



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

the *success* state or not. If it reaches the success state, then we show that it yields an E -unifier. Otherwise, we show that there is no E -unifier. We provide this correctness proof of narrowing for the E -unifiability problem in the proof assistant Isabelle/HOL.

Narrowing was originally studied in the context of equational unification, but later it was also studied in the context of the *reachability problem* [14, 22, 27, 28]. Given a rewrite system \mathcal{R} and two terms s and t , the reachability problem is stated as follows: is there a substitution σ such that $s\sigma \rightarrow_{\mathcal{R}}^* t\sigma$? We say that this reachability problem is *satisfiable* if there is such a substitution σ . If no such a substitution exists, then this problem is said to be *infeasible* [21].

Narrowing is known to be *weakly complete* [22] for reachability analysis in the sense that it can find all \mathcal{R} normalized solutions if some reasonable executability assumptions on \mathcal{R} are provided. In [28], the authors proposed a semi-decision procedure, called *back-and-force narrowing*, for solving reachability goals, which is guaranteed to find a solution if it exists.

In this paper, we provide a formalized proof of some sufficient conditions of satisfying reachability problems using ordinary narrowing. Also, given a semi-complete TRS \mathcal{R} and two terms s and t , where t is a *strongly-irreducible term* [7] (e.g. a constructor term), we show a formalized proof that if narrowing terminates, then it can provide a decision procedure whether the reachability problem from s to t is satisfiable or infeasible.

Ordinary narrowing (without special encoding) has some limitations on E -unifiability and reachability analysis. In particular, it is not (directly) applicable to E -unifiability and reachability analysis consisting of multiple goals. E -unification consisting of multiple goals is considered in [15, 24] using inference rules, but they are not concerned with E -unifiability consisting of multiple goals. Meanwhile, reachability analysis consisting of multiple goals is considered in [22, 28], but they are not concerned with E -unification/ E -unifiability.


One may also use narrowing with special encoding for considering multiple narrowing goals. For example, if u_1 (resp. v_1) and u_2 (resp. v_2) are E -unifiable, then $\bar{f}(u_1, u_2)$ and $\bar{f}(v_1, v_2)$ are also E -unifiable, where \bar{f} is a new symbol. This encoding is applicable to narrowing-based E -unification/ E -unifiability consisting of multiple goals (cf. [9, 11, 12]), but has some limitations on reachability and multiset reachability analysis, which will be discussed later in this paper.

We present multiset narrowing based on multiset rewriting in order to generalize narrowing in multiset setting because identical elements (or states) in a multiset can reach different elements (or states). For example, consider the multiset $S = \{f(x, y), f(x, y)\}$, the (renamed) rewrite system $\mathcal{R} = \{f(a, b) \rightarrow d, f(a, z_1) \rightarrow g(z_1), f(z_2, a) \rightarrow d, g(a) \rightarrow c\}$, the target multiset $G = \{c, d\}$, and a variant of a reachability problem: is there a substitution σ such that $S\sigma$ can reach G by \mathcal{R} ? If we simply use the rule $f(a, b) \rightarrow d$ using the substitution $\{x \mapsto a, y \mapsto b\}$, then $f(x, y)\sigma$ reaches d but it does not reach c using the rewrite steps by \mathcal{R} . Using multiset narrowing discussed later in this paper, we can find a substitution $\sigma = \{x \mapsto a, y \mapsto a\}$, which allows $S\sigma$ to reach G using the rewriting steps by \mathcal{R} , i.e., multiset narrowing provides a means to solve multiset reachability problems.

Furthermore, both E -unifiability and reachability analysis are considered in the unified multiset narrowing framework. Our multiset narrowing works on multisets of ordinary terms for multiset reachability analysis, multisets of equational terms for E -unification/ E -unifiability along with certain restricted cases of reachability analysis, and multisets of pairs of terms for reachability analysis. It is applicable to E -unification problems and (ordinary) reachability problems consisting of multiple goals, which is generic in the sense that it simply encapsulates (ordinary) rewriting/narrowing for multiset rewriting/narrowing. In particular, it provides a complete method for E -unification and E -unifiability consisting of multiple goals, where E is represented by a complete rewrite system.

Meanwhile, Isabelle [26] is a generic proof assistant, i.e., a computer program that allows its users to express concepts in mathematics and computer science and to prove them using a logical calculus. While formalization of term rewriting has been done extensively in Isabelle (e.g., IsaFoR [1]), formalization of narrowing has not been done much yet in proof assistants including Isabelle. Our formalization of narrowing is built on IsaFoR (Isabelle/HOL *Formalization of Rewriting*). The relevant Isabelle theory files inside IsaFoR¹ under the directory `thys/Narrowing` are as follows:

<code>Narrowing.thy</code>	<code>Equational_Narrowing.thy</code>
<code>Multiset_Narrowing.thy</code>	<code>Equational_Narrowing_Unification.thy</code>
<code>Equational_Narrowing_Reachability.thy</code>	<code>Multiset_Narrowing_Unification.thy</code>
<code>Multiset_Narrowing_Reachability.thy</code>	

In the remainder of this paper, we provide hyperlinks (marked by ) to an HTML rendering for our formalized proofs in Isabelle/HOL.

2 Preliminaries

The definitions and results in this section can be found in [3, 6, 10, 18, 23]. We consider first-order terms over some signature \mathcal{F} (consisting of function symbols f, g, h, \dots with fixed arities) and some infinite set of variables $x, y, z, \dots \in \mathcal{V}$. A *position* within a term is a list of indices where ε denotes the empty position, also called the *root position*. The set of positions of a term are defined as $\mathcal{Pos}(x) = \{\varepsilon\}$ and $\mathcal{Pos}(f(t_1, \dots, t_n)) = \{\varepsilon\} \cup \{ip \mid 1 \leq i \leq n, p \in \mathcal{Pos}(t_i)\}$. Given $p \in \mathcal{Pos}(t)$, we write $t|_p$ for the subterm of t at position p , i.e., $t|_\varepsilon = t$ and $f(t_1, \dots, t_n)|_{ip} = t_i|_p$. The set of positions $\mathcal{Pos}(t)$ of a term t is partitioned into function positions $\mathcal{FPos}(t)$ and variable positions $\mathcal{VPos}(t)$, where $\mathcal{FPos}(t) = \{p \in \mathcal{Pos}(t) \mid t|_p \notin \mathcal{V}\}$. For $p \in \mathcal{Pos}(t)$, we denote by $t[s]_p$ the term that is obtained from t by replacing the subterm at position p by s .

The set of variables occurring in a term t is denoted by $\mathcal{V}(t)$.

A *substitution* σ is a mapping from \mathcal{V} to $T(\mathcal{F}, \mathcal{V})$ such that $\{x \in \mathcal{V} \mid x\sigma \neq x\}$ is finite. This set is called the *domain* of σ , which is denoted by $\mathcal{D}\sigma$, while the set of variables introduced by σ is denoted by $\mathcal{I}\sigma$. Substitutions are extended to mappings from $T(\mathcal{F}, \mathcal{V})$ to $T(\mathcal{F}, \mathcal{V})$ in the obvious way. In the remainder of this paper, we also write $s\sigma := \sigma(s)$ for substitutions σ and terms s , and $(\sigma \circ \theta)(s) := s\theta\sigma$ for substitutions θ, σ and terms s .

The *restriction* $\sigma \upharpoonright_{\mathcal{V}}$ of a substitution σ to \mathcal{V} is defined as follows:

$$\sigma \upharpoonright_{\mathcal{V}} x = \begin{cases} x\sigma & \text{if } x \in \mathcal{V} \\ x & \text{otherwise} \end{cases}$$

A *variable renaming* is a bijective substitution from \mathcal{V} to \mathcal{V} . We write $\sigma = \tau[\mathcal{V}]$ if $\sigma \upharpoonright_{\mathcal{V}} = \tau \upharpoonright_{\mathcal{V}}$ and $\sigma \leq \tau[\mathcal{V}]$ if there is a substitution θ such that $\theta \circ \sigma = \tau[\mathcal{V}]$.

An *equation* is a pair (s, t) of terms, written $s \approx t$. We denote by \approx_E the least congruence on $T(\mathcal{F}, \mathcal{V})$ that is closed under substitutions and contains a set of equations E . If $s \approx_E t$ for two terms s and t , then s and t are *E-equivalent*.

A substitution σ is a *unifier* of two terms s and t if $s\sigma = t\sigma$. It is a *most general unifier* (or *mgu* for short) if for every unifier θ of s and t , there exists a substitution λ such that $\theta = \lambda \circ \sigma$. Two terms s and t are *E-unifiable* if there exists a substitution σ such that $s\sigma \approx_E t\sigma$.

¹ <http://cl-informatik.uibk.ac.at/isafor/#downloads>
http://cl-informatik.uibk.ac.at/experiments/ITP2024/ceta_with_narrowing.zip for this paper.

A TRS \mathcal{R} is a set of ordered pairs of terms, called *rules*, where a rule is usually written $\ell \rightarrow r$. For each rule $\ell \rightarrow r$, we assume that the set of variables occurring in ℓ includes the set of variables occurring in r , i.e., $\mathcal{V}(\ell) \supseteq \mathcal{V}(r)$. The induced rewrite relation is written as $\rightarrow_{\mathcal{R}}$ and can be defined either via positions or via contexts: $s \rightarrow_{\mathcal{R}} t$ if there is some $\ell \rightarrow r \in \mathcal{R}$ and substitution σ such that $s|_p = \ell\sigma$ and $t = s[r\sigma]_p$ for some $p \in \text{Pos}(s)$ (or equivalently $s = C[\ell\sigma]$ and $t = C[r\sigma]$ for some context C).

A substitution σ is *normalized* (w.r.t. a TRS \mathcal{R}) if $x\sigma$ is a normal form for every $x \in \mathcal{D}\sigma$. A substitution σ is *normalizable* (w.r.t. a TRS \mathcal{R}) if $x\sigma$ has a normal form for every $x \in \mathcal{D}\sigma$.

A TRS \mathcal{R} is *confluent* if $\mathcal{R}^* \leftarrow \cdot \rightarrow_{\mathcal{R}}^* \subseteq \rightarrow_{\mathcal{R}}^* \cdot \mathcal{R}^* \leftarrow$. A TRS \mathcal{R} is *strongly normalizing* (SN) if there is no infinite reduction sequence $t_1 \rightarrow_{\mathcal{R}} t_2 \rightarrow_{\mathcal{R}} t_3 \rightarrow_{\mathcal{R}} \dots$. A TRS \mathcal{R} is *weakly normalizing* (WN) if every term has a normal form. A TRS \mathcal{R} is *complete* if it is confluent and strongly normalizing. A TRS \mathcal{R} is *semi-complete* if it is confluent and weakly normalizing.

A term t is *strongly irreducible* (w.r.t. \mathcal{R}) if $t\sigma$ is a normal form (w.r.t. \mathcal{R}) for all normalized substitutions σ .

A *multiset* is a collection of elements in which elements can occur more than once. More formally, a multiset is a function from an element set S to the natural numbers, giving the multiplicity of each element. This paper is only concerned with finite multisets.

3 Narrowing

► **Definition 1.** A term t is *narrowable into a term t'* if there exist a position $p \in \mathcal{FPos}(t)$, a variant² $\ell \rightarrow r$ of a rewrite rule in \mathcal{R} , and a substitution σ such that

- σ is a most general unifier of $t|_p$ and ℓ ,
- $t' = t[r]_p\sigma$.

Then, we write $t \rightsquigarrow_{[p, \ell \rightarrow r, \sigma]} t'$ or simply $t \rightsquigarrow_{\sigma, \mathcal{R}} t'$ (or more simply \rightsquigarrow). The relation \rightsquigarrow is called *narrowing*. Also, we write $t \rightsquigarrow_{\sigma, \mathcal{R}}^* t'$ if there exists a narrowing derivation $t = t_1 \rightsquigarrow_{\sigma_1, \mathcal{R}} t_2 \rightsquigarrow_{\sigma_2, \mathcal{R}} \dots \rightsquigarrow_{\sigma_{n-1}, \mathcal{R}} t_n = t'$ such that $\sigma = \sigma_{n-1} \circ \dots \circ \sigma_2 \circ \sigma_1$. If $n = 1$, then $\sigma = \varepsilon$.

► **Lemma 2 (Lifting Lemma).** Let \mathcal{R} be a TRS. Suppose that we have terms s and t , a normalized substitution θ and a set of variables V such that $\mathcal{V}(s) \cup \mathcal{D}\theta \subseteq V$ and $t = s\theta$. If $t \rightarrow_{\mathcal{R}}^* t'$, then there exist a term s' and substitutions θ', σ such that

- $s \rightsquigarrow_{\sigma, \mathcal{R}}^* s'$,
- $s'\theta' = t'$,
- $\theta' \circ \sigma = \theta[V]$,
- θ' is normalized. □

Now, we may add a fresh binary function symbol $\approx^?$ and a fresh constant \top to the set of function symbols and assume that \mathcal{R} contains the rewrite rule $x \approx^? x \rightarrow \top$

► **Definition 3.** Equational terms are the terms of the following form $s \approx^? t$, where s and t do not contain any occurrences of $\approx^?$ and \top .

We may use the lifting lemma for equational terms because equational terms are simply some specific type of terms. We often denote equational terms using uppercase letters, such as S, T, U , etc, while ordinary terms are denoted by lowercase letters, such as s, t, u , etc. We assume that if S is an equational term, then $S\sigma$ is also an equational term for any substitution σ . In other words, any substitution does not allow to introduce the special symbols $\approx^?$ and \top in its range.

² See Definition 3.1 in [23] for details.

In our Isabelle/HOL formalization, the definition of narrowing (see Definition 1) is done using `inductive_set` in Isabelle. Here, s narrows into t iff $(s, t, \delta) \in \text{narrowing_step}$.³

inductive_set `narrowing_step` where

" $(t = (\text{replace_at } s \ p \ (\text{snd } rl)) \cdot \delta \wedge \omega \bullet rl \in \mathcal{R} \wedge (\text{vars_term } s \cap \text{vars_rule } rl = \{\}) \wedge p \in \text{fun_poss } s \wedge \text{mgu } (s|_p) \ (\text{fst } rl) = \text{Some } \delta) \Rightarrow (s, t, \delta) \in \text{narrowing_step}$ "

Above, the renaming ω is applied to the rule rl , expressed by $\omega \bullet rl$, so that no variable shares between s and rl . This corresponds to a variant of a rewrite rule $l \rightarrow r$ in Definition 1, where $l \rightarrow r$ is denoted here by rl . For renaming, we use the earlier formalization of *permutation for renaming* [17] in `IsaFoR`. Now, we formalize whether a narrowing derivation $s \rightsquigarrow_{\sigma}^* t$ holds or not, which cannot simply use the reflexive and transitive closure of the relation derived from `narrowing_step` because σ should be combined and computed for the narrowing steps from s to t .

definition `narrowing_derivation` where

" $\text{narrowing_derivation } s \ s' \ \sigma \longleftrightarrow (\exists n. (\exists f \tau. f \ 0 = s \wedge f \ n = s' \wedge (\forall i < n. ((f \ i), (f \ (\text{Suc } i))), (\tau \ i)) \in \text{narrowing_step}) \wedge (\text{if } n = 0 \text{ then } \sigma = \text{Var} \text{ else } \sigma = \text{compose } (\text{map } (\lambda i. (\tau \ i)) [0.. < n])))$ "

Above, $s \rightsquigarrow_{\sigma}^* t$ is true, denoted by $(s, t, \sigma) \in \text{narrowing_derivation}$, if there are functions f and τ forming the chains of narrowing steps and their corresponding narrowing substitutions, where the end points of the chain formed by f are s and t , respectively, and σ is the composition of all substitutions of the chain formed by the function τ . (Here, if the length of the chain is 0, then σ is simply the identity substitution (i.e., $\sigma = \text{Var}$).)

Next, we need to formalize equational terms in Definition 3 in order to formalize the results in Sections 4 and 5. Formalization of equational terms needs some special treatment because of the new symbols $\approx^?$ and \top . Also, s and t in an equational term $s \approx^? t$ should not contain any occurrences of $\approx^?$ and \top . We introduce two function symbols using `locale additional_function_symbols`. Here, the binary function symbol \doteq corresponds to $\approx^?$ in Definition 3. In the following, a term t is a `wf_equational_term` if t is either the constant \top (i.e., `Fun` \top `[]`) or it is an equational term of the form $u \approx^? v$, where the binary symbol $\approx^?$ and the constant \top do not occur in any of u and v .

locale `additional_function_symbols` = **fixes** `DOTEQ` :: " f " (" \doteq ") **and** `TOP` :: " f " (" \top ")
begin

definition `wf_equational_term` where

" $\text{wf_equational_term } t \longleftrightarrow ((t = \text{Fun } \top \ []) \vee (\exists u v. t = \text{Fun } \doteq \ [u :: ('f, 'v) \text{ term}, v :: ('f, 'v) \text{ term}] \wedge (\doteq, 2) \notin \text{funas_term } u \wedge (\doteq, 2) \notin \text{funas_term } v) \wedge (\top, 0) \notin \text{funas_term } u \wedge (\top, 0) \notin \text{funas_term } v)$ "

...
end

Above, the *term* is represented by the datatype in `IsaFoR`:

datatype (α, β) `term` = `Var` β | `Fun` α `((α, β) term list)`

where α and β are type parameters.

³ Here, \mathcal{R} is added as an argument of `narrowing_step` implicitly using a `locale` in Isabelle.

24:6 Formalization of Narrowing-Based E -Unifiability, Reachability, and Infeasibility

In the Narrowing directory (below the thys directory in IsaFoR), `Narrowing.thy` is concerned with narrowing without using equational terms, while `Equational_Narrowing.thy` is concerned with equational narrowing using equational terms. Note that \mathcal{R} in the former file denotes the usual rewrite system with the condition that for each $\ell \rightarrow r \in \mathcal{R}$, $\mathcal{V}(\ell) \supseteq \mathcal{V}(r)$ and ℓ is not a variable, while \mathcal{R} in the latter file additionally includes the rule $x \approx^? x \rightarrow \top$, written by a pair $(\text{Fun} \doteq [\text{Var } x, \text{Var } x], \text{Fun } \top [])$ in our formalization. We may also need to consider the original rewrite system from \mathcal{R} excluding the rule $x \approx^? x \rightarrow \top$, where the binary symbol \doteq and the constant \top do not occur in the original rewrite system. We use the Isabelle's locale [5] to specify these in `Equational_Narrowing.thy`.

```

locale equational_narrowing = narrowing  $\mathcal{R}$  + additional_function_symbols DOTEQ TOP" +
for  $\mathcal{R} ::$       "'f, 'v :: infinite) trs"
...
fixes  $\mathcal{R}' ::$   "'f, 'v :: infinite) trs"
and  $\mathcal{F} ::$       "'f sig"
and  $\mathcal{D} ::$       "'f sig"
and  $x ::$        "'v"
assumes        "wf_trs  $\mathcal{R}$ "
and            " $\mathcal{R} = \mathcal{R}' \cup \{(Fun \doteq [Var x, Var x], Fun \top [])\}$ "
and            "funas_trs  $\mathcal{R}' \subseteq \mathcal{F}$ "
and            " $\mathcal{D} = \{(\doteq, 2), (\top, 0)\}$ "
and            " $\mathcal{D} \cap \mathcal{F} = \{\}$ "
...

```

Above, \mathcal{R}' is the original rewrite system, while \mathcal{R} is the rewrite system $\mathcal{R} = \mathcal{R}' \cup \{(Fun \doteq [Var x, Var x], Fun \top [])\}$. We assume that the function symbols of the original rewrite system \mathcal{R}' is contained in \mathcal{F} , which is written as $funas_trs \mathcal{R}' \subseteq \mathcal{F}$. Also, \mathcal{D} is the set of fresh symbols $\{(\doteq, 2), (\top, 0)\}$, which should be disjoint from the original set of function symbols \mathcal{F} (i.e., $\mathcal{D} \cap \mathcal{F} = \{\}$).

Note that the lifting lemma is a key lemma for narrowing, which states that a rewriting sequence can be “lifted” to a narrowing derivation. Our formalization includes four lifting lemmas, i.e., the lifting lemma for narrowing in `Narrowing.thy`, the lifting lemma for equational narrowing in `Equational_Narrowing.thy`, and the lifting lemma for multiset narrowing (see Lemma 22) and its slight variation in `Multiset_Narrowing.thy`, respectively. Here, we consider our formalization of the lifting lemma in equational narrowing, which is given as follows:

```

lemma lifting_lemma:
fixes  $V ::$  "'v :: infinite) set" and  $S ::$  "'f, 'v) term" and  $T ::$  "'f, 'v) term"
assumes  "normal_subst  $\mathcal{R} \theta$ "
and     "wf_equational_term  $S$ "
and     " $T = S \cdot \theta$ "
and     "vars_term  $S \cup subst\_domain \theta \subseteq V$ "
and     " $(T, T') \in rstep \mathcal{R}^*$ "
and     "finite  $V$ "
shows   " $\exists \sigma \theta' S'. narrowing\_derivation S S' \sigma \wedge T' = S' \cdot \theta' \wedge wf\_equational\_term S' \wedge$ 
normal_subst  $\mathcal{R} \theta' \wedge (\sigma \circ_s \theta') \upharpoonright_S V = \theta \upharpoonright_S V$ "

```

There are slight differences between the formalization statement above and Lemma 2. Here, we use `wf_equational_terms` instead of ordinary terms. Each narrowing step transforms one `wf_equational_term` into another `wf_equational_term`. Also, we assume that V is finite because we only consider finite `wf_equational_terms` and finite substitution domains for their associated substitutions. It is easier to rename the variables of the rules distinct from a finite V instead of the infinite V . (For example, if V is the universe of all variables of the given

type, then we cannot rename the variables of the rules distinct from V .) Also, in the above formalization statement of the lifting lemma, $(T, T') \in (\text{rstep } \mathcal{R})^*$ denotes the rewriting sequence from T to T' , where the formalization of `rstep` is already available from `IsaFoR` [1] (see below):

inductive_set `rstep`: $"_ \Rightarrow (f', v) \text{ term rel}"$ for $\mathcal{R} :: "(f', v) \text{ trs}"$ **where**
 $"\text{rstep} : \bigwedge C \sigma l r. (l, r) \in \mathcal{R} \implies s = C\langle l \cdot \sigma \rangle \implies t = C\langle r \cdot \sigma \rangle \implies (s, t) \in \text{rstep } \mathcal{R}"$

Above, $(\sigma \circ_s \theta')|_S V$ (resp. $\theta|_S V$) denotes the restriction of a substitution $\sigma \circ_s \theta'$ (resp. θ) to a set of variables V , where the restriction of a substitution `subst_restrict` is also available from `IsaFoR` (see below):

definition `subst_restrict`: $"(f', v) \text{ subst} \Rightarrow' v \text{ set} \Rightarrow (f', v) \text{ subst}"$ (infix `"|s"` 67) **where**
 $"\sigma|_S V = (\lambda x. \text{if } x \in V \text{ then } \sigma(x) \text{ else } \text{Var } x)"$

Similarly to the proof of the lifting lemma in [23], the proof of the formalization of the lifting lemma is proceeded by the induction on the length of the reduction sequence from T to T' . To this end, from the assumption $(T, T') \in (\text{rstep } \mathcal{R})^*$, we may obtain a chain and a number in such a way that

obtain $f\ n$ **where** $"f\ 0 = T"$ and $"f\ n = T'"$ and $"\forall i < n. (f\ i, f\ (\text{Suc } i)) \in \text{rstep } \mathcal{R}"$

Then we show the following statement using induction on n :

$\exists \sigma \theta' S'. \text{narrowing_derivation_num } S\ S'\ \sigma\ n \wedge T' = S' \cdot \theta' \wedge \text{wf_equation_term } S' \wedge \text{normal_subst } \mathcal{R}\ \theta' \wedge (\sigma \circ_s \theta')|_S V = \theta|_S V.$

Above, the `narrowing_derivation_num` is simply `narrowing_derivation` with the number of derivation steps being explicitly specified:

definition `narrowing_derivation_num` **where**
 $"\text{narrowing_derivation_num } s\ s'\ \sigma\ n \longleftrightarrow (\exists f \tau. f\ 0 = s \wedge f\ n = s' \wedge (\forall i < n. ((f\ i), (f\ (\text{Suc } i))), (\tau\ i)) \in \text{narrowing_step}) \wedge (\text{if } n = 0 \text{ then } \sigma = \text{Var} \text{ else } \sigma = \text{compose } (\text{map } (\lambda i. (\tau\ i)) [0.. < n]))"$

We leave it to our formalization for all the technical details of the proof of the lifting lemma.

4 E -unifiability

Narrowing is known to be a complete method of solving E -unification problems if E can be represented by a semi-complete rewrite system [23]. The completeness of narrowing w.r.t. E -unification is derived from the lifting lemma using a semi-complete rewrite system representing E . The underlying idea of using narrowing is as follows (cf. [28]): A narrowing step from a term s may represent many rewrite steps starting with instances of s . If $s\theta \rightarrow_{\mathcal{R}} t'$ is a rewrite step from $s\theta$ using a (fresh variant of) rule $\ell \rightarrow r$ at a non-variable position p of s , then $s|_p$ and ℓ are unifiable. Then, using the most general unifier δ of $s|_p$ and ℓ , we have a rewrite step $s\delta \rightarrow_{\mathcal{R}} t$ by applying the same rule $\ell \rightarrow r$ at the same position p of s , where $t' = t\sigma$ for some substitution σ . Now, the narrowing step $s \rightsquigarrow_{\delta, \mathcal{R}} t$ may represent different rewriting steps for each unifier τ of $s|_p$ and ℓ , where $s \rightsquigarrow_{\tau, \mathcal{R}} t$ implies $s\delta \rightarrow_{\mathcal{R}} t$. This can be extended to narrowing sequences in such a way that $s \rightsquigarrow_{\sigma, \mathcal{R}}^* t$ implies $s\delta \rightarrow_{\mathcal{R}}^* t$. The following lemma is used for both narrowing-based E -unification and the reachability analysis in the next section.

► **Lemma 4.**

- (i) $s \rightsquigarrow_{\sigma, \mathcal{R}}^* t$ implies $s\sigma \rightarrow_{\mathcal{R}}^* t$. ☑
- (ii) $s \approx^? t \rightsquigarrow_{\sigma, \mathcal{R}}^* \top$ implies $s\sigma \approx^? t\sigma \rightarrow_{\mathcal{R}}^* \top$. ☑

Proof. For the proof of (i), we proceed by induction on the length of the narrowing derivation $s \rightsquigarrow_{\sigma, \mathcal{R}}^* t$. The base case is immediate because we have $s = t$ and $\sigma = \varepsilon$ (i.e., the identity substitution). For the inductive case, we have some u such that $s \rightsquigarrow_{\sigma_1, \mathcal{R}}^* u \rightsquigarrow_{\sigma_2, \mathcal{R}} t$, where the length of the narrowing derivation $s \rightsquigarrow_{\sigma_1, \mathcal{R}}^* u$ is one less than the length of the narrowing derivation in $s \rightsquigarrow_{\sigma, \mathcal{R}}^* t$ with $\sigma = \sigma_2 \circ \sigma_1$. The induction hypothesis yields $s\sigma_1 \rightarrow_{\mathcal{R}}^* u$. Also, by Definition 1, we see that $u\sigma_2 \rightarrow_{\mathcal{R}} t$ from $u \rightsquigarrow_{\sigma_2, \mathcal{R}} t$. Now, we have $(s\sigma_1)\sigma_2 \rightarrow_{\mathcal{R}}^* u\sigma_2 \rightarrow_{\mathcal{R}} t$, and thus the conclusion of (i) follows. We omit the proof of (ii), since it is almost identical to the proof of (i). ◀

Recall that we have the rule $x \approx^? x \rightarrow \top$ included in \mathcal{R} , where \top is a fresh constant symbol. This means that if $s\theta \approx^? t\theta \rightarrow_{\mathcal{R}}^* \top$, then θ is an \mathcal{R} -unifier of s and t because $s\theta$ and $t\theta$ should be joined by \mathcal{R} . (Otherwise, no rewriting sequence by \mathcal{R} from $s\theta \approx^? t\theta$ reaches \top .) Now, the following lemma directly follows from this observation using Lemma 4(ii).

► **Lemma 5** ([23]). *Given a TRS \mathcal{R} , if $s \approx^? t \rightsquigarrow_{\sigma, \mathcal{R}}^* \top$ for some substitution σ , then s and t are \mathcal{R} -unifiable.* ☑

In the above, given a set of equations E represented by a rewrite system \mathcal{R} , E -unifiable is formalized in the following way, where eq is a pair of terms for representing an equation, and τ denotes an E -unifier.

definition " E -unifiable $eq \iff (\exists \tau. ((fst\ eq) \cdot \tau, (snd\ eq) \cdot \tau \in (rstep\ \mathcal{R})^{\leftrightarrow*}))$ "

► **Example 6.** Let $E = \{f(x, 0) \approx g(x), g(b) \approx c\}$ and consider the unification problem $f(x, y) \approx_E^? c$. A rewrite system for E is $\mathcal{R} = \{f(x, 0) \rightarrow g(x), g(b) \rightarrow c, x \approx^? x \rightarrow \top\}$, where the rule $x \approx^? x \rightarrow \top$ is added using the fresh constant \top . We rename the rules in \mathcal{R} whenever necessary, where variables with subscripts denote the renamed variables in this example. First, find the *mgu* of $f(x, y)$ and $f(x_1, 0)$ in $f(x_1, 0) \rightarrow g(x_1)$, which is $\sigma_1 = \{x \mapsto x_1, y \mapsto 0\}$. This yields the narrowing step $(f(x, y) \approx^? c) \rightsquigarrow_{\sigma_1} (g(x_1) \approx^? c)$. Next, find the *mgu* of $g(x_1)$ and $g(b)$, which is $\sigma_2 = \{x_1 \mapsto b\}$. This yields the narrowing step $(g(x_1) \approx^? c) \rightsquigarrow_{\sigma_2} (c \approx^? c)$. Then, find the *mgu* of $c \approx^? c$ and $x_2 \approx^? x_2$ in $x_2 \approx^? x_2 \rightarrow \top$, which is $\sigma_3 = \{x_2 \mapsto c\}$. This yields the narrowing step $c \approx^? c \rightsquigarrow_{\sigma_3} \top$.

We see that $\sigma := \sigma_3 \circ \sigma_2 \circ \sigma_1$ is an \mathcal{R} -unifier (or an E -unifier) of $f(x, y)$ and c , where $\sigma = \{x \mapsto b, y \mapsto 0, x_1 \mapsto b, x_2 \mapsto c\}$.

Now, given a semi-complete TRS \mathcal{R} , if θ is an \mathcal{R} -unifier of s and t (i.e., $s\theta \approx_{\mathcal{R}} t\theta$), then $s\theta \approx^? t\theta \rightarrow_{\mathcal{R}}^* \top$ because \mathcal{R} is confluent. By the semi-completeness of \mathcal{R} , a normal substitution θ' of θ exists such that $s\theta' \approx^? t\theta' \rightarrow_{\mathcal{R}}^* \top$, and thus θ' is also an \mathcal{R} -unifier of s and t . Applying the lifting lemma yields a narrowing sequence $s \approx^? t \rightsquigarrow_{\sigma, \mathcal{R}}^* \top$ such that $\sigma \leq \theta' [\mathcal{V}(s) \cup \mathcal{V}(t)]$. By Lemma 4(ii), we have $s\sigma \approx^? t\sigma \rightarrow_{\mathcal{R}}^* \top$, and thus σ is also an \mathcal{R} -unifier of s and t . Since we have $\theta \approx_{\mathcal{R}} \theta'$ and $\sigma \leq \theta' [\mathcal{V}(s) \cup \mathcal{V}(t)]$, we see that $\sigma \leq_{\mathcal{R}} \theta [\mathcal{V}(s) \cup \mathcal{V}(t)]$. This observation implies that for every \mathcal{R} -unifier of s and t , a more general \mathcal{R} -unifier can be found by narrowing. The completeness of narrowing for E -unification was originally proposed by Hullot [18], where E is represented by a complete TRS. Later, it was shown that the semi-completeness of TRS suffices for the completeness of narrowing for E -unification [23].

► **Theorem 7** ([23]). *Let \mathcal{R} be a semi-complete TRS. If $s\theta \approx_{\mathcal{R}} t\theta$, then there is a narrowing derivation $s \approx^? t \rightsquigarrow_{\sigma, \mathcal{R}}^* \top$ such that $\sigma \leq_{\mathcal{R}} \theta [\mathcal{V}(s) \cup \mathcal{V}(t)]$.* ☑

Unfortunately, the completeness of narrowing for E -unification alone does not imply E -unifiability by narrowing, which is also important in equational reasoning. In the remainder of this section, we show that given a semi-complete TRS \mathcal{R} , if narrowing terminates, then it provides a decision procedure for E -unifiability.

► **Lemma 8.** *Given a TRS \mathcal{R} , if there is no narrowing derivation $s \approx^? t \rightsquigarrow_{\sigma, \mathcal{R}}^* \top$ for any substitution σ , then there is no normal substitution θ satisfying $s\theta \approx^? t\theta \rightarrow_{\mathcal{R}}^* \top$. \square*

Proof. Suppose to the contrary that there is a normal substitution θ satisfying $s\theta \approx^? t\theta \rightarrow_{\mathcal{R}}^* \top$. Let $V =: \mathcal{V}(S) \cup \mathcal{D}\theta$, where $S = s \approx^? t$. Then, by Lemma 2, there exists some substitution σ such that $s \approx^? t \rightsquigarrow_{\sigma, \mathcal{R}}^* \top$, which is the required contradiction. ◀

► **Example 9.** Consider $\mathcal{R} = \{a \rightarrow b, f(a, b) \rightarrow c\}$ and $s = f(x, x)$ and $t = c$. Then $s \approx^? t$ is not narrowable, so there is no narrowing derivation $s \approx^? t \rightsquigarrow_{\sigma, \mathcal{R}}^* \top$ for any substitution σ . By Lemma 8, there is no normal substitution θ satisfying $s\theta \approx^? t\theta \rightarrow_{\mathcal{R}}^* \top$. However, there is a *non-normal* substitution $\delta := \{x \mapsto a\}$ satisfying $s\delta \approx^? t\delta \rightarrow_{\mathcal{R}}^* \top$, i.e., $f(a, a) \approx^? c \rightarrow_{\mathcal{R}} f(a, b) \approx^? c \rightarrow_{\mathcal{R}} c \approx^? c \rightarrow_{\mathcal{R}} \top$, where $s\delta = f(a, a)$ and $t\delta = c$.

The following lemma is immediate by observing that given a confluent TRS, $s \leftrightarrow_{\mathcal{R}}^* t$ implies that s and t are joinable.

► **Lemma 10.** *Given a confluent TRS \mathcal{R} , $s \leftrightarrow_{\mathcal{R}}^* t$ implies $s \approx^? t \rightarrow_{\mathcal{R}}^* \top$. \square*

► **Lemma 11.** *Given a semi-complete TRS \mathcal{R} , if there is no narrowing derivation $s \approx^? t \rightsquigarrow_{\sigma, \mathcal{R}}^* \top$ for any substitution σ , then s and t have no R -unifier. \square*

Proof. Assume that there is no narrowing derivation $s \approx^? t \rightsquigarrow_{\sigma, \mathcal{R}}^* \top$ for any substitution σ . Then, by Lemma 8, there is no normal substitution θ satisfying $s\theta \approx^? t\theta \rightarrow_{\mathcal{R}}^* \top$. Now, suppose, towards a contradiction, that s and t have an \mathcal{R} -unifier. Then, there is some substitution τ such that $s\tau \leftrightarrow_{\mathcal{R}}^* t\tau$. Since \mathcal{R} is semi-complete, there is a normal substitution τ' of τ such that $s\tau' \leftrightarrow_{\mathcal{R}}^* t\tau'$. Now, we have $s\tau' \approx^? t\tau' \rightarrow_{\mathcal{R}}^* \top$ by Lemma 10, which is the required contradiction. ◀

From Lemmas 5 and 11, we have the following theorem of E -unifiability by narrowing.

► **Theorem 12.** *Given a semi-complete TRS \mathcal{R} , if all narrowing derivations starting from $s \approx^? t$ terminate (or simply \rightsquigarrow terminates), then we can decide whether $s \approx^? t$ has an \mathcal{R} -unifier or not. \square*

5 Reachability and Infeasibility

The *reachability problem* [14, 27] is one of the fundamental problems in term rewriting systems, which originally has the following form: Given a TRS \mathcal{R} and a source term s , does s reach t by a rewriting sequence, written $s \rightarrow_{\mathcal{R}}^* t$? This problem has the following generalization [21, 27] for s and t containing variables: Is there a substitution σ such that $s\sigma \rightarrow_{\mathcal{R}}^* t\sigma$? If there is no such substitution, then the problem is called *infeasible* [21, 27]. In this paper, by the reachability problem, we mean the generalized reachability problem discussed above. In our Isabelle/HOL formalization, *reachable* and *infeasible* for a pair of terms are formalized as follows:

definition "*reachable eq* $\longleftrightarrow (\exists \tau. ((fst\ eq) \cdot \tau, (snd\ eq) \cdot \tau \in (rstep\ \mathcal{R})^*))$

definition "*infeasible eq* $\longleftrightarrow (\neg(\exists \tau. ((fst\ eq) \cdot \tau, (snd\ eq) \cdot \tau \in (rstep\ \mathcal{R})^*))$

24:10 Formalization of Narrowing-Based E -Unifiability, Reachability, and Infeasibility

The following lemma provides a sufficient condition of satisfying the reachability problem using narrowing, which requires neither the confluence nor the termination of the underlying TRS.

► **Lemma 13.**

- (i) If there is some substitution σ such that $s \rightsquigarrow_{\sigma, \mathcal{R}}^* t\sigma$, then the reachability problem from s to t is satisfiable. ☑
- (ii) If there is some substitution σ such that $s \rightsquigarrow_{\sigma, \mathcal{R}}^* t'\sigma$ and $t'\sigma$ and $t\sigma$ are unifiable, then the reachability problem from s to t is satisfiable. ☑

Proof. The proof of (i) is immediate using Lemma 4. For the proof of (ii), we have $s\sigma \rightarrow_{\mathcal{R}}^* t'\sigma$ from $s \rightsquigarrow_{\sigma, \mathcal{R}}^* t'\sigma$ using Lemma 4. Since $t'\sigma$ and $t\sigma$ are unifiable, there is some *mgu* δ such that $(t'\sigma)\delta = (t\sigma)\delta$. Then, we have $(s\sigma)\delta \rightarrow_{\mathcal{R}}^* (t'\sigma)\delta = (t\sigma)\delta$, and thus the reachability problem from s to t is satisfiable using substitution $\delta \circ \sigma$. ◀

► **Example 14.** Let $\mathcal{R} = \{f(x, x) \rightarrow g(x), a \rightarrow b\}$. For the reachability problem from $f(y, a)$ to $g(b)$, we have $f(y, a) \rightsquigarrow_{\sigma_1, \mathcal{R}} g(a)$, where $\sigma_1 = \{x \mapsto a, y \mapsto a\}$ is the *mgu* of $f(x, x)$ and $f(y, a)$. Then, we have $g(a) \rightsquigarrow_{\varepsilon, \mathcal{R}} g(b)$, so the reachability problem from $f(y, a)$ to $g(b)$ is satisfiable by Lemma 13(i) using substitution $\sigma_1 = \{x \mapsto a, y \mapsto a\}$.

Lemma 13(ii) provides a means to compute a solution of the reachability problem from s to t using a narrowing tree starting from s . Since a narrowing derivation along with its substitution are computed incrementally, a typical way of computing a solution of the reachability problem using a narrowing tree is to use the breadth-first search for each length of narrowing derivations and expand the narrowing tree (if it is possible) when a solution of the reachability problem cannot be found. (A more efficient way of solving reachability problems is considered in the next section.)

However, narrowing is known to be *weakly complete* [22] in reachability analysis in the sense that it may fail to find a solution of the reachability problem even if it exists. In particular, narrowing may fail to find a *non-normalized* solution of a reachability problem.

► **Example 15.** Given $\mathcal{R} = \{a \rightarrow b, a \rightarrow c, g(f(b), f(c)) \rightarrow a\}$, consider the reachability problem from $g(f(x), f(x))$ to a . The problem is satisfiable using substitution $\{x \mapsto a\}$ (i.e., $g(f(a), f(a)) \rightarrow_{\mathcal{R}} g(f(b), f(a)) \rightarrow_{\mathcal{R}} g(f(b), f(c)) \rightarrow_{\mathcal{R}} a$), but we may not apply Lemma 13(ii) because there is neither a narrowing step from $g(f(x), f(x))$ nor is it unifiable with a .

In what condition the reachability problem is shown to be either satisfiable or infeasible using narrowing? In the remainder of this section, if \mathcal{R} is semi-complete and t is a strongly-irreducible term (e.g. a constructor term), then we show that a narrowing derivation $s \approx^? t \rightsquigarrow_{\sigma, \mathcal{R}}^* \top$ for some substitution σ implies the reachability from s to t , while no narrowing derivation $s \approx^? t \rightsquigarrow_{\sigma, \mathcal{R}}^* \top$ for any substitution σ implies the infeasibility of the reachability problem from s to t , assuming that all narrowing derivations from $s \approx^? t$ terminates.

► **Lemma 16.** Let \mathcal{R} be a semi-complete TRS and t be a strongly irreducible term. If there is some substitution σ such that $s \approx^? t \rightsquigarrow_{\sigma, \mathcal{R}}^* \top$, then the reachability problem from s to t is satisfiable. ☑

Proof. Suppose that there is some substitution σ such that $s \approx^? t \rightsquigarrow_{\sigma, \mathcal{R}}^* \top$. Then, by Lemma 4(ii), we have $s\sigma \approx^? t\sigma \rightarrow_{\mathcal{R}}^* \top$. Since \mathcal{R} is semi-complete, there is a normal substitution σ' of σ such that $s\sigma \approx^? t\sigma \rightarrow_{\mathcal{R}}^* s\sigma' \approx^? t\sigma'$ and $s\sigma' \approx^? t\sigma' \rightarrow_{\mathcal{R}}^* \top$. Also, $t\sigma'$ is a normal form of \mathcal{R} because t is strongly irreducible. Since $s\sigma' \approx^? t\sigma' \rightarrow_{\mathcal{R}}^* \top$ and $t\sigma'$ is normal form of \mathcal{R} , we may infer that $s\sigma' \rightarrow_{\mathcal{R}}^* t\sigma'$, and thus the conclusion follows. ◀

► **Lemma 17.** *Let \mathcal{R} be a semi-complete TRS and t be a strongly irreducible term. If there is no narrowing derivation $s \approx^? t \rightsquigarrow_{\sigma, \mathcal{R}}^* \top$ for any substitution σ , then the reachability problem from s to t is infeasible. \square*

Proof. Assume that there is no narrowing derivation $s \approx^? t \rightsquigarrow_{\sigma, \mathcal{R}}^* \top$ for any substitution σ . Then, by Lemma 8, there is no normal substitution θ satisfying $s\theta \approx^? t\theta \rightarrow_{\mathcal{R}}^* \top$. Now, suppose, towards a contradiction, that the reachability problem from s to t is satisfiable. Then, there is a substitution τ such that $s\tau \rightarrow_{\mathcal{R}}^* t\tau$. Since \mathcal{R} is weakly normalizing, there is a normal substitution τ' of τ such that $s\tau' \xrightarrow{\mathcal{R}}^* s\tau \rightarrow_{\mathcal{R}}^* t\tau \rightarrow_{\mathcal{R}}^* t\tau'$. We see that $t\tau'$ is a normal form because t is a strongly irreducible term and τ' is a normal substitution. Since \mathcal{R} is confluent and $t\tau'$ is a normal form of \mathcal{R} , we have $s\tau' \rightarrow_{\mathcal{R}}^* t\tau'$, and thus $s\tau' \approx^? t\tau' \rightarrow_{\mathcal{R}}^* \top$, which is the required contradiction. \blacktriangleleft

From Lemmas 16 and 17, we have the following decidability result of the reachability problem using narrowing. (Note that Lemma 13 only provides a sufficient condition of satisfying the reachability problem using narrowing.)

► **Theorem 18.** *Let \mathcal{R} be a semi-complete TRS and t be a strongly irreducible term. If all narrowing derivations starting from $s \approx^? t$ terminate (or simply \rightsquigarrow terminates), then we can decide whether the reachability problem from s to t is satisfiable or not (i.e., infeasible). \square*

6 Multiset Narrowing

In this section, we consider *multiset narrowing* for multiset reachability analysis and multiple goals in the reachability and E -unification problems. Our multiset narrowing⁴ is adapted from *Narrowing Calculus* (NC) in [24], but it is also concerned with multisets of ordinary terms, equational terms, and pairs of terms. Note that a multiset is a generalization of a set, allowing elements in the multiset to occur more than once. It has an additional flexibility because identical elements (or states) in a multiset can reach different elements (or states).

Now, we consider multiset narrowing for multisets of terms (or equational terms). First, we consider *multiset rewriting* for multisets of terms (or equational terms).

► **Definition 19.** *Let S and T be multisets of terms. We write $S \rightarrow_{[\mathcal{R}, M]} T$ if there exists a term $s \in S$ such that $s \rightarrow_{\mathcal{R}} t$ and $T = (S - \{s\}) \cup \{t\}$.*

► **Definition 20.** *Given a multiset of terms $S = \{t_1, \dots, t_n\}$, the multiset reachability problem is described as follows: is there a substitution σ such that $S\sigma := \{t_1\sigma, \dots, t_n\sigma\}$ reaches the target multiset of terms $G = \{t'_1, \dots, t'_n\}$ using multiset rewriting, i.e., $S\sigma \rightarrow_{[\mathcal{R}, M]}^* G$? If there is such a substitution σ , then we say that the multiset reachability problem from S to G is satisfiable. Otherwise, we say that it is infeasible.*

In the above definition, the source multiset S and the target multiset G are fixed for multiset reachability analysis which can be done using the following multiset narrowing.

► **Definition 21.** *A multiset of terms S is narrowable into a multiset of terms T if there exist a term $s \in S$ and a substitution σ such that*

- $s \rightsquigarrow_{\sigma, \mathcal{R}} t$,
- $T = ((S - \{s\})\sigma \cup \{t\})$.

⁴ Narrowing in a multiset environment is also considered in CHR [16], but it is considered in the context of logic programming, which does not consider multisets of ordinary terms.

24:12 Formalization of Narrowing-Based E -Unifiability, Reachability, and Infeasibility

Then, we write $S \rightsquigarrow_{\sigma, \mathcal{R}, M} T$. Also, we write $S \rightsquigarrow_{\sigma, \mathcal{R}, M}^* S'$ if there exists a narrowing derivation $S = S_1 \rightsquigarrow_{\sigma_1, \mathcal{R}, M} S_2 \rightsquigarrow_{\sigma_2, \mathcal{R}, M} \cdots \rightsquigarrow_{\sigma_{n-1}, \mathcal{R}, M} S_n = S'$ such that $\sigma = \sigma_{n-1} \circ \cdots \circ \sigma_2 \circ \sigma_1$. If $n = 1$, then $\sigma = \varepsilon$.

Intuitively speaking, $S \rightarrow_{[\mathcal{R}, M]} T$ if T is obtained by replacing one element (term) in S using a rewriting step in \mathcal{R} , while $S \rightsquigarrow_{\sigma, \mathcal{R}, M} T$ if T is obtained by replacing one element (term) in S using a narrowing step in Definition 1 and then applying the narrowing substitution to the remaining multiset $S - \{s\}$.

In our Isabelle/HOL formalization, we use finite multisets for multiset narrowing, where a finite multiset is a finite collection of elements, denoted by $\{\#x_1, \dots, x_n\#$ in Isabelle. Duplication is allowed and orders are irrelevant in multisets, i.e., $\{\#s, t, t, s\# = \{\#t, t, s, s\#$. Also, $+$ denotes multiset sum and $-$ denotes multiset difference. Now, the multiset reduction in Definition 19 can be used for multisets of both ordinary and equational terms. Then, $S \rightarrow_{[\mathcal{R}, M]} T$ iff $(S, T) \in \text{multiset_reduction_step}$ (see below).

inductive_set multiset_reduction_step where

" $s \in \# S \wedge T = (S - \{\#s\#) + \{\#t\#) \wedge (s, t) \in \text{rstep } \mathcal{R} \Rightarrow (S, T) \in \text{multiset_reduction_step}$ "

The corresponding multiset narrowing in Definition 21 is formalized as follows, where $S \rightsquigarrow_{\sigma, \mathcal{R}, M} T$ iff $(S, T, \sigma) \in \text{multiset_narrowing_step}$.

inductive_set multiset_narrowing_step where

" $(s, t) \in \# S \wedge T = (\text{subst_term_multiset } \sigma (S - \{\#s\#) + \{\#t\#) \wedge (s, t, \sigma) \in \text{narrowing_step} \Rightarrow (S, T, \sigma) \in \text{multiset_narrowing_step}$ "

The lifting lemma for multisets of terms can be easily adapted from Lemma 2.⁵

► **Lemma 22.** *Let \mathcal{R} be a TRS. Suppose we have two multisets of terms S and T , a normalized substitution θ and a set of variables V such that $\mathcal{V}(S) \cup \mathcal{D}\theta \subseteq V$ and $T = S\theta$. If $T \rightarrow_{[\mathcal{R}, M]}^* T'$, then there exist a multiset of terms S' and substitutions θ', σ such that*

- $S \rightsquigarrow_{\sigma, \mathcal{R}, M}^* S'$,
- $S'\theta' = T'$,
- $\theta' \circ \sigma = \theta[V]$,
- θ' is normalized. ☑

The following lemma can be proved by induction on the length of the multiset narrowing derivation $S \rightsquigarrow_{\sigma, \mathcal{R}, M}^* T$ using the observation that $S'\sigma' \rightarrow_{[\mathcal{R}, M]} T'$ whenever $S' \rightsquigarrow_{\sigma', \mathcal{R}, M} T'$ (cf. Lemma 4).

► **Lemma 23.** *Let \mathcal{R} be a TRS and S be a multiset of terms (or equational terms). Then, $S \rightsquigarrow_{\sigma, \mathcal{R}, M}^* T$ implies $S\sigma \rightarrow_{[\mathcal{R}, M]}^* T$. ☑*

► **Lemma 24.** *If there are some substitutions σ and η such that $S \rightsquigarrow_{\sigma, \mathcal{R}, M}^* S'$ and $S'\eta = G$, then the multiset reachability problem from S to G is satisfiable. ☑*

Proof. Suppose that there are some substitution σ and η such that $S \rightsquigarrow_{\sigma, \mathcal{R}, M}^* S'$ and $S'\eta = G$. Then, by Lemma 23, we have $S\sigma \rightarrow_{[\mathcal{R}, M]}^* S'$. By Definition 19 and easy induction on the length of multiset rewriting steps, we may infer that $\rightarrow_{[\mathcal{R}, M]}^*$ is closed under substitutions. Now, we have $S\sigma\eta \rightarrow_{[\mathcal{R}, M]}^* S'\eta = G$, and thus the conclusion follows. ◀

⁵ The lifting lemma for multisets of equational terms is also a slight variation of the lifting lemma for multisets of terms, where Definition 3 needs to be checked.

► **Example 25.** We consider the multiset reachability problem introduced in Section 1. Let $S = \{f(x, y), f(x, y)\}$, the (renamed) rewrite system $\mathcal{R} = \{f(a, b) \rightarrow d, f(a, z_1) \rightarrow g(z_1), f(z_2, a) \rightarrow d, g(a) \rightarrow c\}$, and the target multiset $G = \{c, d\}$. Multiset narrowing starts with $S = \{f(x, y), f(x, y)\}$ and narrow into $S_1 = \{g(z_1), f(a, z_1)\}$ using the rule $f(a, z_1) \rightarrow g(z_1)$ with substitution $\sigma_1 = \{x \mapsto a, y \mapsto z_1\}$. Then, it narrows into $S_2 = \{c, f(a, a)\}$ using the rule $g(a) \rightarrow c$ with substitution $\sigma_2 = \{z_1 \mapsto a\}$. Finally, it narrows into $S_3 = \{c, d\}$ using the rule $f(z_2, a) \rightarrow d$, with substitution $\sigma_3 = \{z_2 \mapsto a\}$. Then by Lemma 24, the above multiset reachability problem is satisfied with substitution $\sigma = \sigma_3 \circ \sigma_2 \circ \sigma_1 = \{x \mapsto a, y \mapsto a, z_1 \mapsto a, z_2 \mapsto a\}$.

► **Lemma 26.** *If there is no multiset narrowing derivation $S \rightsquigarrow_{\sigma, \mathcal{R}, M}^* S'$ for any substitution σ and η with $S'\eta = G$, then there is no normal substitution θ satisfying the multiset reachability problem from S to G .* ☑

The above lemma describes the *weak completeness* of multiset narrowing w.r.t. multiset reachability analysis. For example, the multiset reachability problem from $\{g(f(x), f(x))\}$ to $\{a\}$ using \mathcal{R} in Example 15 is satisfiable using substitution $\{x \mapsto a\}$, but there is no multiset narrowing step from $\{g(f(x), f(x))\}$ nor is there some substitution η such that $\{g(f(x), f(x))\eta\} = \{a\}$.

► **Lemma 27.**

- (i) *If \mathcal{R} is strongly normalizing, then $\rightarrow_{[\mathcal{R}, M]}$ is strongly normalizing.* ☑
- (ii) *If \mathcal{R} is complete, then $\rightarrow_{[\mathcal{R}, M]}$ is confluent.* ☑

► **Lemma 28.** *Given a complete TRS \mathcal{R} , if there is no multiset narrowing derivation $S \rightsquigarrow_{\sigma, \mathcal{R}, M}^* S'$ for any substitution σ and η with $S'\eta = G$ and G is in normal form w.r.t. $\rightarrow_{[\mathcal{R}, M]}$, then there is no substitution θ satisfying the multiset reachability problem from S to G .* ☑

Proof. Assume that there is no multiset narrowing derivation $S \rightsquigarrow_{\sigma, \mathcal{R}, M}^* S'$ for any substitution σ and η with $S'\eta = G$. Then, by Lemma 26, there is no normal substitution θ satisfying the multiset reachability problem from S to G . Now, suppose to the contrary that there is some substitution θ satisfying the multiset reachability problem from S to G , i.e., $S\theta \rightarrow_{[\mathcal{R}, M]}^* G$. By Lemma 27, $\rightarrow_{[\mathcal{R}, M]}$ is strongly normalizing and confluent. Now, we have $S\theta \rightarrow_{[\mathcal{R}, M]}^* S\theta'$, where θ' is the normal substitution of θ . (This can be shown using a straightforward induction on the size of $S\theta$.) Since $\rightarrow_{[\mathcal{R}, M]}$ is strongly normalizing and confluent and G is in normal form w.r.t. $\rightarrow_{[\mathcal{R}, M]}$, we have $S\theta \rightarrow_{[\mathcal{R}, M]}^* S\theta' \rightarrow_{[\mathcal{R}, M]}^* G$, contradicting that there is no normal substitution satisfying the multiset reachability problem from S to G . ◀

From Lemmas 24 and 28, we have the following decidability result of multiset reachability analysis using multiset narrowing.

► **Theorem 29.** *Let \mathcal{R} be a complete TRS \mathcal{R} , S and G be multisets of terms, and G be in normal form w.r.t. $\rightarrow_{[\mathcal{R}, M]}$. If all multiset narrowing derivations starting from S terminate, then we can decide whether the multiset reachability problem from S to G is satisfiable or not (i.e., infeasible).* ☑

Meanwhile, multiset narrowing can also be used for E -unification problems consisting of multiple goals. In the following, by a slight abuse of notation, we denote by \top a finite multiset consisting only of \top 's or simply \top in Definition 3. The next theorem provides the completeness of multiset narrowing for E -unification problems consisting of multiple goals.

► **Theorem 30.** Let \mathcal{R} be a complete TRS and $S = \{s_1 \approx^? t_1, \dots, s_n \approx^? t_n\}$ be a multiset of equational terms. If there is some \mathcal{R} -unifier θ satisfying $s_k\theta \approx_{\mathcal{R}} t_k\theta$ for all $1 \leq k \leq n$, then there is some multiset narrowing derivation $S \rightsquigarrow_{\sigma, \mathcal{R}, M}^* \top$ such that $\sigma \leq_{\mathcal{R}} \theta[\mathcal{V}(S)]$. \square

Next, we consider E -unifiability consisting of multiple goals using multiset narrowing. The following lemma provides a sufficient condition of satisfying an E -unifiability problem (consisting of multiple goals) using multiset narrowing.

► **Lemma 31.** Let \mathcal{R} be a TRS and $S = \{s_1 \approx^? t_1, \dots, s_n \approx^? t_n\}$ be a multiset of equational terms. If $S \rightsquigarrow_{\sigma, \mathcal{R}, M}^* \top$ for some substitution σ , then s_k and t_k for all $1 \leq k \leq n$ are \mathcal{R} -unifiable. \square

Proof. Suppose $S \rightsquigarrow_{\sigma, \mathcal{R}, M}^* \top$. Then, we have $S\sigma \rightarrow_{[R, M]}^* \top$ by Lemma 23. Also, $S\sigma \rightarrow_{[R, M]}^+ \top$ because it needs at least one step including the step using the rule $x \approx x \rightarrow \top$. Now, observe that for any nonempty $S' \subset S\sigma$, we have $S' \rightarrow_{[R, M]}^+ \top$. Therefore, for any $1 \leq k \leq n$, we have $\{s_k\sigma \approx t_k\sigma\} \rightarrow_{[R, M]}^+ \top$. Now, we proceed by induction on the number of $\rightarrow_{[R, M]}^+$ -steps in $\{s_k\sigma \approx t_k\sigma\} \rightarrow_{[R, M]}^+ \top$ and show that $s_k\sigma \xleftrightarrow{*}_{\mathcal{R}} t_k\sigma$.

The base case is obvious, i.e., $s_k\sigma = t_k\sigma$. For the inductive case, consider s' and t' , where $\{s_k\sigma \approx t_k\sigma\} \rightarrow_{[R, M]} \{s' \approx t'\}$ and $\{s' \approx t'\} \rightarrow_{[R, M]}^+ \top$. The induction hypothesis yields $s' \xleftrightarrow{*}_{\mathcal{R}} t'$. Since $\{s_k\sigma \approx t_k\sigma\} \rightarrow_{[R, M]} \{s' \approx t'\}$, we see that either $s_k\sigma \rightarrow_{\mathcal{R}} s'$ with $t_k\sigma = t'$ or $t_k\sigma \rightarrow_{\mathcal{R}} t'$ with $s_k\sigma = s'$ by Definition 19, and thus the conclusion follows from $s_k\sigma \xrightarrow{*}_{\mathcal{R}} s' \xleftrightarrow{*}_{\mathcal{R}} t' \xleftrightarrow{*}_{\mathcal{R}} t_k\sigma$. \blacktriangleleft

► **Lemma 32.** Let \mathcal{R} be a complete TRS and $S = \{s_1 \approx^? t_1, \dots, s_n \approx^? t_n\}$ be a multiset of equational terms. If there is no multiset narrowing derivation $S \rightsquigarrow_{\sigma, \mathcal{R}, M}^* \top$ for any substitution σ , then there is no \mathcal{R} -unifier σ satisfying $s_k\sigma \approx_{\mathcal{R}} t_k\sigma$ for all $1 \leq k \leq n$, where \mathcal{R} is viewed as a set of equations. \square

From Lemmas 31 and 32, we have the following theorem of E -unifiability (consisting of multiple goals) by multiset narrowing.

► **Theorem 33.** Let \mathcal{R} be a complete TRS and $S = \{s_1 \approx^? t_1, \dots, s_n \approx^? t_n\}$ be a multiset of equational terms. If all multiset narrowing derivation starting from S terminate, then we can decide whether there is an \mathcal{R} -unifier σ satisfying $s_k\sigma \approx_{\mathcal{R}} t_k\sigma$ for all $1 \leq k \leq n$. \square

Next, we adapt the narrowing discussed in [22] for (ordinary) reachability analysis using multisets of pairs of terms. Given a rewrite system \mathcal{R} and pairs of terms $(s_1, t_1), \dots, (s_n, t_n)$, the purpose of reachability analysis is to determine whether there is a substitution σ such that $s_1\sigma \rightarrow_{\mathcal{R}}^* t_1\sigma \wedge \dots \wedge s_n\sigma \rightarrow_{\mathcal{R}}^* t_n\sigma$. Here, the reachability problem is represented by the multiset $\{(s_k, t_k) \mid 1 \leq k \leq n\}$.

► **Definition 34.** Let S and T be multisets of the pairs of terms. We write $S \rightarrow_{[\mathcal{R}, M_p]} T$ if there exists a pair of terms $(s, t) \in S$ such that $s \rightarrow_{\mathcal{R}} u$ and $T = (S - \{(s, t)\}) \cup \{(u, t)\}$.

► **Definition 35.** A multiset of pairs of terms S is narrowable into a multiset of pairs of terms T if there exists a pair of terms $(s, t) \in S$ and a substitution σ such that

- $s \rightsquigarrow_{\sigma, \mathcal{R}} u$, and
- $T = (S - \{(s, t)\})\sigma \cup \{(u, t\sigma)\}$.

Then, we write $S \rightsquigarrow_{\sigma, \mathcal{R}, M_p}^* T$. Also, we write $S \rightsquigarrow_{\sigma, \mathcal{R}, M_p}^* S'$ if there exists a narrowing derivation $S = S_1 \rightsquigarrow_{\sigma_1, \mathcal{R}, M_p} S_2 \rightsquigarrow_{\sigma_2, \mathcal{R}, M_p} \dots \rightsquigarrow_{\sigma_{n-1}, \mathcal{R}, M_p} S_n = S'$ such that $\sigma = \sigma_{n-1} \circ \dots \circ \sigma_2 \circ \sigma_1$. If $n = 1$, then $\sigma = \varepsilon$.

Intuitively, $S \rightarrow_{[\mathcal{R}, M_p]} T$ if T is obtained by replacing one pair of elements (s, t) in S with (u, t) using $s \rightarrow_{\mathcal{R}} u$. Only the first element in a pair can be rewritten by \mathcal{R} , while the second element serves as a target and is intact for $\rightarrow_{[\mathcal{R}, M_p]}$ -steps. Meanwhile, $S \rightsquigarrow_{\sigma, \mathcal{R}, M_p} T$ if T is obtained by replacing one pair of elements (s, t) in S with $(u, t\sigma)$ from $s \rightsquigarrow_{\sigma, \mathcal{R}} u$ and then applying the narrowing substitution to the remaining multiset $S - \{(s, t)\}$.

In our Isabelle/HOL formalization, for the multiset reduction in Definition 34, we use the following inductive set in Isabelle such a way that $S \rightarrow_{[\mathcal{R}, M_p]} T$ iff $(S, T) \in \text{multiset_pair_reduction_step}$. (Here, \mathcal{R} is implicitly included as a parameter of `multiset_pair_reduction_step` in the locale.)

inductive_set multiset_pair_reduction_step where

" $(s, t) \in \# S \wedge T = (S - \{\#(s, t)\} + \{\#(u, t)\}) \wedge (s, u) \in \text{rstep } \mathcal{R} \Rightarrow (S, T) \in \text{multiset_pair_reduction_step}$ "

Similarly, for the multiset narrowing in Definition 35, we use the following inductive set in such a way that $S \rightsquigarrow_{\sigma, \mathcal{R}, M_p} T$ iff $(S, T, \sigma) \in \text{multiset_pair_narrowing_step}$.

inductive_set multiset_pair_narrowing_step where

" $(s, t) \in \# S \wedge T = (\text{subst_pairs_multiset } \sigma (S - \{\#(s, t)\}) + \{\#(u, t \cdot \sigma)\}) \wedge (s, u, \sigma) \in \text{narrowing_step} \Rightarrow (S, T, \sigma) \in \text{multiset_pair_narrowing_step}$ "

► **Definition 36.**

- (i) We say that a multiset of pairs of terms $\{(s_k, t_k) \mid 1 \leq k \leq n\}$ is trivially unifiable if $s_k = t_k$ for all $1 \leq k \leq n$.
- (ii) We say that a multiset of pairs of terms $\{(s_k, t_k) \mid 1 \leq k \leq n\}$ is syntactically unifiable with a substitution θ if $s_k\theta = t_k\theta$ for all $1 \leq k \leq n$.
- (iii) We say that a substitution τ is a solution of the reachability problem represented by $S = \{(s_1, t_1), \dots, (s_n, t_n)\}$ if $s_1\tau \rightarrow_{\mathcal{R}}^* t_1\tau \wedge \dots \wedge s_n\tau \rightarrow_{\mathcal{R}}^* t_n\tau$.

► **Lemma 37.** Let \mathcal{R} be a TRS and $S = \{(s_1, t_1), \dots, (s_n, t_n)\}$ be a multiset of pairs of terms. If $S \rightarrow_{[\mathcal{R}, M_p]}^* S'$ and S' is trivially unifiable, then $s_1 \rightarrow_{\mathcal{R}}^* t_1 \wedge \dots \wedge s_n \rightarrow_{\mathcal{R}}^* t_n$. ☑

Proof. We proceed by induction on the number of $\rightarrow_{[\mathcal{R}, M_p]}^*$ -steps in $S \rightarrow_{[\mathcal{R}, M_p]}^* S'$. The base case is trivial, i.e., $S = S'$. For the inductive case, consider $S \rightarrow_{[\mathcal{R}, M_p]} U$ and $U \rightarrow_{[\mathcal{R}, M_p]}^* S'$. From $S \rightarrow_{[\mathcal{R}, M_p]} U$, we have some $(s, t) \in S$, $s \rightarrow_{\mathcal{R}} u$, and $U = (S - \{(s, t)\}) \cup \{(u, t)\}$. By the induction hypothesis, for all pairs (v, w) in U , we have $v \rightarrow_{\mathcal{R}}^* w$. This means that $u \rightarrow_{\mathcal{R}}^* t$ and for all pairs $(v', w') \in (S - \{(s, t)\})$, we have $v' \rightarrow_{\mathcal{R}}^* w'$. Therefore, it remains to show that $s \rightarrow_{\mathcal{R}}^* t$, which is obvious from $s \rightarrow_{\mathcal{R}} u$ and $u \rightarrow_{\mathcal{R}}^* t$. ◀

► **Proposition 38.** Let \mathcal{R} be a TRS and $S = \{(s_1, t_1), \dots, (s_n, t_n)\}$ be a multiset of pairs of terms. If $S \rightsquigarrow_{\sigma, \mathcal{R}, M_p}^* S'$ and S' is syntactically unifiable with θ , then $\theta \circ \sigma$ is a solution of the reachability problem represented by $S = \{(s_1, t_1), \dots, (s_n, t_n)\}$. ☑

Proof. Suppose $S \rightsquigarrow_{\sigma, \mathcal{R}, M_p}^* S'$. Then, we have $S\sigma \rightarrow_{[\mathcal{R}, M_p]}^* S'$ by adapting the proof of Lemma 4. Also, the relation $\rightarrow_{[\mathcal{R}, M_p]}^*$ is closed under substitutions, which can be shown using induction on the number of $\rightarrow_{[\mathcal{R}, M_p]}$ -steps. Then, we have $(S\sigma)\theta \rightarrow_{[\mathcal{R}, M_p]}^* S'\theta$, where $S'\theta$ is trivially unifiable. Thus, the conclusion follows by Lemma 37. ◀

The above proposition provides a sufficient condition of satisfying a reachability problem consisting of multiple goals using multiset narrowing on multisets of pairs of terms. However, it alone does not provide the decidability of a reachability problem consisting of multiple goals.

Next, we consider multiset narrowing on multisets of equational terms again (instead of multisets of pairs of terms) for ordinary reachability problems. Similarly to Definition 36(iii), we say that a substitution σ is a *solution* of the reachability problem represented by a multiset $S = \{s_1 \approx^? t_1, \dots, s_n \approx^? t_n\}$ if $s_1\sigma \rightarrow_{\mathcal{R}}^* t_1\sigma \wedge \dots \wedge s_n\sigma \rightarrow_{\mathcal{R}}^* t_n\sigma$. If σ is a *solution* of the reachability problem represented by S , then we say that the reachability problem represented by S is *satisfiable*. Otherwise, if there is no solution of the reachability problem represented by S , then we say that the reachability problem represented by S is *infeasible*.

► **Lemma 39.** *Let \mathcal{R} be a TRS and $S = \{s_1 \approx^? t_1, \dots, s_n \approx^? t_n\}$ be a multiset of equational terms such that $s_1 \rightarrow_{\mathcal{R}}^* t_1 \wedge \dots \wedge s_n \rightarrow_{\mathcal{R}}^* t_n$ and each t_k , $1 \leq k \leq n$, is a normal form of \mathcal{R} . Then, $S \rightarrow_{[\mathcal{R}, M]}^* \top$. ☑*

► **Lemma 40.** *Let $S = \{s_1 \approx^? t_1, \dots, s_n \approx^? t_n\}$ be a multiset of equational terms. If there is no multiset narrowing derivation $S \rightsquigarrow_{\sigma, \mathcal{R}, M}^* \top$ for any substitution σ , then there is no normal substitution θ satisfying $S\theta \rightarrow_{[\mathcal{R}, M]}^* \top$. ☑*

► **Lemma 41.** *Let \mathcal{R} be a semi-complete TRS and $S = \{s_1 \approx^? t_1, \dots, s_n \approx^? t_n\}$ be a multiset of equational terms, where each t_k , $1 \leq k \leq n$, is a strongly irreducible term. If there is no multiset narrowing derivation $S \rightsquigarrow_{\sigma, \mathcal{R}, M}^* \top$ for any substitution σ , then the reachability problem represented by S is infeasible. ☑*

Proof. Assume that there is no multiset narrowing derivation $S \rightsquigarrow_{\sigma, \mathcal{R}, M}^* \top$ for any substitution σ . Then, by Lemma 40, there is no normal substitution θ satisfying $S\theta \rightarrow_{[\mathcal{R}, M]}^* \top$. Now, suppose, towards a contradiction, that the reachability problem represented by S is satisfiable. Then, there is a substitution τ such that $s_1\tau \rightarrow_{\mathcal{R}}^* t_1\tau \wedge \dots \wedge s_n\tau \rightarrow_{\mathcal{R}}^* t_n\tau$. Since \mathcal{R} is weakly normalizing, there is a normal substitution τ' of τ such that $s_k\tau' \xrightarrow{\mathcal{R}}^* s_k\tau \rightarrow_{\mathcal{R}}^* t_k\tau \rightarrow_{\mathcal{R}}^* t_k\tau'$ for all $1 \leq k \leq n$. We see that each $t_k\tau'$, $1 \leq k \leq n$, is in normal form (w.r.t. \mathcal{R}) because t_k is a strongly irreducible term and τ' is a normal substitution. Since \mathcal{R} is confluent and each $t_k\tau'$, $1 \leq k \leq n$, is in normal form (w.r.t. \mathcal{R}), we have $s_k\tau' \rightarrow_{\mathcal{R}}^* t_k\tau'$ for all $1 \leq k \leq n$. Now, we have $S\tau' \rightarrow_{[\mathcal{R}, M]}^* \top$ by Lemma 39, which is the required contradiction. ◀

► **Lemma 42.** *Let \mathcal{R} be a semi-complete TRS and $S = \{s_1 \approx^? t_1, \dots, s_n \approx^? t_n\}$ be a multiset of equational terms, where each t_k , $1 \leq k \leq n$, is a strongly irreducible term. If $S \rightsquigarrow_{\sigma, \mathcal{R}, M}^* \top$ for some substitution σ , then the reachability problem represented by S is satisfiable. ☑*

Now, we have the following decidability result of a reachability problem (consisting of multiple goals) using multiset narrowing on multisets of equational terms by Lemmas 41 and 42.

► **Theorem 43.** *Let \mathcal{R} be a semi-complete TRS and $S = \{s_1 \approx^? t_1, \dots, s_n \approx^? t_n\}$ be a multiset of equational terms, where each t_k , $1 \leq k \leq n$, is a strongly irreducible term. If all multiset narrowing derivations starting from S terminate, then we can decide whether the reachability problem represented by S is satisfiable or not (i.e., infeasible). ☑*

7 Related Work and Discussion

In this paper, we have focused on an Isabelle/HOL formalization of narrowing and multiset narrowing. There are other important narrowing techniques, such as *basic* [23], *conditional* [6], *constrained* [8], *nominal* [2], and *folding variant* [12] narrowing, which have not been discussed in this paper. For E -unification and reachability analysis, there are also existing narrowing-based computational tools (not using an Isabelle/HOL proof assistant); in particular, see the *Maude* system [11] using folding variant narrowing.

Meanwhile, multiset narrowing presented in this paper provides a natural method for multiset reachability analysis. Note that there are some limitations on simulating multiset rewriting (resp. multiset narrowing) using ordinary rewriting (resp. ordinary narrowing). Consider, for example, $S = \{s_1, s_2, s_3, s_4\}$ and $T = \{t_1, s_2, s_3, s_4\}$, where all s_i , $1 \leq i \leq 4$, are distinct, $s_1 \rightarrow_{\mathcal{R}} t_1$, and thus $S \rightarrow_{[\mathcal{R}, M]} T$. If we simulate the multiset rewriting $S \rightarrow_{[\mathcal{R}, M]} T$ using ordinary rewriting with a new function symbol \bar{f} , we have to consider the following cases: (1) $\bar{f}(s_1, s_2, s_3, s_4) \rightarrow_{\mathcal{R}} \bar{f}(t_1, s_2, s_3, s_4)$, (2) $\bar{f}(s_2, s_1, s_3, s_4) \rightarrow_{\mathcal{R}} \bar{f}(s_2, t_1, s_3, s_4)$, \dots , (24) $\bar{f}(s_4, s_3, s_2, s_1) \rightarrow_{\mathcal{R}} \bar{f}(s_4, s_3, s_2, t_1)$. Here, $S \rightarrow_{[\mathcal{R}, M]} T$ is a compact representation of the above 24 cases. Similarly, let $S = \{s_1, s_2, s_3, s_4\}$ as above and $U = \{u_1, u_2, u_3, u_4\}$, where all u_i , $1 \leq i \leq 4$, are distinct. Now, determining whether $S \rightsquigarrow_{\sigma, \mathcal{R}, M}^* U$ using some σ exists is a compact representation of determining whether one of the following 24 cases of ordinary narrowing using some σ_i exists with a new function symbol \bar{g} : (1) $\bar{g}(s_1, s_2, s_3, s_4) \rightsquigarrow_{\sigma_1, \mathcal{R}}^* \bar{g}(u_1, u_2, u_3, u_4)$, (2) $\bar{g}(s_1, s_2, s_3, s_4) \rightsquigarrow_{\sigma_2, \mathcal{R}}^* \bar{g}(u_2, u_1, u_3, u_4)$, \dots , (24) $\bar{g}(s_1, s_2, s_3, s_4) \rightsquigarrow_{\sigma_{24}, \mathcal{R}}^* \bar{g}(u_4, u_3, u_2, u_1)$. Here, without using multiset narrowing, one may have to create 24 (ordinary) narrowing trees in the worst case (with possibly many duplicated narrowing steps) for the corresponding multiset reachability problem.

When considering multiset reachability problems by determining whether a substitution σ exists such that $S\sigma \rightarrow_{[\mathcal{R}, M]}^* U$, multiset narrowing provides a simple and compact sufficient condition of satisfying the multiset reachability problem, i.e., $S \rightsquigarrow_{\sigma, \mathcal{R}, M}^* U$ using some σ .

8 Conclusion

Although narrowing plays an important role in equational unification and reachability analysis, formalization of narrowing and its related results on equational unification and reachability analysis has not been much done in the proof assistants. We have presented a new Isabelle/HOL formalization of narrowing and multiset narrowing for E -unifiability and (multiset) reachability analysis. The results discussed in this paper are built on `IsaFoR` (Isabelle/HOL Formalization of Rewriting) [1].

Given a semi-complete rewrite system \mathcal{R} representing E and two terms s and t , we show a formalized correctness proof that if all narrowing derivations starting from $s \approx^? t$ terminate (or simply \rightsquigarrow terminates), then we can decide whether s and t are E -unifiable.

We have also presented multiset narrowing and its formalization for multiset reachability analysis. Our multiset narrowing is generic in the sense that it encapsulates (the ordinary) rewriting and narrowing for multiset rewriting and multiset narrowing. It is also applicable to E -unifiability/ E -unification and reachability problems consisting of multiple goals. In particular, given a complete rewrite system \mathcal{R} , it provides a complete method for \mathcal{R} -unifiability problems consisting of multiple goals, where \mathcal{R} is viewed as a set of equations. Furthermore, if \mathcal{R} is semi-complete and the right-hand sides of multiple goals in a reachability problem are strongly irreducible terms, then it provides a decision procedure for the reachability problem if it terminates. (Recall that if \mathcal{R} is complete, then \mathcal{R} is semi-complete, but not vice versa.)

Finally, much work still remains ahead. In particular, developing and formalizing parallel multiset rewriting/narrowing is a potential future research direction. It is also interesting to see whether multiset narrowing encapsulating other rewriting and narrowing strategies (such as *basic narrowing* [23]) can improve the multiset narrowing discussed in this paper.

References

- 1 An Isabelle/HOL Formalization of Rewriting for Certified Tool Assertions. Computational Logic group at the University of Innsbruck, <http://c1-informatik.uibk.ac.at/isafor/>.


- 2 Mauricio Ayala-Rincón, Maribel Fernández, and Daniele Nantes-Sobrinho. Nominal narrowing. In Delia Kesner and Brigitte Pientka, editors, *1st International Conference on Formal Structures for Computation and Deduction, FSCD 2016, June 22-26, 2016, Porto, Portugal*, volume 52 of *LIPICs*, pages 11:1–11:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPICs.FSCD.2016.11.
- 3 Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, Cambridge, UK, 1998.
- 4 Franz Baader and Wayne Snyder. Unification Theory. In *Handbook of Automated Reasoning*, Volume I, chapter 8, pages 445–532. Elsevier, Amsterdam, 2001.
- 5 Clemens Ballarin. Tutorial to Locales and Locale Interpretation. URL: <http://isabelle.in.tum.de/doc/locales.pdf>.
- 6 Alexander Bockmayr. *Contributions to the Theory of Logic-Functional Programming*. PhD thesis, Fakultät für Informatik, Universität Karlsruhe, 1990.
- 7 Andrew Cholewa, Santiago Escobar, and José Meseguer. Constrained narrowing for conditional equational theories modulo axioms. *Sci. Comput. Program.*, 112:24–57, 2015. doi:10.1016/J.SCICP.2015.06.001.
- 8 Hubert Comon and Claude Kirchner. Constraint Solving on Terms. In Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, Hubert Comon, Claude Marché, and Ralf Treinen, editors, *Constraints in Computational Logics: Theory and Applications International Summer School, CCL '99 Gif-sur-Yvette, France, September 5–8, 1999 Revised Lectures*, pages 47–103. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001. doi:10.1007/3-540-45406-3_2.
- 9 Hubert Comon-Lundh and Stéphanie Delaune. The Finite Variant Property: How to Get Rid of Some Algebraic Properties. In Jürgen Giesl, editor, *Term Rewriting and Applications, 16th International Conference, RTA 2005, Nara, Japan, April 19-21, 2005, Proceedings*, volume 3467 of *Lecture Notes in Computer Science*, pages 294–307. Springer, 2005. doi:10.1007/978-3-540-32033-3_22.
- 10 Nachum Dershowitz and David A. Plaisted. Rewriting. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning (in 2 volumes)*, pages 535–610. Elsevier and MIT Press, 2001.
- 11 Francisco Durán, Steven Eker, Santiago Escobar, Narciso Martí-Oliet, José Meseguer, Rubén Rubio, and Carolyn L. Talcott. Equational Unification and Matching, and Symbolic Reachability Analysis in Maude 3.2 (System Description). In Jasmin Blanchette, Laura Kovács, and Dirk Pattinson, editors, *Automated Reasoning - 11th International Joint Conference, IJCAR 2022, Haifa, Israel, August 8-10, 2022, Proceedings*, volume 13385 of *Lecture Notes in Computer Science*, pages 529–540. Springer, 2022. doi:10.1007/978-3-031-10769-6_31.
- 12 Santiago Escobar, Ralf Sasse, and José Meseguer. Folding Variant Narrowing and Optimal Variant Termination. In Peter Csaba Ölveczky, editor, *Rewriting Logic and Its Applications - 8th International Workshop, WRLA 2010, Held as a Satellite Event of ETAPS 2010, Paphos, Cyprus, March 20-21, 2010, Revised Selected Papers*, volume 6381 of *Lecture Notes in Computer Science*, pages 52–68. Springer, 2010. doi:10.1007/978-3-642-16310-4_5.
- 13 M. Fay. First-Order Unification in Equational Theories. In *Fourth International Workshop on Automated Deduction, Austin, Texas, Proceedings*, pages 161–167, 1979.
- 14 Guillaume Feuillade, Thomas Genet, and Valérie Viet Triem Tong. Reachability Analysis over Term Rewriting Systems. *J. Autom. Reason.*, 33(3-4):341–383, 2004. doi:10.1007/S10817-004-6246-0.
- 15 Jean H Gallier and Wayne Snyder. Complete sets of transformations for general E-unification. *Theoretical Computer Science*, 67(2-3):203–260, 1989. doi:10.1016/0304-3975(89)90004-2.
- 16 Michael Hanus. CHR(Curry): Interpretation and Compilation of Constraint Handling Rules in Curry. In Enrico Pontelli and Tran Cao Son, editors, *Practical Aspects of Declarative Languages - 17th International Symposium, PADL 2015, Portland, OR, USA, June 18-19, 2015. Proceedings*, volume 9131 of *Lecture Notes in Computer Science*, pages 74–89. Springer, 2015. doi:10.1007/978-3-319-19686-2_6.

- 17 Nao Hirokawa, Aart Middeldorp, and Christian Sternagel. A New and Formalized Proof of Abstract Completion. In Gerwin Klein and Ruben Gamboa, editors, *Interactive Theorem Proving - 5th International Conference, ITP 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, volume 8558 of *Lecture Notes in Computer Science*, pages 292–307. Springer, 2014. doi:10.1007/978-3-319-08970-6_19.
- 18 Jean-Marie Hullot. Canonical forms and unification. In Wolfgang Bibel and Robert A. Kowalski, editors, *5th Conference on Automated Deduction, Les Arcs, France, July 8-11, 1980, Proceedings*, volume 87 of *Lecture Notes in Computer Science*, pages 318–334. Springer, 1980. doi:10.1007/3-540-10009-1_25.
- 19 Hélène Kirchner. On the Use of Constraints in Automated Deduction. In Andreas Podelski, editor, *Constraint Programming: Basics and Trends, Châtillon Spring School, Châtillon-sur-Seine, France, May 16 - 20, 1994, Selected Papers*, volume 910 of *Lecture Notes in Computer Science*, pages 128–146. Springer, 1994. doi:10.1007/3-540-59155-9_8.
- 20 John W. Lloyd. *Foundations of Logic Programming, 3rd Edition*. Springer, 2012.
- 21 Salvador Lucas and Raúl Gutiérrez. Use of logical models for proving infeasibility in term rewriting. *Inf. Process. Lett.*, 136:90–95, 2018. doi:10.1016/J.IPL.2018.04.002.
- 22 José Meseguer and Prasanna Thati. Symbolic reachability analysis using narrowing and its application to verification of cryptographic protocols. *High. Order Symb. Comput.*, 20(1-2):123–160, 2007. doi:10.1007/S10990-007-9000-6.
- 23 Aart Middeldorp and Erik Hamoen. Completeness results for basic narrowing. *Appl. Algebra Eng. Commun. Comput.*, 5:213–253, 1994. doi:10.1007/BF01190830.
- 24 Aart Middeldorp, Satoshi Okui, and Tetsuo Ida. Lazy Narrowing: Strong Completeness and Eager Variable Elimination. *Theor. Comput. Sci.*, 167(1&2):95–130, 1996. doi:10.1016/0304-3975(96)00071-0.
- 25 Robert Nieuwenhuis. Decidability and Complexity Analysis by Basic Paramodulation. *Inf. Comput.*, 147(1):1–21, 1998. doi:10.1006/INCO.1998.2730.
- 26 Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.
- 27 Christian Sternagel and Akihisa Yamada. Reachability Analysis for Termination and Confluence of Rewriting. In Tomás Vojnar and Lijun Zhang, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 25th International Conference, TACAS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part I*, volume 11427 of *Lecture Notes in Computer Science*, pages 262–278. Springer, 2019. doi:10.1007/978-3-030-17462-0_15.
- 28 Prasanna Thati and José Meseguer. Complete Symbolic Reachability Analysis Using Back-and-Forth Narrowing. In José Luiz Fiadeiro, Neil Harman, Markus Roggenbach, and Jan J. M. M. Rutten, editors, *Algebra and Coalgebra in Computer Science: First International Conference, CALCO 2005, Swansea, UK, September 3-6, 2005, Proceedings*, volume 3629 of *Lecture Notes in Computer Science*, pages 379–394. Springer, 2005. doi:10.1007/11548133_24.
- 29 Emanuele Viola. E-unifiability via Narrowing. In Antonio Restivo, Simona Ronchi Della Rocca, and Luca Roversi, editors, *Theoretical Computer Science, 7th Italian Conference, ICTCS 2001, Torino, Italy, October 4-6, 2001, Proceedings*, volume 2202 of *Lecture Notes in Computer Science*, pages 426–438. Springer, 2001. doi:10.1007/3-540-45446-2_27.
- 30 Akihiro Yamamoto. Completeness of extended unification based on basic narrowing. In Koichi Furukawa, Hozumi Tanaka, and Tetsunosuke Fujisaki, editors, *Logic Programming '88*, pages 1–10, Berlin, Heidelberg, 1989. Springer Berlin Heidelberg.

Formalizing the Cholesky Factorization Theorem

Carl Kwan ✉ 

The University of Texas at Austin, TX, United States of America

Warren A. Hunt Jr. ✉ 

The University of Texas at Austin, TX, United States of America

Abstract

We present a formal proof of the Cholesky Factorization Theorem, a fundamental result in numerical linear algebra, by verifying formally a Cholesky decomposition algorithm in ACL2. Our mechanical proof of correctness is largely automatic for two main reasons: (1) we employ a derivation which involves partitioning the matrix to obtain the desired result; and (2) we provide an inductive invariant for the Cholesky decomposition algorithm. To formalize (1), we build support for reasoning about partitioned matrices. This is a departure from how typical numerical linear algebra algorithms are presented, i.e. via excessive indexing. To enable (2), we build a new recursive recognizer for a matrix to be Cholesky decomposable and mathematically prove that the recognizer is indeed necessary and sufficient. Guided by the recognizer, ACL2 automatically inducts and verifies the Cholesky decomposition algorithm. We also present our ACL2-based formalization of the decomposition algorithm itself, and discuss how to bridge the gap between verifying a decomposition algorithm and proving the Cholesky Factorization Theorem. To our knowledge, this is the first formalization of the Cholesky Factorization Theorem.

2012 ACM Subject Classification Theory of computation → Automated reasoning; Mathematics of computing → Computations on matrices

Keywords and phrases Numerical linear algebra, Cholesky factorization theorem, Matrix decomposition, Automated reasoning, ACL2

Digital Object Identifier 10.4230/LIPIcs.ITP.2024.25

Supplementary Material *Software (Source Code)*: <https://github.com/ac12/ac12/tree/master/books/projects/cholesky>, archived at `swh:1:dir:59ed119089ee943a78fe019c760fc4f75046e663`

Funding This work was supported in part by Intel Corporation and Amazon Science.

Acknowledgements We would like to thank Robert van de Geijn, Margaret Myers, and the anonymous reviewers for their helpful comments and feedback.

1 Introduction

We present an ACL2-based formalization of the Cholesky Factorization Theorem. Our approach implements a Cholesky decomposition algorithm, `CHOL`, in the ACL2 logic, and we verify its correctness using the ACL2 theorem prover. Our formalization is built on existing ACL2 libraries and theories for basic vector and matrix operations (e.g. addition, multiplication, transpose, etc.), but embedding a Cholesky decomposition algorithm required a significant extension over existing theories. In addition to building support for a partitioned matrix environment and functions for accessing the lower triangular part of a matrix, we also had to develop alternate definitions for matrix operations (e.g. multiplication) and verify them against existing definitions.

We base our Cholesky formalization on the Formal Linear Algebra Methods Environment (FLAME) [8]. FLAME is an approach to systematically deriving numerical linear algebra algorithms and proving¹ them correct. We follow a FLAME derivation for a Cholesky

¹ FLAME is “formal” in the systematic sense, not in the theorem proving sense.



25:2 Formalizing the Cholesky Factorization Theorem

decomposition algorithm but modify the algorithm to better suit ACL2's strength in recursion and induction. The advantage to using the FLAME approach is that it represents linear algebra algorithms in terms of operations on components of a matrix's partitioned representation. One common pitfall with how typical matrix algorithms are presented is the over-reliance on indexing. Intricate indices are a common cause of bugs in programs. Introducing indices for matrix / vector entries can also introduce numerous variables causing formal and automated processes to become intractable. Instead, a partitioned representation of a matrix abstracts away details that are irrelevant to the operations of interest. Another advantage to using FLAME's partitioned representation is that it exposes loop invariants. This enables us to more readily derive inductive invariants and verify the correctness of our Cholesky decomposition algorithm.

One departure from typical proofs of the Cholesky Factorization Theorem, including the one from FLAME, is that we develop a variant of *Sylvester's criterion*, a characterization for symmetric matrices to be positive definite, for use as a hypothesis in our main result. Sylvester's criterion states that a symmetric matrix is positive definite iff its principal leading submatrices have positive determinants. However, determinants do not readily lend themselves to numerical computation. Instead, we look at the diagonal of each principal leading submatrix and posit their positivity. The advantage of using this definition of symmetric positive definite is that it is recursive, amenable to ACL2 formalization, and helps automate the ACL2 proof of correctness for the decomposition algorithm. By correctness, we mean the following:

► **Theorem 1.** *Let A be a symmetric positive definite matrix. Let L be the lower triangular part of $\text{CHOL}(A)$. Then $A = LL^T$.*

Theorem 1 permits us to prove the Cholesky Factorization Theorem.

► **Theorem 2 (Cholesky Factorization Theorem).** *If A is a symmetric positive definite matrix, then $A = LL^T$ for some lower triangular matrix L .*

The Cholesky Factorization Theorem states that symmetric positive definite matrices can be decomposed into the product of a lower triangular matrix and its transpose. While the two theorems are similar, their differences are enhanced when viewed through the lens of ACL2. One distinction is that Theorem 2 is a quantified statement and ACL2 support for quantifiers is limited. Propositional statements about functions, such as Theorem 1, is typically the preferred approach for reasoning in ACL2. The discussion in this paper focuses on issues such as these and our ACL2 formalization.

There are two primary advantages to our choice of ACL2.² First, ACL2 is highly automated with extensive support for rewriting. To discharge the proof of Theorem 1 in ACL2, the only knowledge necessary is the matrix partitioning and a recognizer for the class of matrices on which the algorithm is expected to operate. Our efforts required relatively few user-defined hints, lemmas, or events. Second, ACL2 supports the execution of its functions via an underlying Lisp interpreter defined within the theorem prover logic. Few theorem provers are capable of natively executing formalized functions. This makes verifying a *computational* algorithm such as the Cholesky decomposition in ACL2 particularly meaningful.

² Technically, we use ACL2(r), a version of ACL2 with support for real numbers via nonstandard analysis, which is only necessary for taking square roots in the Cholesky decomposition.

The Cholesky decomposition is fundamental to numerical linear algebra and scientific computing. For example, a common problem involves solving linear systems of the form $Ax = b$, where x and b are vectors of dimension compatible with A . If A is symmetric positive definite, then it has a Cholesky decomposition $A = LL^T$ and we obtain $LL^T x = Ax = b$. Finding x can be efficiently done by first solving $Ly = b$ via forwards substitution and then $L^T x = y$ via backwards substitution. Another practical application of Cholesky is in solving the linear least squares problem $\|y - B\hat{x}\|_2 = \min_{x \in \mathbb{R}^n} \|y - Bx\|_2$ for \hat{x} . By setting $A = B^T B$ and $b = B^T y$, we can find the solution to the linear least squares problem by solving $A\hat{x} = b$ in much the same way as before. In addition to these basic applications, Cholesky can be used to find matrix inverses, perform Monte Carlo simulations, and optimize quadratic forms. This makes Cholesky a vital tool in areas such as engineering, finance, and machine learning.

2 Related Work

To our knowledge, there is no other formal proof for the Cholesky Factorization Theorem. There are a few theorem prover formalizations of other decomposition algorithms. In ACL2, there is a verified executable implementation of an LU decomposition algorithm [13]. Lean’s mathlib contains a formalization of LDL decomposition, but the matrix functions used in the LDL decomposition are not computable [15]. In Isabelle’s Archive of Formal Proofs, there is also a formalization of Schur decomposition [20]. Basic matrix theories have long been formalized using theorem provers, including Coq [17], HOL4 [18], HOL Light [9] and the aforementioned ACL2 [10, 6, 14], Lean [16], and Isabelle [19]. Notably, ACL2’s and Isabelle’s theories of basic matrices provide executable operations.

Our work is inspired heavily by FLAME. While FLAME is “formal” in that it systematically derives numerical algorithms, no formal method or verification is involved with FLAME. The relevance of FLAME to our work is that FLAME introduces a partitioned matrix environment (PME), which enables us to approach Cholesky without the burden of indices. Our derivation is similar to FLAME’s in that it begins with a PME, which naturally leads to a recursive Cholesky variant. However, FLAME’s algorithm is loop based. The FLAME approach to systematically proving the correctness of its algorithms is to identify loop invariants. Since our Cholesky algorithm is recursive, we perform an analogous analysis to guide the verification of our Cholesky algorithm, but with induction invariants. Note a loop-invariant verification approach for a loop-based Cholesky algorithm may also be possible in ACL2 using ACL2’s analogue of Common Lisp loops [2]. Because of ACL2’s long tradition with recursion and the natural correspondence between the derivation and a recursive Cholesky algorithm, we opted to directly verify the recursive algorithm.

3 Deriving a Verifiable Cholesky Decomposition Algorithm

A core idea behind FLAME is to recast algorithms in terms of operations on components of a matrix’s partitioned form. This is called a *partitioned matrix environment* (PME). PME is meant to make linear algebra code more intelligible, and enable the systematic derivation and pen-and-paper proofs of numerical linear algebra algorithms. However, PME also lends itself well to developing recursive matrix algorithms and induction proofs of their correctness. To demonstrate this, we derive a Cholesky decomposition algorithm in this section and verify it formally in Section 4.

25:4 Formalizing the Cholesky Factorization Theorem

Let lower-case Greek letters (e.g. α) denote real numbers, lower-case Latin letters (e.g. v) denote vectors, and upper-case Latin letters (e.g. A) denote matrices. Since Cholesky only deals with symmetric matrices, all matrices will be square. For ease of notation, assume any vectors and matrices in a posed expression are compatible (e.g. if $Ax = b$ and A is $n \times n$, then x and b are $n \times 1$).

Given a (real) symmetric positive definite matrix A , i.e. $A = A^T$ and $v^T Av > 0$ for all nonzero v , a Cholesky decomposition for A is a lower triangular matrix L such that $A = LL^T$. To derive the desired Cholesky decomposition algorithm, partition A and L as follows:

$$A := \left(\begin{array}{c|c} \alpha_{11} & a_{12}^T \\ \hline a_{21} & A_{22} \end{array} \right), \quad L := \left(\begin{array}{c|c} \lambda_{11} & \\ \hline \ell_{21} & L_{22} \end{array} \right).$$

Note that $a_{12} = a_{21}$ since A is symmetric. Moving forward, we drop the “bars” for simplicity. If $A = LL^T$, then

$$\left(\begin{array}{cc} \alpha_{11} & a_{12}^T \\ a_{21} & A_{22} \end{array} \right) = A = LL^T = \left(\begin{array}{cc} \lambda_{11} & \\ \ell_{21} & L_{22} \end{array} \right) \left(\begin{array}{cc} \lambda_{11} & \ell_{21}^T \\ & L_{22}^T \end{array} \right). \quad (1)$$

This is equivalent to

$$\alpha_{11} = \lambda_{11}^2, \quad a_{12}^T = \lambda_{11}\ell_{21}^T, \quad a_{21} = \lambda_{11}\ell_{21}, \quad A_{22} = \ell_{21}\ell_{21}^T + L_{22}L_{22}^T.$$

We want Equation (1) to hold. Since a potential algorithm which computes L is given A , we solve for the components of L :

$$\lambda_{11} = \pm\sqrt{\alpha_{11}}, \quad \ell_{21} = a_{21}\lambda_{11}^{-1}, \quad L_{22}L_{22}^T = A_{22} - \ell_{21}\ell_{21}^T.$$

For our purposes, we pick $\lambda_{11} = \sqrt{\alpha_{11}}$. Note that $L_{22}L_{22}^T$ is a Cholesky decomposition for $A_{22} - \ell_{21}\ell_{21}^T$. This suggests a algorithm which updates α_{11} and a_{21} , and recurses on $A_{22} - \ell_{21}\ell_{21}^T$. Indeed, Algorithm 1 computes a Cholesky decomposition. The three “if” branches handle base cases where the matrix is empty or only α_{11} and a_{21} are updated. The recursive step updates A_{22} as well. Note that the algorithm accepts non-square matrices. Even though we are only interested in the output of the algorithm under symmetric positive definite inputs, we nonetheless deal with the non-square cases in order to simplify the acceptance of our algorithm into the ACL2 logic.

Highlighted in Algorithm 1 are the components α_{11} , a_{21} , a_{12}^T , and A_{22} of A prior to and after their updates in the algorithm. Note that the only component of A that is passed to the recursive call is A_{22} , the “bottom right” part of A . This suggests the remaining components need not be updated anymore. Indeed, highlighted in red are the components of A that still need to be updated. Prior to entering the main body of the algorithm, all components still need to be updated. But after the updates to α_{11} , a_{21} , and A_{22} , the only component that still needs to be updated is A_{22} . The other components are already in Cholesky decomposition form. As the recursive algorithm progresses, “layers” of the matrix are replaced by its Cholesky decomposition.³ Visually, this progression is represented in Figure 1. Step (1) represents a matrix prior to the updates in a recursive iteration. Green indicates portions of the matrix that are already Cholesky decomposed and red indicates portions of the matrix that still need to be updated. Step (2) represents the matrix within the main body of a recursive iteration of Algorithm 1. Purple indicates the portions of the

³ This also means Algorithm 1 computes a Cholesky decomposition in place.

■ **Algorithm 1** A recursive Cholesky decomposition algorithm.

```

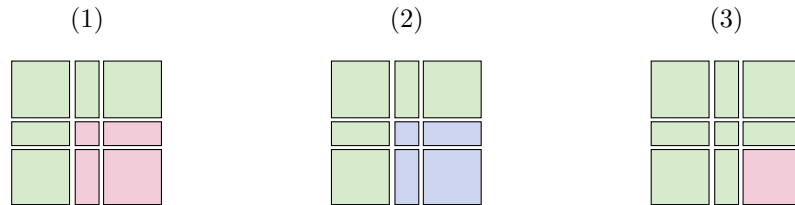
procedure CHOL( $A \in \mathbb{R}^{n \times m}$ )
  Partition  $A = \begin{pmatrix} \alpha_{11} & a_{12}^T \\ a_{21} & A_{22} \end{pmatrix}$ 
  ▷ If  $n, m > 1$ , then  $\alpha \in \mathbb{R}$ ,  $a_{21} \in \mathbb{R}^{(n-1) \times 1}$ ,  

 $a_{12}^T \in \mathbb{R}^{1 \times (m-1)}$ ,  $A_{22} \in \mathbb{R}^{(n-1) \times (m-1)}$ 

  if  $m = 0$  or  $n = 0$  then ▷ Edge case
    return ( ) ▷ Return an empty matrix
  else if  $n = 1$  then ▷ Base case
    return  $\begin{pmatrix} \sqrt{\alpha_{11}} \\ a_{21}\alpha_{11}^{-1} \end{pmatrix}$ 
  else if  $m = 1$  then ▷ Base case
    return  $\begin{pmatrix} \sqrt{\alpha_{11}} & a_{21}^T \end{pmatrix}$ 
  else ▷ Recursive case
     $\alpha_{11} := \sqrt{\alpha_{11}}$ 
     $a_{21} := a_{21}\alpha_{11}^{-1}$ 
     $A_{22} := A_{22} - a_{21}a_{21}^T$ 
    return  $\begin{pmatrix} \alpha_{11} & a_{12}^T \\ a_{21} & \text{CHOL}(A_{22}) \end{pmatrix}$ 

```

■ **Figure 1** Progress of Algorithm 1: (1) prior to updates; (2) during updates; (3) after updates.



matrix that are being updated. Step (3) represents the matrix after the updates are made. As the algorithm progresses, the proportion of the matrix not yet Cholesky decomposed decreases, until no part of the matrix needs to be updated, at which point the algorithm terminates.

The derivation and visual progression of Algorithm 1 suggests an induction invariant, that is, performing the updates in the algorithm computes a Cholesky decomposition for all matrix components except for the “bottom right”. Since the recursive call operates on a smaller matrix, the procedure eventually terminates and indeed computes a Cholesky decomposition for A . An inductive argument, with induction hypothesis stating essentially $A_{22} - \ell_{21}\ell_{21}^T = L_{22}L_{22}^T$ is Cholesky decomposable, would be sufficient to discharge a proof of the correctness of Algorithm 1, thus providing a roadmap to verifying the Cholesky Factorization Theorem formally.

25:6 Formalizing the Cholesky Factorization Theorem

■ **Table 1** Common ACL2 functions, macros, and other commands used in this paper.

<i>Command</i>	<i>Description</i>
<code>define</code>	Define a function symbol, enforce guard checking, and more
<code>defthm</code>	Name and prove a theorem, e.g. (<code>defthm <-add-1 (< x (add-1 x))</code>)
<code>list</code>	Define a list, e.g. (<code>list 1 2 3</code>) returns (1 2 3)
<code>car</code>	Returns the head of a list, e.g. (<code>car (list 1 2 3)</code>) returns 1
<code>cons</code>	Construct a pair, e.g. (<code>cons 1 (list 2)</code>) returns (1 2)
<code>/</code>	Divide two numbers or return the reciprocal of a number, e.g. (<code>/ 1 2</code>) or (<code>/ 2</code>)
<code>acl2-sqrt</code>	Square root of an ACL2 number, e.g. (<code>acl2-sqrt 2</code>)
<code>b*</code>	Binder for local variables; often used to simplify control flow statements

4 Formally Verifying the Cholesky Factorization Theorem

To verify the Cholesky Factorization Theorem formally we need to demonstrate that every symmetric positive matrix has a Cholesky decomposition. We embed our Cholesky decomposition algorithm into the ACL2 logic, verify it, and apply it to compute a witness for the desired theorem. Table 1 lists some commonly used ACL2 functions, macros, and commands in general. Comprehensive ACL2 documentation is freely available and searchable online [3].

We employ some existing ACL2 primitive matrix functions [10] in order to formalize Algorithm 1, and we define our own functions to support reasoning about decomposition algorithms in general, accessing their results, and executing them. We also formalize alternate definitions for primitive matrix operations and prove them equivalent to the existing ones. Table 2 lists some of these ACL2 matrix functions.

4.1 Formalizing the Decomposition Algorithm

Our ACL2 formalization of Algorithm 1 is shown in Program 1. The `b*` in the definition of `chol` is an ACL2 macro for binding local variables with support for control flow. The first argument to `b*` is a list of “bindings” and the second argument is the ACL2 expression to which the bindings apply. For example, the third binding in Program 1’s `b*` is

```
(alph (car (col-car A)))
```

which declares the local variable `alph` to be equal to `(car (col-car A))`, i.e. the first element of the first column in `A`. The `b*` macro also supports early-exit bindings. For example, the first binding in the same `b*` is

```
((unless (mbt (matrixp A))) (m-empty))
```

which is triggered when `A` is not an ACL2 matrix and an empty matrix (`m-empty`) is returned. The macro `mbt` is logically equivalent to its argument (i.e. `(mbt x)` equals `x`) but immediately evaluates to `t` during runtime (ignoring `x`). This optimization is permitted by guard verification (discussed later). Provided no early exit bindings are triggered, the `b*` expression returns the second argument – in Program 1, this is

```
(row-cons (cons alph a12)
          (col-cons a21 (chol A22)))
```

Note extra edge cases, such as those handling when `A` is not a matrix, appear in Program 1 but not Algorithm 1. In ACL2, logical functions are total, that is, all functions map all objects

■ **Table 2** ACL2 linear algebra functions.

<i>Function</i>	<i>Intended Signature</i>	<i>Description</i>
<code>matrixp</code>	$\mathbb{R}^{n \times m} \rightarrow \{\mathbf{t}, \mathbf{nil}\}$	Matrix recognizer, e.g. (<code>matrixp (list (list 1 0))</code>) returns <code>t</code>
<code>m-emptyp</code>	$\mathbb{R}^{n \times m} \rightarrow \{\mathbf{t}, \mathbf{nil}\}$	Empty matrix recognizer, e.g. (<code>m-emptyp nil</code>) returns <code>t</code>
<code>m-empty</code>	$\{\} \rightarrow \mathbb{R}^{0 \times 0}$	Returns an empty matrix, e.g. (<code>m-empty</code>) returns <code>nil</code>
<code>mzero</code>	$\mathbb{N} \times \mathbb{N} \rightarrow \mathbb{R}^{n \times m}$	Returns a zero matrix, e.g. (<code>mzero 1 2</code>) returns <code>((0 0))</code>
<code>row-car</code>	$\mathbb{R}^{n \times m} \rightarrow \mathbb{R}^m$	Returns the first row of a matrix, e.g. <div style="background-color: #f0f0f0; padding: 2px;"><code>(row-car (list (list 1 2) (list 3 4)))</code></div> returns <code>(1 2)</code>
<code>col-car</code>	$\mathbb{R}^{n \times m} \rightarrow \mathbb{R}^n$	Returns the first column of a matrix, e.g. replacing <code>row-car</code> with <code>col-car</code> in the previous example returns <code>(1 3)</code>
<code>row-cdr</code>	$\mathbb{R}^{n \times m} \rightarrow \mathbb{R}^{(n-1) \times m}$	Remove a matrix's first row, e.g. <div style="background-color: #f0f0f0; padding: 2px;"><code>(row-cdr (list (list 1 2) (list 3 4)))</code></div> returns <code>((3 4))</code>
<code>col-cdr</code>	$\mathbb{R}^{n \times m} \rightarrow \mathbb{R}^{n \times (m-1)}$	Remove a matrix's first column, e.g. replacing <code>row-cdr</code> with <code>col-cdr</code> in the previous example returns <code>((2) (4))</code>
<code>row-cons</code>	$\mathbb{R}^{n \times m} \rightarrow \mathbb{R}^{(n+1) \times m}$	Append a row to a matrix, e.g. <div style="background-color: #f0f0f0; padding: 2px;"><code>(row-cons (list 1 2) (list (list 3 4)))</code></div> returns <code>((1 2) (3 4))</code>
<code>col-cons</code>	$\mathbb{R}^{n \times m} \rightarrow \mathbb{R}^{n \times (m+1)}$	Append a column to a matrix, e.g. <div style="background-color: #f0f0f0; padding: 2px;"><code>(col-cons (list 1 3) (list (list 2) (list 4)))</code></div> returns <code>((1 2) (3 4))</code>
<code>m+</code>	$\mathbb{R}^{n \times m} \times \mathbb{R}^{n \times m} \rightarrow \mathbb{R}^{n \times m}$	Matrix addition, e.g. <div style="background-color: #f0f0f0; padding: 2px;"><code>(m+ (list (list 0 1) (list 2 3)) (list (list 2 3) (list 4 5)))</code></div> returns <code>((2 4) (6 8))</code>
<code>m*</code>	$\mathbb{R}^{n \times m} \times \mathbb{R}^{m \times \ell} \rightarrow \mathbb{R}^{n \times \ell}$	Matrix multiplication, e.g., replacing <code>m+</code> with <code>m*</code> in the previous example returns <code>((4 5) (16 21))</code>
<code>sm*</code>	$\mathbb{R} \times \mathbb{R}^{n \times m} \rightarrow \mathbb{R}^{n \times m}$	Scalar-matrix multiplication, e.g. <div style="background-color: #f0f0f0; padding: 2px;"><code>(sm* 2 (list (list 1 2) (list 3 4)))</code></div> returns <code>((2 4) (6 8))</code>
<code>sv*</code>	$\mathbb{R} \times \mathbb{R}^n \rightarrow \mathbb{R}^n$	Scalar-vector multiplication, e.g. (<code>sv* 2 (list 1 2)</code>) returns <code>(2 4)</code>
<code>out-*</code>	$\mathbb{R}^n \times \mathbb{R}^n \rightarrow \mathbb{R}^{n \times n}$	Outer product, e.g. (<code>out-* (list 1 2) (list 3 4)</code>) returns <code>((3 4) (6 8))</code>
<code>get-L</code>	$\mathbb{R}^{n \times m} \rightarrow \mathbb{R}^{n \times m}$	Get a matrix's lower triangular part, e.g. <div style="background-color: #f0f0f0; padding: 2px;"><code>(get-L (list (list 1 2) (list 3 4)))</code></div> returns <code>((1 0) (3 4))</code>
<code>mtrans</code>	$\mathbb{R}^{n \times m} \rightarrow \mathbb{R}^{m \times n}$	Matrix transpose, e.g. (<code>mtrans (list (list 1 2))</code>) returns <code>((1) (2))</code>

25:8 Formalizing the Cholesky Factorization Theorem

■ **Program 1** ACL2 implementation of a Cholesky decomposition algorithm (Algorithm 1).

```
(define chol ((A matrixp))
:guard (and (equal (col-count A) (row-count A))
            (equal (mtrans A) A))
:measure (and (col-count A) (row-count A)) ...
(mbe
:logic
(b* ( ;; BASE CASES
      ((unless (mbt (matrixp A))) (m-empty)) ;; If A not a matrix, return empty
      ((if (m-empty A) A) ;; If A empty, return A
         (alph (car (col-car A))) ;; alph := "top left" scalar in A
         ((unless (realp alph)) ;; If alph not real, return a zero
            (mzero (row-count A) ;; matrix of the same dimensions
                   (col-count A))) ;; as A
         ((if (<= alph 0)) ;; If alph not positive, return a
            (mzero (row-count A) ;; zero matrix of the same
                   (col-count A))) ;; dimensions as A

      ;; PARTITION
      (a21 (col-car (row-cdr A))) ;; [ alph | a12 ] := A
      (a12 (row-car (col-cdr A))) ;; [ ----- ]
      (A22 (col-cdr (row-cdr A))) ;; [ a21 | A22 ]
      (alph (acl2-sqrt alph)) ;; alph := sqrt(alph)

      ;; BASE CASES
      ((if (m-empty (col-cdr A)) ;; If A is a column, return
          (row-cons (list alph) ;; [ 1 ] [ a1 ] = [ a1 ] = A
                   (sm* (/ alph) ;; [ a2/a1 ] [ a2 ]
                        (row-cdr A)))) ;; [ ... ] [ ... ]
      ((if (m-empty (row-cdr A)) ;; If A is a row, return
          (row-cons (cons alph a12) ;; [ alph a12 ]
                   (m-empty))))

      ;; UPDATE
      (a21 (sv* (/ alph) a21)) ;; a21 := a21 / alph
      (A22 (m+ A22 (sm* -1 (out-* a21 a21)))))) ;; A22 := A22 - a21 * a21T

      ;; RECURSE
      (row-cons (cons alph a12) ;; [ alph | a12 ]
                (col-cons a21 (chol A22)))) ;; [ ----- ]
                ;; [ a21 | CHOL(A22) ]

:exec
(b* ... ) ... ) ...)
```

in the logic. In Program 1, the logic of `chol` is required to handle cases where it is passed non-matrix objects. Extra branches require more computational resources. To alleviate some of this overhead, we can use *guards* to prevent the *execution* of functions on unintended inputs. The macro `define` is a wrapper for `defun` that simplifies common hygienic practices, such as using guards, when introducing new ACL2 functions. In Program 1, the guards for `chol` are

```
(and (equal (col-count A) (row-count A))
      (equal (mtrans A) A))
```

as indicated by the `:guard` key and `(matrixp A)` as indicated by the formal arguments to `chol`. We deploy guard checking to prevent code execution under inappropriate or unexpected circumstances, helping catch potential issues early during program execution and avoiding unintended consequences, thus enhancing the robustness and reliability of ACL2 code. Another advantage of providing guards is it can reduce the computation performed by the Lisp back-end. For example, if `A` is known to be `matrixp`, then we no longer need

to perform the first check in the `b*` macro. If we know A is square, then we can reduce the number of base cases. The `mbe` macro enables a user to introduce a function logically defined by the term passed to the `:logic` key but executed with the code passed to the `:exec` key when the guards are satisfied. Of course, it must be proven that the logical definition and executed code are equivalent under these conditions.

4.2 Modifying Sylvester's Criterion

Recall that Theorems 1 and 2 hypothesize A to be symmetric positive definite. The usual definition for positive definiteness states that $v^T A v > 0$ for all nonzero $v \in \mathbb{R}^n$, which is a quantified statement. In symbols, this is

$$\forall v \in \mathbb{R}^n, v \neq 0 \implies v^T A v > 0.$$

This condition is not optimal for our theorem proving needs. On one hand, variables in ACL2 theorem statements are implicitly universally quantified at the top level. Quantifiers are also further supported via Skolem functions. However, Skolem functions are not executable. We want a recognizer for positive definite matrices to be executable because it can serve as guard for future functions,⁴ and an executable recognizer more readily triggers the automatic rewrite rules which enable us to verify the Cholesky decomposition.

Instead of using the typical definition of positive definiteness, we use a definition which involves looking at the *leading principal submatrices*. Informally, the leading principal submatrices of a matrix A are the “top left” submatrices of size $k \times k$ for $k \in [1, n]$. For example, if a square matrix A is partitioned as

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}$$

and A_{11} is $k \times k$, then A_{11} would be the k -th leading principal submatrix of A . This approach is particularly useful for our computational purposes because it enables us to exploit the block structures of a matrix and restate matrix properties in terms of the same properties on smaller submatrices, such as determinants by Schur's formula.

► **Proposition 3** (Schur's formula). *Let $A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}$. Then*

$$\det(A) = \det(A_{11}) \det(A_{22} - A_{21} A_{11}^{-1} A_{12})$$

if A_{11} is invertible. Similarly,

$$\det(A) = \det(A_{22}) \det(A_{11} - A_{12} A_{22}^{-1} A_{21})$$

if A_{22} is invertible. [4]

Schur's formula enables us to use the following alternate definition for positive definiteness.

► **Definition 4** (Sylvester's criterion). *A symmetric matrix is positive definite iff all its leading principal submatrices have positive determinants.*

⁴ We could also place ACL2's Cholesky decomposition implementation in a wrapper function with a recognizer for symmetric positive matrices as a guard.

25:10 Formalizing the Cholesky Factorization Theorem

■ **Algorithm 2** An algorithm for checking whether symmetric matrices are positive definiteness.

procedure PD(A)

Partition $A = \begin{pmatrix} \alpha_{11} & a_{12}^T \\ a_{21} & A_{22} \end{pmatrix}$ ▷ If $n, m > 1$, then $\alpha \in \mathbb{R}$, $a_{21} \in \mathbb{R}^{(n-1) \times 1}$,
 $a_{12}^T \in \mathbb{R}^{1 \times (m-1)}$, $A_{22} \in \mathbb{R}^{(n-1) \times (m-1)}$.

if $m = 0$ or $n = 0$ **then** ▷ Base case

return True

if $\alpha \leq 0$ **then** ▷ Check if determinant is nonpositive

return False

return PD($A_{22} - a_{21}\alpha_{11}^{-1}a_{12}^T$) ▷ Recursive case

This is also a quantified statement with the added issue that determinants are known to be computationally uncooperative. To avoid quantifiers, note the $(k-1)$ -th leading principal submatrix of A is also the $(k-1)$ -th leading principal submatrix of the k -th leading principal submatrix of A . The positivity of the former affects the positivity of the latter. This line of reasoning leads to a recursive (and, in particular, executable) recognizer.

Instead of explicitly computing determinants, we recursively check the positivity of the leading principal submatrix's determinant as shown in Algorithm 2. To understand this algorithm intuitively, consider the following partition

$$A = \begin{pmatrix} A_{11} & a_{12} & A_{13} \\ a_{21}^T & \alpha_{22} & a_{23}^T \\ A_{31} & a_{32} & A_{33} \end{pmatrix}.$$

Suppose A_{11} is the $(k-1)$ -th leading principal submatrix of A and suppose we know $\det(A_{11}) > 0$. Then the determinant of the k -th leading principal submatrix of A is

$$\det \begin{pmatrix} A_{11} & a_{12} \\ a_{21}^T & \alpha_{22} \end{pmatrix} = \det(A_{11}) \det(\alpha_{11} - a_{21}^T A_{11}^{-1} a_{12}) = \det(A_{11}) (\alpha_{11} - a_{21}^T a_{12} / \det(A_{11})) \quad (2)$$

is positive iff $\alpha_{11} - a_{21}^T a_{12} / \det(A_{11}) > 0$. In Algorithm 2, this check is performed immediately after the recursive call.

To see an (informal) mathematical proof for why Algorithm 2 works, we require one more result.

► **Proposition 5.** *Suppose A is symmetric and partition $A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}$. Then A is positive definite iff A_{11} is positive definite and $A_{22} - A_{21}A_{11}^{-1}A_{12}$ is positive definite. [4]*

We are now ready to (informally) prove the correctness of Algorithm 2.

► **Theorem 6.** *Algorithm 2 returns “True” on a symmetric matrix A iff every leading principle submatrix of A has a positive determinant.*

Proof. To see the forwards direction, proceed by induction on the size n of A . For $n = 1$, we have $A = (\alpha_{11}) > 0$ when Algorithm 2 recognizes $\alpha > 0$. Let $n = k$ and suppose Algorithm 2 recognizes positive definiteness on symmetric matrices of size $k-1$. Partition

$$k = \begin{pmatrix} \alpha_{11} & a_{21}^T \\ a_{21} & A_{22} \end{pmatrix}.$$

■ **Program 2** ACL2 function for checking positive definiteness.

```
(define positive-definite-p ((A matrixp))
:guard (equal (col-count A) (row-count A))
:measure (and (row-count A) (col-count A))
:returns (pd booleanp)
(b* (;; BASE CASES
      ((unless (matrixp A)) nil) ;; If A not a matrix, return empty
      ((if (m-emptyp A) t) ;; If A empty, return A

      ;; CHECK IF DETERMINANT SO FAR IS POSITIVE
      (alph (car (col-car A))) ;; alph := "top left" scalar in A
      ((unless (realp alph)) nil) ;; If alph not real, return nil
      ((unless (< 0 alph)) nil) ;; If alph not positive, return nil

      ;; BASE CASES
      ((if (m-emptyp (row-cdr A))) t) ;; If A is a row, return t
      ((if (m-emptyp (col-cdr A))) t) ;; If A is a column, return t

      ;; PARTITION
      (a12 (row-car (col-cdr A))) ;; [ alph | a12 ] := A
      (a21 (col-car (row-cdr A))) ;; [ ----- ]
      (A22 (col-cdr (row-cdr A))) ;; [ a21 | A22 ]

      ;; COMPUTE THE SCHUR COMPLEMENT
      (alph (acl2-sqrt alph))
      (a12 (sv* (/ alph) a12))
      (a21 (sv* (/ alph) a21))
      (A22 (m+ A22 (sm* -1 (out-* a12 a21)))))) ;; A22 := A22 - a12 * a21T / alph

      ;; RECURSE
      (positive-definite-p A22)) ;; Check if A22 is positive definite
/// ...)
```

Thanks to Proposition 5, A is positive definite iff α_{11} and $A_{22} - a_{21}\alpha_{11}^{-1}a_{21}^T$ are both positive definite. If Algorithm 2 returns “True” on A , then we must have $\alpha_{11} > 0$ and $\text{PD}(A_{22} - a_{21}\alpha_{11}^{-1}a_{21}^T)$ is true. Clearly, α_{11} is positive definite. Note A_{22} and $a_{21}\alpha_{11}^{-1}a_{21}^T$ are both symmetric and $(k-1) \times (k-1)$. By the induction hypothesis, $A_{22} - a_{21}\alpha_{11}^{-1}a_{21}^T$ is also positive definite.

To see the other direction, we prove the contrapositive. Suppose Algorithm 2 returns “False” after k recursive calls, i.e. Algorithm 2 returned “True” $k-1$ times. Then some α_{11} must have been zero or negative. But we’ve already seen from Equation (2) that α_{11} is a factor in the determinant of the k -th leading principal submatrix. Since Algorithm 2 returned “True” all the other $k-1$ times, any other factor in the expansion via Schur’s formula must be positive. Thus the determinant of the k -th leading principal submatrix must be zero or negative. ◀

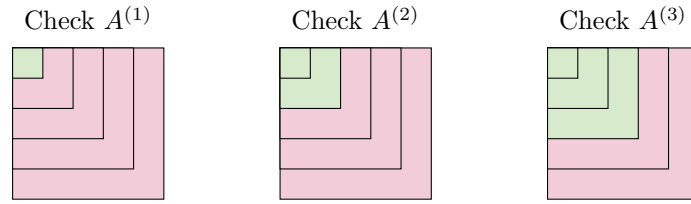
The ACL2 implementation of Algorithm 2 is shown in Program 2. Visually, the progress of Program 2 is demonstrated in Figure 2. The structure of the program is similar to that of Program 1 in that it also progresses diagonally from the “top left” to the “bottom right”. This similarity enables certain rewrite rules to fire and automatically verify the correctness of Program 1.

4.3 Verifying the Decomposition Algorithm

Given an executable and recursive condition for positive definiteness, we are now ready to formally prove Theorem 1 in ACL2. However, let’s first look at the informal mathematical proof.

25:12 Formalizing the Cholesky Factorization Theorem

■ **Figure 2** Progress of Program 2; $A^{(k)}$ denotes the k -th leading principal submatrix.



■ **Program 3** ACL2 theorem for the correctness of a Cholesky decomposition program (Program 1).

```
(defthm chol-correctness
  (b* ((L      (get-L (chol A)))
      (Lt     (mtrans L)))
      (implies (and (equal (mtrans A) A)
                    (positive-definite-p A)
                    (equal (col-count A) (row-count A)))
                (equal (m* L Lt) A))))
```

► **Theorem 1.** *Let A be a symmetric positive definite matrix. Let L be the lower triangular part of $\text{CHOL}(A)$. Then $A = LL^T$.*

Proof. Verifying the correctness of Program 1 amounts to induction on the size of A . The case where $n = 1$ is straightforward. Suppose CHOL computes the Cholesky decomposition for symmetric positive definite matrices of dimension $(k - 1) \times (k - 1)$. To see that CHOL computes the appropriate decomposition when A is $k \times k$, we just need to unravel one “layer” of LL^T and apply our induction hypothesis. Again partition

$$A = \begin{pmatrix} \alpha_{11} & a_{12}^T \\ a_{21} & A_{22} \end{pmatrix}$$

so that

$$\text{CHOL}(A) = \begin{pmatrix} \sqrt{\alpha_{11}} & a_{12}^T \\ a_{21}/\sqrt{\alpha_{11}} & \text{CHOL}(A_{22} - a_{21}a_{12}^T/\alpha_{11}) \end{pmatrix}$$

according to Algorithm 1. Let L be the lower triangular part of $\text{CHOL}(A)$ and let L_{22} be the lower triangular part of $\text{CHOL}(A_{22} - a_{21}a_{12}^T/\alpha_{11})$. Since $A_{22} - a_{21}a_{12}^T/\alpha_{11}$ is a symmetric positive definite $(k - 1) \times (k - 1)$ matrix, the lower triangular part of CHOL on the same matrix is its own Cholesky decomposition, i.e. $L_{22}L_{22}^T = A_{22} - a_{21}a_{12}^T/\alpha_{11}$. Then

$$\begin{aligned} LL^T &= \begin{pmatrix} \sqrt{\alpha_{11}} & \\ a_{21}/\sqrt{\alpha_{11}} & L_{22} \end{pmatrix} \begin{pmatrix} \sqrt{\alpha_{11}} & \\ a_{21}/\sqrt{\alpha_{11}} & L_{22} \end{pmatrix}^T \\ &= \begin{pmatrix} \alpha_{11} & a_{21}^T \\ a_{21} & L_{22}L_{22}^T + a_{21}a_{21}^T/\alpha_{11} \end{pmatrix} \\ &= \begin{pmatrix} \alpha_{11} & a_{21}^T \\ a_{21} & A_{22} \end{pmatrix} \\ &= A \end{aligned} \tag{3}$$

as desired. ◀

Program 3 contains the ACL2 theorem for verifying the correctness of Program 1. In addition to `(positive-definite-p A)`, the hypothesis posits that $A = A^T$ and that A is square. The extra symmetry condition is necessary because `positive-definite-p` in

■ **Program 4** ACL2 Skolem function positing the existence of an LU decomposition.

```
(defun-sk chol-fact-exists (A)
  (exists (L) (and (lower-tri-p L) (equal (m* L (mtrans L)) A))))
```

■ **Program 5** Theorem event automatically introduced into ACL2 by `defun-sk` in Program 4.

```
(defthm chol-fact-exists-suff
  (implies (and (lower-tri-p L)
                (equal (m* L (mtrans L)) A))
           (chol-fact-exists A))
  ;; L is lower triangular
  ;; L * L^T = A
  ;; A has a Cholesky decomposition
```

Program 2 doesn't assume that `a12` is equal to `a21`. Similarly, Equation (3) in the proof of Theorem 1 requires $a_{21} = a_{12}$ in order to recover A . Ultimately, the similarity between the structure of `positive-definite-p` in Program 2 and `chol` in Program 1 is what enables us to discharge `chol-correctness` in Program 3 automatically.

4.4 From Decomposition Algorithm to Factorization Theorem

Theorem 1 is distinct from statement of Theorem 2, which posits the existence of a Cholesky decomposition in its conclusion.

► **Theorem 2** (Cholesky Factorization Theorem). *If A is a symmetric positive definite matrix, then $A = LL^T$ for some lower triangular matrix L .*

Reasoning in ACL2 typically takes place by making propositional statements about functions, such as Program 3 or, equivalently, Theorem 1. One advantage to this is that ACL2 is highly automated. One disadvantage to this is that expressing statements such as Theorem 2 can be a challenge.

While the ACL2 logic is quantifier-free, reasoning about quantified statements is still supported by way of *Skolem functions*. A Skolem function in ACL2, introduced by `defun-sk`, is a function whose body has an outermost quantifier. For example, an ACL2 Skolem function for the latter part of Theorem 2 is Program 4. The function `chol-fact-exists` states “there exists an L such that L is lower triangular and $LL^T = A$ ”. The function `lower-tri-p` is simply a recognizer for lower triangular matrices, the ACL2 code for which we omit for brevity. To state Theorem 2 then amounts to placing a call to `chol-fact-exists` in the conclusion of a typical ACL2 theorem. Strictly speaking, we are still reasoning by asserting a function within a propositional statement; the function simply describes a quantified statement. The specifics of `defun-sk` are beyond the scope of this paper. The upshot is that a theorem of the form seen in Program 5 is automatically introduced into the ACL2 logical universe. Essentially, the theorem `chol-fact-exists-suff` states that if L is lower triangular and L multiplied by its transpose equals A , then there exists a Cholesky decomposition for A . Ultimately, we want to eliminate the hypotheses involving L and have the conclusion hold conditioned purely on A . If a witness is provided for L , then we can prove Theorem 2. Given Program 3, the clear witness is the lower triangular part of `(chol A)`. We pass the witness by instantiating `chol-fact-exists-suff` and replacing L with `(get-L (chol A))` using a hint in the desired theorem. The desired theorem is Program 6.

25:14 Formalizing the Cholesky Factorization Theorem

■ **Program 6** The Cholesky Factorization Theorem in ACL2.

```
(defthm cholesky-factorization-theorem
  (implies (and (equal (mtrans A) A)
                (positive-definite-p A)
                (equal (col-count A) (row-count A)))
           (chol-fact-exists A))
  :hints (("Goal" :use ((:instance chol-fact-exists-suff (L (get-L (chol A))))))))
```

■ **Table 3** ACL2 statistics related to the Cholesky Factorization Theorem.

Lines of code	1140
Events	192
Prover steps	5 029 675
Verification time (s)	7.91
Memory allocated (GB)	1.37

5 Conclusion

In this paper, we formalize and verify a Cholesky decomposition algorithm. Our work is open sourced as an ACL2 “book”, which can be found in the ACL2 GitHub repository [12] and as part of future ACL2 releases [11]. We invite interested readers to try decomposing their own matrices using ACL2. A summary of ACL2 statistics for this work is shown in Table 3. Events are updates to the ACL2 logical world, such as new definitions or theorems. Prover steps are the number of steps to justify the events. Verification was performed on a laptop with an Apple M1 Pro CPU.

Few theorem prover formalizations of numerical linear algebra algorithms exist; this is likely because typical numerical algorithms heavily employ indexing and few theorem provers are equipped to reason in this manner. We embed the FLAME environment into ACL2 so that we may verify the veracity of such algorithms. FLAME separates itself from other approaches by presenting algorithms which are designed to be proven correct, in no small part due to how matrices are partitioned in the derivation. The FLAME approach facilitates a useful derivation that enables us to develop an elegant ACL2 proof for the Cholesky Factorization Theorem.

Our Cholesky decomposition algorithm takes square roots in each recursive update which is why we use ACL2(r). Otherwise, using the vanilla version of ACL2 (without support for real and complex irrationals) would be sufficient. The presentation in this paper used our logical definition of Cholesky, which involves taking square roots by way of operations involving a nonstandard objects. To make execution more amenable, we define an alternate Cholesky program which employs an iterative square root function `sqrt-iter`. This iterative square root has been verified to converge to the logical square root `acl2-sqrt` [7]. It is also possible to reason about square roots in vanilla ACL2 using only its algebraic properties, e.g. by augmenting the field of ACL2 numbers with $\sqrt{\quad}$. Instead of developing a new theory in ACL2, we decided to simply use ACL2(r). Moreover, a theory of infinitesimals, such as the one supported by the non-standard analysis in ACL2(r), would be useful for any future attempt at verifying the *backwards error analysis* of formalized numerical linear algebra algorithms.

One major future direction for our work is to develop an ACL2 framework for reasoning about backwards error analysis. Backwards error analysis is paramount to ensuring the stability of numerical algorithms, thus providing a form of safety to their critical applications.

Recent ACL2 developments have enabled support for ACL2 computations involving floating-point numbers [1], providing us with an appropriate framework to reason about floating-point implementations of numerical algorithms. Moreover, expressing stability involves taking the norms of vectors and matrices, which motivates our future ACL2 investigation into these topics.

In addition to formalizing stability, we want to develop an ACL2 theory of norms because they are used in other numerical algorithms which deserve verification, such as QR decomposition. The QR decomposition is another fundamental algorithm in scientific computing with numerous applications and is usually introduced by the Gram-Schmidt process. However, we anticipate the *Householder* QR decomposition algorithm, a more stable alternative to Gram-Schmidt, to share the structure of Algorithm 1. The similar structures suggest that Householder QR may find an ACL2 verification in much the same way as our Cholesky decomposition algorithm in this paper. We will target a Householder QR decomposition algorithm for verification as future work.

Thus far we have discussed the use of ACL2 as a proof assistant for verifying theorems and formalizing theories. Indeed, the relatively low ratio of lines of code to theorem prover steps from Table 3 indicates that ACL2 has a high degree of automation in the context of proving pure mathematical results. But another future direction is to use our work (for not just the verification of, but also) *in* scientific computing’s most critical applications, which span fields such as machine learning, medical imaging, bioengineering, finance, structural engineering, aerospace, and much more. To make a verified numerical linear algebra library practical, it also needs to be efficient. A concrete optimization we can make in Algorithm 1 is to ignore the strictly upper triangular part of A . Note that because A is assumed to be symmetric and only symmetric updates are performed, namely $A_{22} := A_{22} - a_{21}a_{21}^T$, there is no need to update both the lower triangular and the upper triangular parts. Moreover, once the algorithm terminates, we are only interested in the lower triangular part of the result. Instead, we can update merely the lower triangular part of $A_{22} := A_{22} - a_{21}a_{21}^T$, performing a so-called *symmetric rank-one update*, to halve the computational cost of the algorithm.

Improvements can also be made to improve the baseline execution speed of our matrix programs. By default, numeric computations are offloaded to Lisp. Significant efforts, now part of the standard ACL2 toolbox, have been made to improve the executional efficiency of certain kinds of models. One such improvement is the development of *single-threaded objects* (stobjs) [5]. Logically, a stobj is a standard ACL2 association list; on the backend, updates to a stobj are made via destructive memory assignments, making them highly practical in situations where execution speed is vital. Stobjs were originally developed to improve the simulation speeds of ACL2 microprocessor models. Given our simple “list of lists” model of matrices, stobjs and its supporting framework, such as “access” and “update” functions, can also be used to represent ACL2 matrices and matrix operational semantics, respectively. The efficiency of destructive assignments in memory during field updates is ample motivation to investigate the potential use of stobjs for numerical linear algebra algorithms, especially those which involve computing a result in place, such as the Cholesky decomposition algorithm we formalize in this paper.

There are very few theorem prover formalizations of numerical linear algebra algorithms – even fewer have support for execution. Developing non-executable libraries of numerical algorithms (formal or otherwise) seems antithetical to their computational nature. In addition to presenting the first formalization of a major theorem in linear algebra, our work in this paper sets a precedent for verified scientific computing in the future.

References

- 1 ACL2. *Df: Support for floating-point operations*. Accessed 2024-06-06. URL: https://cs.utexas.edu/users/moore/acl2/manuals/current/manual/index-seo.php/ACL2___DF.
- 2 ACL2. *Loop\$*. Accessed 2024-06-06. URL: https://cs.utexas.edu/users/moore/acl2/manuals/current/manual/index.html?topic=ACL2___LOOP_42.
- 3 ACL2. *User manual for the ACL2 Theorem Prover and the ACL2 Community Books*. Accessed 2023-07-12. URL: <https://cs.utexas.edu/users/moore/acl2/current/manual/index.html>.
- 4 Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004. doi:10.1017/CB09780511804441.
- 5 Robert S. Boyer and J. Strother Moore. Single-threaded objects in ACL2. In *Proceedings of the 4th International Symposium on Practical Aspects of Declarative Languages, PADL '02*, pages 9–27, Berlin, Heidelberg, 2002. Springer-Verlag. doi:10.5555/645772.667953.
- 6 Ruben Gamboa, John Cowles, and Jeff Van Baalen. Using ACL2 arrays to formalize matrix algebra, 2003. URL: <https://cs.uwoy.edu/~ruben/static/pdf/matalg.pdf>.
- 7 Ruben A. Gamboa and Matt Kaufmann. Nonstandard analysis in ACL2. *J. Autom. Reason.*, 27(4):323–351, November 2001. doi:10.1023/A:1011908113514.
- 8 John A. Gunnels, Fred G. Gustavson, Greg M. Henry, and Robert A. van de Geijn. FLAME: Formal linear algebra methods environment. *ACM Trans. Math. Softw.*, 27(4):422–455, December 2001. doi:10.1145/504210.504213.
- 9 John Harrison. The HOL Light theory of Euclidean space. *Journal of Automated Reasoning*, 50:173–190, 2012. doi:10.1007/s10817-012-9250-9.
- 10 Joe Hendrix. Matrices in ACL2, 2003. URL: <https://cs.utexas.edu/users/moore/acl2/workshop-2003/contrib/hendrix/hendrix.pdf>.
- 11 Matt Kaufmann and J Strother Moore. ACL2 home page. <https://cs.utexas.edu/~moore/acl2/>, 1997. Accessed 2024-06-25.
- 12 Matt Kaufmann and J Strother Moore. ACL2 system and community books. <https://github.com/acl2/acl2>, 2014. Accessed 2024-06-25.
- 13 Carl Kwan. Classical LU decomposition in ACL2. *Electronic Proceedings in Theoretical Computer Science*, 393:1–5, November 2023. doi:10.4204/eptcs.393.1.
- 14 Carl Kwan and Mark R. Greenstreet. Real vector spaces and the Cauchy-Schwarz inequality in ACL2(r). *Electronic Proceedings in Theoretical Computer Science*, 280:111–127, October 2018. doi:10.4204/eptcs.280.9.
- 15 Lean mathlib3 documentation: LDL decomposition. https://leanprover-community.github.io/mathlib_docs/linear_algebra/matrix/ldl.html. Accessed 2023-07-13.
- 16 Lean mathlib3 documentation: Matrices. https://leanprover-community.github.io/mathlib_docs/data/matrix/basic.html. Accessed 2023-07-13.
- 17 ZhengPu Shi and Gang Chen. Integration of multiple formal matrix models in Coq. In Wei Dong and Jean-Pierre Talpin, editors, *Dependable Software Engineering. Theories, Tools, and Applications*, pages 169–186, Cham, 2022. Springer Nature Switzerland. doi:10.1007/978-3-031-21213-0_11.
- 18 Zhiping Shi, Yan Zhang, Zhenke Liu, Xinan Kang, Yong Guan, Jie Zhang, and Xiaoyu Song. Formalization of matrix theory in HOL4. *Advances in Mechanical Engineering*, 6:195–276, 2014. doi:10.1155/2014/195276.
- 19 Christian Sternagel and René Thiemann. Executable matrix operations on matrices of arbitrary dimensions. *Archive of Formal Proofs*, June 2010. , Formal proof development. URL: <https://isa-afp.org/entries/Matrix.html>.
- 20 René Thiemann and Akihisa Yamada. Matrices, Jordan normal forms, and spectral radius theory. *Archive of Formal Proofs*, August 2015. , Formal proof development. URL: https://isa-afp.org/entries/Jordan_Normal_Form.html.

The Rewster: Type Preserving Rewrite Rules for the Coq Proof Assistant

Yann Leray  

LS2N, Nantes Université, France
Inria, Nantes, France

Gaëtan Gilbert

Inria, Nantes, France

Nicolas Tabareau 

Inria, Nantes, France

Théo Winterhalter 

Inria Saclay, France

Abstract

In dependently typed proof assistants, users can declare axioms to extend the ambient logic locally with new principles and propositional equalities governing them. Additionally, rewrite rules have recently been proposed to allow users to extend the logic with new definitional equalities, enabling them to handle new principles with a computational behaviour. While axioms can only break consistency, the addition of arbitrary rewrite rules can break other important metatheoretical properties such as type preservation. In this paper, we present an implementation of rewrite rules on top of the Coq proof assistant, together with a modular criterion to ensure that the added rewrite rules preserve typing. This criterion, based on bidirectional type checking, is formally expressed in PCUIC – the type theory of Coq recently developed in the MetaCoq project.

2012 ACM Subject Classification Theory of computation → Type theory

Keywords and phrases type theory, dependent types, rewrite rules, type preservation, Coq

Digital Object Identifier 10.4230/LIPIcs.ITP.2024.26

1 Introduction

Dependently typed languages are the basis of major proof assistants like Agda [11], Coq [15] and Lean [5]. In these systems, since general equality is undecidable, it is split into a decidable fragment called *definitional* and the complete but undecidable *propositional* equality. While propositional equality can be extended through axioms, definitional equality is defined upfront and limited in a way that can hinder proof developments and usability. Rewrite rules, as recently proposed by Cockx et al. [3], can mitigate these limitations, by allowing users to extend definitional equality in a powerful way through custom reductions. A standard example one can write using our implementation is parallel plus, an addition which can reduce on constructors from both sides, something which cannot be defined using a fixpoint.

```
Symbol pplus : ℕ → ℕ → ℕ.  
Infix "++" := pplus.  
Rewrite Rules pplus_rew :=  
| 0 ++ ?n ↔ ?n      | S ?m ++ ?n ↔ S (?m ++ ?n)  
| ?m ++ 0 ↔ ?m      | ?m ++ S ?n ↔ S (?m ++ ?n).
```

The symbol `pplus` behaves as standard addition, but with additional definitional equalities, that only hold propositionally for standard addition. For instance, `n ++ 0` and `0 ++ n` are both definitionally equal to `n`. This example shows the definition of a function that could already be defined, albeit with fewer definitional properties, but it is also possible to extend



© Yann Leray, Gaëtan Gilbert, Nicolas Tabareau, and Théo Winterhalter;
licensed under Creative Commons License CC-BY 4.0

15th International Conference on Interactive Theorem Proving (ITP 2024).

Editors: Yves Bertot, Temur Kutsia, and Michael Norrish; Article No. 26; pp. 26:1–26:18

Leibniz International Proceedings in Informatics



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

26:2 The Rewster: Type Preserving Rewrite Rules for the Coq Proof Assistant

the logic of Coq with new powerful constructions like inductive-recursive types using rewrite rules. As an illustration, consider the definition of (a simple version of) a universe of types:

```
Axiom U : Type.
Symbol El : U → Type.
Symbols (N : U) (Pi : ∀ a : U, (El u → U) → U).
Symbol U_rect : ∀(P : U → Type), P N → (∀ a b, (∀ A : El a, P (b A)) → P (Pi a b)) → ∀ u, P u.
Rewrite Rules U_rect_rew :=
| U_rect ?P ?pN ?pPi N ~> ?pN
| U_rect ?P ?pN ?pPi (Pi ?a ?b) ~> ?pPi ?a ?b (fun (A : El ?a) => U_rect ?P ?pN ?pPi (?b A)).
Rewrite Rules El_red :=
| El N ~> N
| El (Pi ?a ?b) ~> ∀ (A : El ?a), El (?b A).
```

Without rewrite rules, this construction can be fully postulated with axioms using propositional equality, but in practice neither the elimination principle `U_rect` nor function `El` will compute, which makes the construction much less convenient to use. From this point of view, strictly positive indexed inductive types currently available in Coq can be seen as one particular class of inductive constructions that have been anticipated, justified and provided natively in the system. Rewrite rules allow users to go outside this fragment, for instance by defining a non-strictly positive inductive type.¹

However, contrarily to axioms which can only endanger consistency, the power of rewrite rules needs to be put in check or important metatheoretical properties can be broken. The work of Cockx et al. puts in light that the first property that rewrite rules must verify is confluence, otherwise subject reduction can be broken and type checking quickly becomes undecidable. They develop a criterion that can be checked to determine if a rewriting system satisfies confluence, namely the triangle property [3].

To preserve subject reduction of the complete theory, rewrite rules must also verify a second property, type preservation, that is if $t : T$ and t rewrites to t' , then $t' : T$. In [3], type preservation of rewrite rules is simply postulated, and one of the main contributions of this paper is to present a criterion to decide type preservation. A simple and intuitive check for type preservation is to require that the equality induced by the rewrite rule should be well typed for every possible instance ($\forall \vec{x}, p = r$). There are however problems with this criterion as terms don't have a unique type in Coq because of the subtyping introduced by universe cumulativity (`Type@{u}` is a subtype of `Type@{v}` when $u \leq_u v$). Let us see a few examples of how rewrite rules may break type preservation in ways that are difficult to notice.

► **Example 1.** If we consider an identity function on types, which simply returns its argument, cumulativity should mandate that the domain universe be smaller than the codomain universe. However, in the following example with a rewrite rule, this restriction is not enforced.

```
Symbol id@{u v} : Type@{u} → Type@{v}.
Rewrite Rule id_rew := id ?T ~> ?T.
Universe a.
Check id Type@{a} : id Type@{a}.
```

The reason is that the naive check only mandates that both universes share a common bigger universe – something which is always verified. Thus, the check doesn't forbid this rule which nonetheless breaks consistency by allowing the existence of a term $U : U$ [8, 9].

¹ The fact that general non-strictly positive inductive types are unsafe is due to Coquand and Paulin [4], but the argument uses impredicativity.

This paper demonstrates that establishing a sound criterion for type checking, in the absence of unique types, requires an asymmetry between both sides of the rewrite rules. Indeed, having a common type is not enough, but the correct criterion ensures that the right-hand side of a rewrite rule *checks* against the principal type *inferred* from the pattern on the left-hand side.

Another issue with the naive criterion is that the type of the variables \vec{x} that appear in the pattern might also not be unique, and just testing for one possibility that works is not enough.

► **Example 2.** Let us consider a symbol which extracts the domains of product types, which can be defined as follows:

```
Symbol dom : Prop → Prop.
Rewrite Rule dom_rew := dom (∀ (x : ?A), ?B) ↔ ?A.
Eval compute in dom (∀ (x : ℕ), True). (* ℕ : Prop *)
```

The problem here is that no information tells us what sort $?A$ has in the pattern in general, since the domain of a product type may have any sort quality (SProp, Prop or Type). Our naive criterion, however, fails to detect the issue because the equality is well typed when both $?A$ and $?B$ have type Prop, since then both the pattern and the replacement term will have that type, and this problematic rule can then be accepted.

The correct criterion will make use of typing in patterns to try to find equalities between pattern variables, so they can be used when typing the replacement term. We can however also face an issue if we use the existing unification algorithm as is on patterns, since it uses shortcut heuristics which don't always assume the worst.

► **Example 3.** Consider the following:

```
Inductive Box (b : ℤ) : Type := box. (* box : ∀ (b : ℤ), Box b *)
Symbol I : ℤ → ℤ.
Symbol C : ∀ (b : ℤ), Box b → Box (I b).
Symbol D : ∀ (b : ℤ), Box (I b) → Box b.
Rewrite Rule D_C := D _ (C _ ?b) ↔ ?b.

Rewrite Rule I_rew := I _ ↔ false.

Definition a : Box false := (D false (C _ (box true))).
Eval compute in a. (* box true : Box false *)
```

The rewrite rule $D_ (C_ ?b) \leftrightarrow ?b$ looks enticing, if the box is not to be considered as a truncation. However, we don't know the behaviour of I at that point, and as we add rule $I_ \leftrightarrow \text{false}$, our first rule loses its type preservation property, so it should not in fact have been accepted in the first place.

The last meta-theoretical property that is not endangered by adding axioms but may break with additional rewrite rules is termination. However, non-termination cannot introduce proofs of \perp ; these can only come from the declaration of symbols and rules which are inconsistent propositionally from the start [3, Section 6.4]. It does not break subject reduction either, and we can derive a type checker that is valid irrespective of termination, in the sense that if it answers, the answer is correct. Non-termination can at worst result in type checking divergence. Therefore, we can show subject reduction with our criterion without relying on termination.

A, B, P, T, c, t, \dots	$::=$	x	variable
		$?x\{\vec{t}\}$	metavariable
		s	sort
		A	axiom
		C	constructor
		I	inductive
		$t\ t'$	application
		$\lambda(x : A : s), t$	abstraction
		$\forall(x : A : s), (B : s')$	product
		case c return $P : s$ with \vec{t}	case destruction
		fix $f : T := t$	fixpoint
s, s'	$::=$	\square_u^q	sort
q	$::=$	SProp Prop Type	qualities

■ **Figure 1** The (annotated) syntax of PCUIC, extended with metavariables.

In this paper, we present how rewrite rules can be added to PCUIC, the theory of the kernel of Coq as defined in the MetaCoq project [13, 14] Coq, and extend the work of Cockx et al. [3] by providing a criterion for type preservation in presence of cumulativity. Rewrite rules, as described in this paper and using the syntax demonstrated in the examples, have been implemented and recently integrated into the development branch of the Coq proof assistant.² This means they are expected to be available in the upcoming release 8.20 of Coq.

Outline of the paper. We begin by presenting an extension of PCUIC with metavariables in Section 2. We then define rewrite rules in Section 3 before explaining the type preservation criterion in Sections 4 and 5. Finally, we explore the implementation in Section 6 and conclude with related and future work in Sections 7 and 8.

2 PCUIC with Metavariables

2.1 Syntax

PCUIC, whose syntax is given in Figure 1, is a formal description of the kernel of Coq, as defined in the MetaCoq project [14]. It is a dependent type theory with inductive types, where inductive elimination is split between fixpoints and case analysis. Its sorts are of three different so-called *qualities*: the usual Type hierarchy, with universe levels u as index, and two additional sorts Prop and SProp. These two last sorts are impredicative, and SProp also has definitional uniqueness of proofs, i.e., any two elements of a type in SProp are convertible. Universes are ordered with \leq_u , but when the quality of a sort is Prop or SProp, its universe becomes irrelevant. This defines an ordering \leq_s on sorts with $\square_u^q \leq_s \square_{u'}^{q'} \iff q = q' \wedge (q \in \{\text{SProp}, \text{Prop}\} \vee u \leq_u u')$. We also order $\square_u^{\text{Prop}} <_s \square_{u'}^{\text{Type}}$. One should note that λ -abstractions, products and cases are heavily annotated; the type annotation on the domains is standard for PCUIC, but the additional sort annotations are

² <https://github.com/coq/coq/pull/18038>

only required for the proof of our type preservation criterion and should be considered as virtual. They can be unambiguously obtained from the typing judgment on the term and will also be omitted when they aren't useful.

To account for higher-order rewrite rules, the syntax needs to be extended with higher-order variables – henceforth called metavariables. A metavariable $?x\{\vec{t}\}$ is meant to live in an extended context, where said context extension is to be instantiated by the context instantiation \vec{t} that the metavariable carries. While regular variable substitution will be denoted as $t[\tau]$, metasubstitution – substitution for metavariables – will be denoted as $t\{\sigma\}$. Regular substitutions do nothing to metavariables, they only propagate to the context instantiation: $(?x\{\vec{t}\})[\tau] = ?x\{\vec{t}[\tau]\}$. Dually, metasubstitutions do nothing to variables and operate solely on metavariables as $(?x\{\vec{t}\})\{\sigma\} = \sigma(?x)[\vec{t}\{\sigma\}]$, they also propagate transparently to subterms on all other grammar constructions. Note that metasubstitutions need to be defined on all metavariables of the substituted term, a condition that will be verified for metasubstitutions appearing in rewrite rules.

2.2 Typing

The typing judgment of PCUIC is described in Figure 2. Judgement $\Sigma; \Theta; \Gamma \vdash t : T$ states that term t has type T in local context Γ , metavariable context Θ and environment Σ ; its rules are standard. The type of sorts needs to take into account qualities: $\square_{-}^{\text{Prop}}$ and $\square_{-}^{\text{SProp}}$ both have type \square_0^{Type} and \square_u^{Type} has type $\square_{u+1}^{\text{Type}}$; this is condensed in the universe successor function $\uparrow_q u$ which equals $u + 1$ when $q = \text{Type}$ and 0 otherwise. Similarly, the sort of a product has the quality of its codomain, but its universe level depends on both sorts: $u *_{q'}^q u'$ is $\max(u, u')$ if $q = q' = \text{Type}$ and u' otherwise.

Environment Σ can contain axiom declarations and inductive declarations. An axiom declaration simply gives a name and a type associated with it. An inductive declaration contains a name I , a context of parameters Γ_p , a context of indices Γ_i and a list of constructors, each with its name C_k , context Γ_k and instantiation of the indices \vec{v}_k . Note that \vec{v}_k may depend on Γ_p, Γ_k . Rule (CASE) asks for a scrutinee c of type I with some instantiation of the parameters \vec{p} and some instantiation of the indices \vec{v} , a return type P which depends on a generic instance of the inductive with the given parameters and branches at the return type specialized with the indices of the associated constructor, in the extended context of the constructor. Typing of environment Σ should enforce the usual strict positivity condition on inductive type declarations in order to preserve consistency.

In order to define typing of metavariables, the typing judgement needs to take as input a metavariable context Θ . Its elements are of the form $(\Gamma_{?x} \vdash ?x : T)$, which reads as “ $?x$ is of type T in context extension $\Gamma_{?x}$ ”; which means in turn that the context instantiation associated to $?x$ must instantiate $\Gamma_{?x}$. Technically, the typing rule for metavariables (Rule META) differs from the rule for variables (Rule VAR) in that it additionally checks that $\Sigma; \Theta; \Gamma \vdash \vec{t} : \Gamma_{?x}$, where typing is extended to substitution in a pointwise way.

Note that we remained abstract in our notion of guard condition for fixpoints. Indeed, fixpoints need to recurse on structurally smaller arguments in order to avoid inconsistencies. Moreover to ensure termination, the computation rule for fix is typically guarded so that fixpoints only unfold when they can consume a constructor. We forego this condition as we do not worry about termination in this paper.

$$\begin{array}{c}
 \frac{\Sigma; \Theta \vdash \Gamma \quad (x : T) \in \Gamma}{\Sigma; \Theta; \Gamma \vdash x : T} \text{VAR} \qquad \frac{\Sigma; \Theta \vdash \Gamma \quad (C : T) \in \Sigma}{\Sigma; \Theta; \Gamma \vdash C : T} \text{ENVVAR} \\
 \\
 \frac{\Sigma; \Theta \vdash \Gamma \quad (\Gamma_{?x} \vdash ?x : A) \in \Theta \quad \Sigma; \Theta; \Gamma \vdash \vec{t} : \Gamma_{?x}}{\Sigma; \Theta; \Gamma \vdash ?x\{\vec{t}\} : A} \text{META} \qquad \frac{\Sigma; \Theta \vdash \Gamma}{\Sigma; \Theta; \Gamma \vdash \square_u^q : \square_{\uparrow q u}^{\text{Type}}} \text{SORT} \\
 \\
 \frac{\Sigma; \Theta; \Gamma \vdash f : \forall(x : A), B \quad \Sigma; \Theta; \Gamma \vdash t : A}{\Sigma; \Theta; \Gamma \vdash f t : B[x := t]} \text{APP} \\
 \\
 \frac{\Sigma; \Theta; \Gamma \vdash A : \square_u^q \quad \Sigma; \Theta; \Gamma, x : A \vdash t : B}{\Sigma; \Theta; \Gamma \vdash \lambda(x : A : \square_u^q), t : \forall(x : A), B} \text{LAMBDA} \\
 \\
 \frac{\Sigma; \Theta; \Gamma \vdash A : \square_u^q \quad \Sigma; \Theta; \Gamma, x : A \vdash B : \square_{u'}^{q'}}{\Sigma; \Theta; \Gamma \vdash \forall(x : A : \square_u^q), (B : \square_{u'}^{q'}) : \square_{u^* q, u'}^{q'}} \text{FORALL} \\
 \\
 \frac{(\Gamma_p, \Gamma_i, [I : \forall \Gamma_p, \forall \Gamma_i, \square_{u_I}^{q_I}], \vec{\Gamma}_k, [C_k : \forall(\vec{p} : \Gamma_p), \forall \Gamma_k, I \vec{p} \vec{i}_k]_k) \in \Sigma \quad \Sigma; \Theta; \Gamma \vdash c : I \vec{p} \vec{i} \quad \Sigma; \Theta; \Gamma, (\vec{v} : \Gamma_i), (x : I \vec{p} \vec{v}) \vdash P : \square_u^q \quad \Sigma; \Theta; \Gamma, (\vec{a} : \Gamma_k) \vdash b_k : P[\Gamma_i := \vec{i}_k, x := C_k \vec{p} \vec{a}]_k}{\Sigma; \Theta; \Gamma \vdash \text{case } c \text{ return } P : \square_u^q \text{ with } \vec{b}_k : P[\Gamma_i := \vec{i}, x := c]} \text{CASE} \\
 \\
 \frac{\Sigma; \Theta; \Gamma \vdash T : \square_u^q \quad \Sigma; \Theta; \Gamma, f : T \vdash t : T \quad t \text{ guarded}}{\Sigma; \Theta; \Gamma \vdash \text{fix } f : T := t : T} \text{FIX} \\
 \\
 \frac{\Sigma; \Theta; \Gamma \vdash t : A \quad \Sigma; \Theta; \Gamma \vdash A \leq B \quad \Sigma; \Theta; \Gamma \vdash B : \square_u^q}{\Sigma; \Theta; \Gamma \vdash t : B} \text{CUMUL} \\
 \\
 \frac{}{\Sigma; \Theta; \Gamma \vdash (\lambda(x : A), t) a \rightarrow t[x := a]} \beta \qquad \frac{}{\Sigma; \Theta; \Gamma \vdash \text{fix } f := t \rightarrow t[f := \text{fix } f := t]} \text{FIXRED} \\
 \\
 \frac{(\Gamma_p, \Gamma_i, [I : \forall \Gamma_p, \forall \Gamma_i, \square_{u_I}^{q_I}], \vec{\Gamma}_k, [C_k : \forall(\vec{p} : \Gamma_p), \forall \Gamma_i, I \vec{p} \vec{i}_k]_k) \in \Sigma}{\Sigma; \Theta; \Gamma \vdash \text{case } C_j \vec{p} \vec{a} \text{ return } P \text{ with } \vec{b}_k \rightarrow b_j[\Gamma_j := \vec{a}]} \iota \\
 \\
 \frac{\Sigma; \Theta; \Gamma \vdash \square_u^q \leq_s \square_{u'}^{q'} \quad \Sigma; \Theta; \Gamma \vdash T : \text{SProp} \quad \Sigma; \Theta; \Gamma \vdash t : T \quad \Sigma; \Theta; \Gamma \vdash u : T}{\Sigma; \Theta; \Gamma \vdash \square_u^q \leq_\alpha \square_{u'}^{q'} \quad \Sigma; \Theta; \Gamma \vdash t \leq_\alpha u} \\
 \\
 \frac{?x \in \Theta \quad \Sigma; \Theta; \Gamma \vdash \vec{t} \equiv_\alpha \vec{t}'}{\Sigma; \Theta; \Gamma \vdash ?x\{\vec{t}\} \leq_\alpha ?x\{\vec{t}'\}} \qquad \frac{\Sigma; \Theta; \Gamma \vdash f \leq_\alpha f' \quad \Sigma; \Theta; \Gamma \vdash a \equiv_\alpha a'}{\Sigma; \Theta; \Gamma \vdash f a \leq_\alpha f' a'} \\
 \\
 \frac{\Sigma; \Theta; \Gamma \vdash A \equiv_\alpha A \quad \Sigma; \Theta; \Gamma \vdash t \leq_\alpha t'}{\Sigma; \Theta; \Gamma \vdash \lambda(x : A : \square_u^q), t \leq_\alpha \lambda(x : A' : \square_{u'}^{q'}), t'} \\
 \\
 \frac{\Sigma; \Theta; \Gamma \vdash A \equiv_\alpha A' \quad \Sigma; \Theta; \Gamma \vdash B \leq_\alpha B'}{\Sigma; \Theta; \Gamma \vdash \forall(x : A : \square_u^q), (B : \square_{u_1}^{q_1}) \leq_\alpha \forall(x : A' : \square_{u'}^{q'}), (B' : \square_{u'_1}^{q'_1})}
 \end{array}$$

 ■ **Figure 2** Typing, reduction and cumulativity judgment in PCUIC with metavariables.

p	$::=$	x	variable
		$\square_{?x}^q$	sort
		C	constructor
		I	inductive
		S	symbol
		$p a$	application
		$\lambda(x : a : s), p$	abstraction
		$\forall(x : a : s), (b : s')$	product
		case p return $a : s$ with \vec{b}	case destruction
a, b	$::=$	$p \mid ?x$	pattern / pattern variable
q	$::=$	$\text{SProp} \mid \text{Prop} \mid \text{Type} \mid ?x$	explicit quality / quality variable
s, s'	$::=$	$\square_{?x}^{?y}$	pattern sort annotation

■ **Figure 3** Syntax of patterns p and argument patterns a, b .

2.3 Cumulativity

Our typing judgment contains a cumulativity rule (CUMUL), instead of the more traditional conversion rule. Cumulativity is defined similarly to conversion, except that we allow subtyping on sorts: $\square_u^q \leq_s \square_{u'}^q$. Formally, cumulativity is defined as the transitive closure \leq of $\rightarrow \cup \leftarrow \cup \leq_\alpha$ where \rightarrow is reduction, \leftarrow antireduction (the relation symmetric to \rightarrow) and \leq_α is α -cumulativity. We also define conversion \equiv by replacing α -cumulativity with α -conversion. The reduction is generated by β and ι -reductions, unfolding of fixpoints (Rule FIXRED) and is allowed to happen in any subterm. α -cumulativity is roughly α -equality with the cumulativity rule on sorts explained above, closed by congruence. However, most subterms need to be convertible and not only cumulative in the congruence. Formally, cumulativity \leq_α is defined with the rules described in Figure 2 and the other congruences where all subterms need to be related with \equiv_α , where $t \equiv_\alpha t'$ is defined as $t \leq_\alpha t' \wedge t' \leq_\alpha t$.

PCUIC also supports SProp's proof irrelevance in α -cumulativity. The rule given is more idealistic than the real presentation with relevance marks [7] but the differences won't matter here.

3 Rewrite Rules

A rewrite rule is composed of a left-hand side, called *pattern*, and a right-hand side, called *replacement term*, and written $p \rightsquigarrow r$. When it is applied, a term t is *matched* against the pattern p to extract subterms at the *holes* of the pattern to form a metasubstitution σ , which is then substituted in the replacement term r , resulting in reduction $t \rightarrow r\{\sigma\}$. As explained in the introduction, restrictions are needed so that the metatheory of PCUIC does not break.

3.1 Pattern

The syntax of patterns is described in Figure 3. As a separation from axioms, that never reduce, we introduce in PCUIC a new class of constants named symbols, which is a specific subclass of axioms on which rewrite rules operate.

Patterns corresponds to a subset of terms that need to be *rigid*. Their holes are denoted as metavariables ($?x$, also called pattern variables) in the pattern. Note that, unlike metavariables, pattern variables don't carry context instantiations. Thus to see a pattern

$$\begin{array}{c}
 \frac{}{t \mid ?x \{[?x := t]\}} \quad \frac{}{x \mid x \{\emptyset\}} \quad \frac{C \in \Sigma \text{ (not a def. or axiom)}}{C \mid C \{\emptyset\}} \\
 \\
 \frac{q \text{ explicit quality}}{\square_u^q \mid \square_{?x_u}^q \{[?x_u := u]\}} \quad \frac{}{\square_u^q \mid \square_{?x_q}^{?x_q} \{[?x_q := q; ?x_u := u]\}} \quad \frac{f \mid p \{\sigma_1\} \quad t \mid a \{\sigma_2\}}{f t \mid p a \{\sigma_1 + \sigma_2\}} \\
 \\
 \frac{A \mid a \{\sigma_1\} \quad t \mid p \{\sigma_2\}}{\lambda(x : A : \square_{u'}^{q'}), t \mid \lambda(x : a : \square_u^q), p \{[q := q'; u := u'] + \sigma_1 + \sigma_2\}} \\
 \\
 \frac{A \mid a \{\sigma_1\} \quad B \mid b \{\sigma_2\}}{\forall(x : A : \square_{u_1}^{q_1}), (t : \square_{u_2}^{q_2}) \mid \forall(x : a : \square_{u_1}^{q_1}), (b : \square_{u_2}^{q_2}) \{[\vec{q}_i := \vec{q}'_i; \vec{u}_i := \vec{u}'_i] + \sigma_1 + \sigma_2\}} \\
 \\
 \frac{c \mid p \{\sigma_c\} \quad P \mid a \{\sigma_P\} \quad b_i \mid b_i \{\sigma_i\}}{\text{case } c \text{ return } P : \square_{u'}^{q'} \text{ with } \vec{b}_i \mid \text{case } p \text{ return } a : \square_u^q \text{ with } \vec{b}_i \{[q := q'; u := u'] + \sigma_c + \sigma_P + \sum \sigma_i\}}
 \end{array}$$

■ **Figure 4** Description of pattern matching.

variable as a term, we have to add the identity instantiation $\{[\Delta := \Delta]\}$ for Δ the context in which the pattern variable lives. This gives us an injection $p \mapsto p$ from patterns to terms. From now on, we will simply use the change of color to represent this injection.

The rigidity requirements manifest as limitations on where holes may appear, so patterns are split into bona fide patterns, where holes will *not* be allowed, and *argument patterns* where they will be allowed. Since term reduction happens in a stack machine, the reduction of rewrite rules has to work on the machine representation of terms which have already been reduced to weak head normal form. For instance, having a hole at the head of eliminations (e.g., pattern $?f ?a$ against term $(\lambda x f, f x) 0 S$) would be unstable (does $?a$ match with 0 or S ?) and would not commute with other reductions ($?f$ can match with S after reductions), so it must be forbidden. This means that argument patterns may be anywhere but at the main subterm of an elimination construction, that is an application or a case destruction.

We also have to impose additional restrictions to prove confluence; they are listed and justified in Section 4.1. Finally, since we don't want to try all rewrite rules at all steps in reduction, we restrict patterns such that the head of their main branch of the pattern needs to be a symbol, which will be the starting point to pattern matching during reduction.

In the end, each rewrite rule induces the following reduction rule:

$$\frac{(p \rightsquigarrow r) \in \Sigma \quad t \mid p \{\sigma\}}{\Sigma; \Theta; \Gamma \vdash t \rightarrow r \{\sigma\}} \text{ REW}$$

where $t \mid p \{\sigma\}$ denotes the matching of p against a term t as described in the following section.

3.2 Pattern-Matching

Pattern matching is the operation that tries to match a term t against a pattern p , resulting in a metasubstitution σ when it succeeds. Each pattern constructor can match against the associated term constructor, trying to match recursively, and pattern variable match against

any term to fill the metasubstitution. It is denoted as $t \mid p \{ \sigma \}$ and is formally defined in Figure 4. Actually, pattern matching can also match universes against universe variables and sort qualities against quality variables, so the resulting σ contains in fact a metasubstitution, a universe substitution and a sort quality substitution. These last two substitutions are defined transparently on terms, only having effect on the quality and universe variables they carry. Pattern matching verifies one crucial property: $\forall p \ t \ \sigma, \ t \mid p \{ \sigma \} \implies t \equiv_{\alpha} p \{ \sigma \}$. This α -conversion would be an equality if not for the fact that some technical information is lost by pattern matching, such as names of binders.

Note that patterns are as heavily annotated as terms, but in this case it is required in the normal course of operations. Indeed, during the typing pass, we need to give a type to argument patterns so they can be provided to all pattern variables. This means that all type fields need an annotation to name the quality and universe of said type. These annotations can however not be used during pattern matching, as they contain no computational content; the quality and universe that are matched from the virtual annotation of terms can be considered virtual as well. These type annotations are used to avoid the issue presented in Example 2 where the type of a pattern variable may vary depending on the term being matched.

4 A Criterion to Guarantee that Rewrite Rules are Type Preserving

This section presents the criterion to ensure that rewrite rules preserve typing. To be able to prove any property surrounding subject reduction, we first have to ensure crucial properties like injectivity of product types, which are the consequences of the completeness of algorithmic cumulativity (defined as \rightarrow^* ; \leq_{α} ; \leftarrow^*). Informally, this means that for a given proof of cumulativity, we have to build a standardization of it that starts with repeated \rightarrow , then repeated \leq_{α} , then repeated \leftarrow (that is, we reduce both sides before applying \leq_{α}). To achieve this, we prove in Section 4.1 that we can move the different relations around, which needs confluence and postponement of α -cumulativity after reduction. This idea follows exactly the proof performed in MetaCoq [14].

Then, we define in Section 4.2 (bidirectional) type inference of patterns which is at the heart of the definition of our criterion for subject reduction. Section 4.3 is devoted to the proof that the criterion indeed ensures subject reduction.

4.1 Confluence and Postponement of α -cumulativity

We first turn to the proof of confluence and postponement of α -cumulativity after reduction. To ensure confluence, we want to use the triangle criterion [3]. The criterion only applies with left-linear rewrite rules, which means that patterns variable will be required to appear at most once in the pattern. The triangle criterion on rewrite rules says that when $t \mid p \{ \sigma \}$, with $(p \rightsquigarrow r) \in \Sigma$ and $t' \mid p' \{ \sigma' \}$ with $(p' \rightsquigarrow r') \in \Sigma$ for t' a subterm of t ($t = C[t']$) not in σ (so inspected by pattern p), then $C[r' \{ \sigma' \}] \Rightarrow r \{ \sigma \}$ where \Rightarrow is the parallel reduction (reduction of several redexes at the same time).

However, since PCUIC and the grammar of patterns are richer than the theory of Cockx et al., there are additional concerns we have to consider. First, the pattern syntax is rich enough to allow for β - and ι -redexes, which introduce critical pairs with the rewrite rule being created. One way to fix the issue would be to include these reductions in the triangle criterion, but since the reducts do not fit as patterns, this would be as restrictive as outright forbidding them, the approach we chose. Second, our system includes the sort `SProp` and its conversion rule, so rewriting needs to account for it. This means that patterns whose

$$\begin{array}{c}
 \frac{(\Gamma \vdash ?x : A) \in \Theta}{\Sigma; \Theta; \Gamma \vdash_p ?x \triangleleft A} \text{PATVAR} \qquad \frac{\Sigma; \Theta; \Gamma \vdash_p p \triangleright A \quad \Sigma; \Theta; \Gamma \vdash_p A \leq_p B}{\Sigma; \Theta; \Gamma \vdash_p p \triangleleft B} \text{CONV} \\
 \\
 \frac{(x : A) \in \Gamma}{\Sigma; \Theta; \Gamma \vdash_p x \triangleright A} \text{VAR} \qquad \frac{(C : T) \in \Sigma}{\Sigma; \Theta; \Gamma \vdash_p C \triangleright T} \text{AX-SYMB} \qquad \frac{}{\Sigma; \Theta; \Gamma \vdash_p \square_u^q \triangleright \square_{\uparrow_q u}^{\text{Type}}} \text{SORT} \\
 \\
 \frac{\Sigma; \Theta; \Gamma \vdash_p p \triangleright \forall(x : A), B \quad \Sigma; \Theta; \Gamma \vdash_p a \triangleleft A}{\Sigma; \Theta; \Gamma \vdash_p p a \triangleright B[x := a]} \text{APP} \\
 \\
 \frac{\Sigma; \Theta; \Gamma \vdash_p a \triangleleft \square_u^q \quad \Sigma; \Theta; \Gamma, (x : a) \vdash_p p \triangleright B}{\Sigma; \Theta; \Gamma \vdash_p \lambda(x : a : \square_u^q), p \triangleright \forall(x : a), B} \text{LAMBDA} \\
 \\
 \frac{\Sigma; \Theta; \Gamma \vdash_p a \triangleleft \square_u^q \quad \Sigma; \Theta; \Gamma, (x : a) \vdash_p b \triangleleft \square_{u'}^{q'}}{\Sigma; \Theta; \Gamma \vdash_p \forall(x : a : \square_u^q), b : \square_{u'}^{q'} \triangleright \square_{u *_{q'} u'}^{q'}} \text{FORALL} \\
 \\
 \frac{(\Gamma_p, \Gamma_i, [I : \forall \Gamma_p, \forall \Gamma_i, \square_{u_I}^{q_I}], \vec{\Gamma}_k, [C_k : \forall(\vec{p} : \Gamma_p), \forall \Gamma_k, I \vec{p} \vec{i}_k]) \in \Sigma \quad \Sigma; \Theta; \Gamma \vdash_p p \triangleright I \vec{p} \vec{i} \quad \Sigma; \Theta; \Gamma, (\vec{i} : \Gamma_i), (x : I \vec{p} \vec{i}) \vdash_p a \triangleleft \square_u^q \quad \Sigma; \Theta; \Gamma, \Gamma_k \vdash_p b_k \triangleleft a[\Gamma_i := \vec{i}, x := C_k \vec{p} \vec{i}_k]}{\Sigma; \Theta; \Gamma \vdash_p \text{case } p \text{ return } a : \square_u^q \text{ with } \vec{b}_k \triangleright a[\Gamma_i := \vec{i}, x := p]} \text{CASE}
 \end{array}$$

■ **Figure 5** Type inference of patterns.

type is in **SProp** (or equivalently, patterns which are syntactically irrelevant) can never be inspected reliably since the term could always be swapped by any other term of the same type. Therefore, irrelevant patterns need to be forbidden as well. Note that pattern holes may appear at irrelevant positions, since they do not inspect the term and thus do not conflict with the conversion rule, so the ban is specifically on patterns and not on argument patterns.

Once this is done, the proof of confluence follows exactly the proof done by Cockx et al., which was already a variation on the proof done in [14]. The triangle criterion is used to show that one-step parallel reduction satisfies the triangle lemma saying that for any term t , there exists an optimally reduced term $\rho(t)$ (that performs all possible reductions in parallel) such that $t \Rightarrow \rho(t)$ and for any $t \Rightarrow u$, $u \Rightarrow \rho(t)$. Confluence is then a direct consequence of this triangle lemma and the fact that parallel reduction entails reduction.

Let us now prove postponement of α -cumulativity after reduction. The use of α -cumulativity needs to be moved after all reductions, so we need the following property: $t \rightarrow t' \wedge t \leq_\alpha u \implies \exists u', u \rightarrow^* u' \wedge t' \leq_\alpha u'$ (and the same one if we change the direction of \leq_α). Starting from the proof for regular PCUIC which does an induction on $t \rightarrow t'$, the only additional case of reduction is the application of a rewrite rule. So long as \leq_α consists of congruence rules and cumulativity, there is no issue for reapplying the rewrite rule on u (\leq_α can be postponed after pattern matching and is stable under metasubstitutions). However, when the irrelevance rule of **SProp** is used, we are only able to reapply the rewrite rule if the pattern doesn't inspect the irrelevant subterm. With the restriction introduced above, we ensure that the rewrite rule can be reapplied to u once more.

4.2 Type Inference of Patterns

As mentioned in Section 1, the typing criterion on rewrite rules to ensure subject reduction cannot be solely based on the typing judgment of PCUIC. Actually, what needs to be checked is that the most general type that can be given to a term matching a pattern is a valid type for the right-hand side of the rewrite rule, when its variables also have the most general type allowed by the pattern.

To define this formally, we introduce the notion of type inference in a pattern. This notion is based on bidirectional typing as presented for instance in [10]. Technically, the type of a pattern will be inferred, denoted as $\Sigma; \Theta; \Gamma \vdash_p p \triangleright T$, while the type of an argument pattern will only be checked, denoted as $\Sigma; \Theta; \Gamma \vdash_p a \triangleleft T$. The rules for pattern inference are given in Figure 5.

Most of the rules follow the standard bidirectional discipline. For instance, the type of a variable (**VAR**) is directly inferred from the local context, and the type of an application (**APP**) is inferred by inferring the type for the function and checking the type of the argument pattern against the domain of the function.

Let us remark once again that the sort annotations which correspond to the virtual annotations of terms (e.g., the domain in rule (**FORALL**)) are needed to check the associated type fields, whereas the usual discipline would have been to infer them and coerce them to sorts.

The main specificity with respect to standard bidirectional typing lies in the rule (**PATVAR**) for checking a pattern variable. In our presentation, we consider that Θ is given and mentions exactly the pattern variables occurring in the pattern. The rule then checks that the type mentioned in Θ is exactly the one that is checked. Since our criterion needs for pattern variables to have the most general type they can have, we need this rule to be in checking mode. In an algorithmic presentation, Θ can in fact be constructed solely from the typing of the pattern and can then be considered an output of the typing procedure, along with the inferred type of the whole pattern.

One last crucial rule is (**CONV**), which allows changing bidirectional modes. It makes use of the pattern cumulativity relation, which will be characterised and defined in Section 5, but can for the moment be approximated with regular term cumulativity (they both compare regular terms).

Using this notion of type inference of patterns, we define the following criterion for subjection reduction.

► **Definition 4** (Type-preservation criterion). *A rewrite rule $p \rightsquigarrow r$ is said to be type-preserving in the global environment Σ when there exists a metavariable context Θ and type T_p such that $\Sigma; \Theta; [] \vdash_p p \triangleright T_p$ and $\Sigma; \Theta; [] \vdash r : T_p$.*

We now turn to the proof that this criterion is enough to deduce subject reduction.

4.3 Subject Reduction

In this section, we consider a fixed rewrite rule $p \rightsquigarrow r$ that satisfies the type-preservation criterion in environment Σ_0 for the metavariable environment Θ at type T_p . To show that the type system has subject reduction, we have to show that the rewrite rule is type preserving in all extended environments and all local contexts, which means:

$$\forall \Sigma \supseteq \Sigma_0, \Gamma, t, T, \sigma, \Sigma; []; \Gamma \vdash t : T \wedge t \mid p\{\sigma\} \implies \Sigma; []; \Gamma \vdash r\{\sigma\} : T.$$

We first determine the properties that type inference \vdash_p needs to satisfy before proving the implication. Let us fix an environment $\Sigma \supseteq \Sigma_0$ and a local context Γ .

26:12 The Rewster: Type Preserving Rewrite Rules for the Coq Proof Assistant

1. We need to work under environment Σ instead of Σ_0 . Fortunately, \vdash satisfies environment weakening and we can verify that \vdash_p also satisfies it, so we can really work under assumptions $\Sigma; \Theta; [] \vdash_p p \triangleright T_p$ and $\Sigma; \Theta; [] \vdash r : T_p$.
2. We can use the substitution property of \vdash and the fact that types are well typed to remove all mentions of r in our implication: since $\forall \sigma, \Sigma; []; \Gamma \vdash \sigma : \Theta \implies \Sigma; []; \Gamma \vdash r\{\sigma\} : T_p\{\sigma\}$ and $\Sigma; []; \Gamma \vdash T : \Box_u^q$ for some q and u , implying that $(\Sigma; []; \Gamma \vdash r\{\sigma\} : T_p\{\sigma\} \wedge T_p\{\sigma\} \leq T) \implies \Sigma; []; \Gamma \vdash r\{\sigma\} : T$, it only remains to prove the following anti-substitution lemma:

$$\forall t, T, \sigma, \Sigma; []; \Gamma \vdash t : T \wedge t \mid p\{\sigma\} \implies \Sigma; []; \Gamma \vdash \sigma : \Theta \wedge \Sigma; []; \Gamma \vdash T_p\{\sigma\} \leq T.$$

3. We need σ to completely instantiate Θ , so we make use of the fact that Θ contains exactly one entry per pattern variable of p .
4. Since the types in Θ and T_p can refer to the additional sort annotations in patterns, we need to consider the virtual parts of t and σ in the following.

It is well known that the typing judgment of PCUIC satisfies environment weakening [14]. In fact, our extension with metavariables changes nothing to the proof (the rule on metavariables doesn't mention the environment), so the property still stands in our system. Even better, since our pattern typing judgment resembles regular typing, the same proof can be adapted to work on it, proving that pattern typing also satisfies environment weakening. Let us now prove the anti-substitution lemma.

We first need to introduce more elements and strengthen our property so that the induction can go through. We first split Γ into Γ, Δ and σ into σ_0, σ such that Γ is fixed while Δ is the context extension which corresponds to Δ_p as we go into the pattern and σ_0 corresponds to the already matched portion of the pattern, which will affect Δ and Θ while σ corresponds to the currently matched portion of the pattern. The proof goes by induction on the typing derivation of p , with induction predicate:

$$\Sigma; \Theta; \Delta_p \vdash_p p \triangleright T_p \implies$$

$$\forall \Delta, t, T, \sigma_0, \sigma, \Sigma; []; \Gamma, \Delta \vdash t : T \wedge t \mid p\{\sigma\} \wedge \Delta_p\{\sigma_0\} \equiv \Delta \implies$$

$$\Sigma; []; \Gamma \vdash \sigma : \Theta\{\sigma_0\} \wedge T_p\{\sigma_0, \sigma\} \leq T$$

and the corresponding predicate on argument patterns:

$$\Sigma; \Theta; \Delta_p \vdash_p p \triangleleft T_p \implies$$

$$\forall \Delta, t, T, \sigma_0, \sigma, \Sigma; []; \Gamma, \Delta \vdash t : T \wedge t \mid p\{\sigma\} \wedge T_p\{\sigma_0, \sigma\} \equiv T \wedge \Delta_p\{\sigma_0\} \equiv \Delta \implies$$

$$\Sigma; []; \Gamma \vdash \sigma : \Theta\{\sigma_0\}$$

Let us give a representative subset of all cases needed to prove the induction.

■ Case $?x$:

Hypotheses: $\Sigma; []; \Gamma, \Delta \vdash t : T, \sigma = [?x := t], T_p\{\sigma_0\} \equiv T, \Delta_p\{\sigma_0\} \equiv \Delta$

Goal: $\Sigma; []; \Gamma \vdash (\Delta_p \vdash ?x : T_p)\{\sigma_0, \sigma\}$ which simplifies to $\Sigma; []; \Gamma, \Delta_p\{\sigma_0, \sigma\} \vdash t : T_p\{\sigma_0, \sigma\}$

Proof: Neither Δ_p nor T_p mention $?x$, so they are respectively convertible to Δ and T by hypothesis, which proves our goal.

■ Case $p a$

Hypotheses: $\Sigma; \Theta; \Delta \vdash_p p \triangleright \forall (x : A_p), B_p, \quad \Sigma; \Theta; \Delta \vdash_p a \triangleleft A_p, \quad T_p = B_p[x := a],$

$\Sigma; []; \Gamma, \Delta \vdash f t : T, f \mid p\{\sigma_1\}, t \mid a\{\sigma_2\}, \Delta_p\{\sigma_0\} \equiv \Delta.$

By inversion of typing on an application, there exists A and B such that $\forall(x : A), B \leq T$, $\Sigma; []; \Gamma, \Delta \vdash f : \forall x : A, B$ and $\Sigma; []; \Gamma, \Delta \vdash t : A$.

Then, by induction hypothesis on p , we get $(\forall x : A_p, B_p)\{\sigma_0, \sigma_1\} \leq \forall x : A, B$ and $\Sigma; []; \Gamma \vdash \sigma_1 : \Theta\{\sigma_0\}$.

By the definition of substitution and injectivity of products for \leq , $A_p\{\sigma_0, \sigma_1\} \equiv A$ and $B_p\{\sigma_0, \sigma_1\} \leq B$. Since Δ_p does not contain any metavariable in $|\sigma_1|$, $\Delta_p\{\sigma_0, \sigma_1\} \equiv \Delta$. This means we can use the induction hypothesis on a and get $\Sigma; []; \Gamma \vdash \sigma_2 : \Theta\{\sigma_0, \sigma_1\}$.

Goal: $B_p[x := a]\{\sigma\} \leq B[x := t]$ and $\Sigma; []; \Gamma \vdash \sigma_1, \sigma_2 : \Theta\{\sigma_0\}$. The second goal is immediately proved by concatenating the induction hypotheses.

Proof: by commuting the substitution and metasubstitution,

$$B_p[x := a]\{\sigma\} = B_p\{\sigma\}[x := a\{\sigma\}] \equiv B_p\{\sigma\}[x := t] \leq B[x := t]$$

(a contains only metavariables in $|\sigma_2|$ and $t \mid a\{\sigma_2\}$, so $a\{\sigma\} = a\{\sigma_2\} \equiv t$)

- Case $\forall(x : a : \square_{?u}^{?q}), b : \square_{?u'}^{?q'}$

Hypotheses: $\Sigma; \Theta; \Delta \vdash_p a \triangleleft \square_{?u}^{?q}$, $\Sigma; \Theta; \Delta \vdash_p b \triangleleft \square_{?u'}^{?q'}$, $T_p = \square_{?u * ?q, ?u'}^{?q'}$,

$\Sigma; []; \Gamma, \Delta \vdash \forall(x : A : \square_u^q), (B : \square_{u'}^{q'}) : T$, $A^+ \mid a\{\sigma_1\}$, $B^+ \mid b\{\sigma_2\}$, $\Delta_p\{\sigma_0\} \equiv \Delta$ with $\sigma = [?q := q; ?q' := q'; ?u := u; ?u' := u'] + \sigma_1 + \sigma_2$.

By inversion of typing on a product, we get $\square_{u * q, u'}^{q'} \leq T$, $\Sigma; []; \Gamma, \Delta \vdash A : \square_u^q$ and

$\Sigma; []; \Gamma, \Delta, (x : A) \vdash B : \square_{u'}^{q'}$.

Since Δ_p does not contain any metavariable in $|\sigma_1|$, $(\Delta_p, x : a)\{\sigma_0, \sigma_1\} \equiv \Delta, (x : A)$.

This means we can use both induction hypotheses to get $\Sigma; []; \Gamma \vdash \sigma_1 : \Theta\{\sigma_0\}$ and $\Sigma; []; \Gamma \vdash \sigma_2 : \Theta\{\sigma_0, \sigma_1\}$.

Goal: $\square_{?u * ?q, ?u'}^{?q'}\{\sigma\} \leq \square_{u * q, u'}^{q'}$ and $\Sigma; []; \Gamma \vdash \sigma_1, \sigma_2 : \Theta\{\sigma_0\}$. The second goal is immediately proved by concatenating the induction hypotheses.

Proof: We didn't define yet how $*$ should behave when its quality arguments are variables. Evidently, they need to return the minimal universe, so they must behave as **Prop** to satisfy the required inequality.

- Case **CONV**:

Hypotheses: $\Sigma; []; \Gamma, \Delta \vdash t : T$, $t \mid p\{\sigma\}$, $T'_p\{\sigma_0\} \equiv T$, $\Delta_p\{\sigma_0\} \equiv \Delta$, $T_p\{\sigma\} \leq T$, $T_p \leq_p T'_p$, $\Sigma; []; \Gamma \vdash \sigma : \Theta\{\sigma_0\}$.

Goal: $\Sigma; []; \Gamma \vdash \sigma : \Theta\{\sigma_0\}$. It is one of our hypotheses, our cumulativity hypotheses are useless for now. In Section 5 however, we present the definition of pattern cumulativity to extract some information from these unused hypotheses. Let us simply note that, by symmetry of \equiv and transitivity, $T_p\{\sigma\} \leq T'_p\{\sigma\}$, which we will use alongside $T_p \leq_p T'_p$.

5 Pattern Cumulativity and Equality Extraction

Currently, our criterion is more limited than the one we seek. For instance, it doesn't allow us to infer the type of slightly complex rewrite rules on vectors of natural numbers like

`vtail ?n (vcons ?n' ?a ?v) ~> ?v`

since the pattern has type **vector** $?n$ while the replacement term has type **vector** $?n'$. However, typing ensures that we always have $?n = ?n'$, so we could extract this equality during pattern typing, in the same way that unification is used to collect constraints during the elaboration of a **Coq** term [16]. This way, we can modify the subject reduction criterion to take advantage of these additional conversion hypotheses when checking the type of the right-hand side of the rewrite rule.

$$\begin{array}{c}
 \frac{\Sigma; \Theta; \Gamma \vdash t \rightarrow^* t' \quad \Sigma; \Theta; \Gamma \vdash u \rightarrow^* u' \quad \Sigma; \Theta; \Gamma \vdash_p t' \leq_p u' \triangleright \mathcal{E}}{\Sigma; \Theta; \Gamma \vdash_p t \leq_p u \triangleright \mathcal{E}} \text{RED} \\
 \\
 \frac{C \text{ is injective}}{\Sigma; \Theta; \Gamma \vdash_p C \leq_p C \triangleright []} \text{INJ} \qquad \frac{x \in \Gamma}{\Sigma; \Theta; \Gamma \vdash_p x \leq_p x \triangleright []} \text{REL} \\
 \\
 \frac{\Sigma \vdash q =_q q' \wedge u \leq_u u' \text{ possible}}{\Sigma; \Theta; \Gamma \vdash_p \square_u^q \leq_p \square_{u'}^{q'} \triangleright [u \leq_u u'; q =_q q']} \text{SORT} \\
 \\
 \frac{\Sigma; \Theta; \Gamma \vdash_p A \equiv_p B \triangleright \mathcal{E} \quad \Sigma; \Theta; \Gamma, (x : A) \vdash_p t \leq_p u \triangleright \mathcal{E}' \quad \mathcal{B} \in \{\lambda, \forall\}}{\Sigma; \Theta; \Gamma \vdash_p \mathcal{B}(x : A), t \leq_p \mathcal{B}(x : B), u \triangleright \mathcal{E} \cup \mathcal{E}'} \text{LAMBDA/FORALL} \\
 \\
 \frac{\Sigma; \Theta; \Gamma \vdash_p f \leq_p f' \triangleright \mathcal{E} \quad \Sigma; \Theta; \Gamma \vdash_p a \equiv_p a' \triangleright \mathcal{E}' \quad \Sigma; \Gamma \vdash f, f' \text{ whne}}{\Sigma; \Theta; \Gamma \vdash_p f a \leq_p f' a' \triangleright \mathcal{E} \cup \mathcal{E}'} \text{APP} \\
 \\
 \frac{\Sigma; \Theta; \Gamma, \Gamma_i \vdash_p P \equiv_p P' \triangleright \mathcal{E}_r \quad \begin{array}{c} \Sigma; \Theta; \Gamma \vdash_p c \equiv_p c' \triangleright \mathcal{E}_c \\ [\Sigma; \Theta; \Gamma, \Gamma_k \vdash_p b_k \equiv_p b'_k \triangleright \mathcal{E}_{b,k}] \end{array} \quad \Sigma; \Gamma \vdash c, c' \text{ whne}}{\Sigma; \Theta; \Gamma \vdash_p \text{case } c \text{ return } P \text{ with } \vec{b} \leq_p \text{case } c' \text{ return } P' \text{ with } \vec{b}' \triangleright \mathcal{E}_c \cup \mathcal{E}_r \cup \bigcup_i \mathcal{E}_{b,i}} \text{CASE} \\
 \\
 \frac{}{\Sigma; \Theta; \Gamma \vdash_p ?x \equiv_p t \triangleright [\Delta \vdash ?x := t]} \text{PATVAR} \\
 \\
 \frac{t \text{ or } u \text{ contains a pattern variable or a symbol in head position}}{\Sigma; \Theta; \Gamma \vdash_p t \leq_p u \triangleright []} \text{FAILSAFE} \\
 \\
 \frac{x \in \Gamma}{\Sigma; \Gamma \vdash x \text{ whne}} \quad \frac{\Sigma; \Gamma \vdash f \text{ whne}}{\Sigma; \Gamma \vdash f a \text{ whne}} \quad \frac{\Sigma; \Gamma \vdash c \text{ whne}}{\Sigma; \Gamma \vdash \text{case } c \text{ return } P \text{ with } \vec{b} \text{ whne}}
 \end{array}$$

■ **Figure 6** The rules for conversion judgments \leq_p and \equiv_p (replace all \leq_p with \equiv_p in the rules).

We proceed by defining a notion of pattern conversion and cumulativity (\equiv_p and \leq_p) that collect the set of equalities \mathcal{E} necessarily satisfied by substituted terms: we replace the binary relation $t \leq_p u$ with the ternary relation $t \leq_p u \triangleright \mathcal{E}$, which satisfies the following property:

$$\begin{aligned}
 \forall t, u, \mathcal{E}, \quad \Sigma_0; \Theta; \Delta_p \vdash_p t \leq_p u \triangleright \mathcal{E} &\implies \\
 \forall \Sigma \supseteq \Sigma_0, \Xi, \Gamma, \sigma, \quad \Sigma; \Xi; \Gamma \vdash \sigma : \Theta \wedge \Sigma; \Xi; \Gamma, \Delta_p \{\sigma\} \vdash t \{\sigma\} \leq u \{\sigma\} & \\
 \implies \forall (\Delta_i \vdash t_i \equiv u_i) \in \mathcal{E}, \quad \Sigma; \Xi; \Gamma, \Delta \{\sigma\} \vdash t_i \{\sigma\} \equiv u_i \{\sigma\}. &
 \end{aligned}$$

Note that we are still very free in the implementation of \leq_p because, as long as we don't claim any equality ($\mathcal{E} = []$), even the full relation would be sound. However, we must remember that our primary goal is to extract as many enforced equalities as possible.

Our main objective is thus to apply safe inversions of conversion, to make sure that our equalities necessarily hold:

- If $t_1 \{\sigma\} \equiv u$ and $t_1 \rightarrow^* t_2$, then by reflexivity of \equiv and substitutivity $t_1 \{\sigma\} \equiv t_2 \{\sigma\}$ and thus $t_2 \{\sigma\} \equiv u$. This means that we are allowed to reduce in \equiv_p .

- When a term is in weak head normal form, its congruence rule is invertible. This includes products, sorts, λ -abstractions, inductives, constructors and all weak head neutral terms (whne). Therefore, the general strategy for pattern conversion will be to reduce our arguments to weak head normal form and then to invert the congruence rule of the head constructor.
- Non-injective symbols may be equipped with any rule in the future, so the congruence rules of their elimination context (as would-be neutral terms) are not invertible. This means we cannot extract information in this case and should simply accept with no equality (FAILSAFE). This approach is a way to avoid the failure of subject reduction described in Example 3.
- Metavariables may take any (well-typed) value, so the congruence rules of their elimination context (as would-be neutral terms) are not invertible. However, the conversion can be extracted as an equality. To keep the procedure easily decidable, we only keep the equality when the elimination context is empty ($?x\{\Delta := \Delta\} \equiv t$). Also, if we extract more than one equality for a pattern variable, we will discard all but one.

The definition of $t \leq_p u \triangleright \mathcal{E}$ is formally given in Figure 6. We use this notion of cumulativity on patterns producing equality constraints to extend inference of patterns to also produce a set of equalities: $\Sigma; \Theta; [] \vdash_p p \triangleright T_p, \mathcal{E}$.

In the setting we consider, all equations are of the form $\Delta \vdash ?x := a$ and can then be used in the regular typing judgment by adding reduction $(\Delta \vdash ?x := a) \in \mathcal{E} \implies ?x\{\bar{t}\} \rightarrow a[\Delta := \bar{t}]$ in the definition of cumulativity. We note $\vdash_{\mathcal{E}}$ the typing judgment where cumulativity is extended with the equalities in \mathcal{E} .

► **Definition 5** (Extended type-preservation criterion). *A rewrite rule $p \rightsquigarrow r$ is said to be type-preserving in the global environment Σ when there exists a metavariable context Θ and type T_p and set of equalities \mathcal{E} such that $\Sigma; \Theta; [] \vdash_p p \triangleright T_p, \mathcal{E}$ and $\Sigma; \Theta; [] \vdash_{\mathcal{E}} r : T_p$.*

The proof of subject reduction can be extended easily to this setting.

6 Implementation

Rewrite rules have been implemented and integrated into the Coq proof assistant with [pull request #18038](#). The criterion for type preservation has been implemented but [pull request #19290](#) is yet to be integrated to the Coq proof assistant.

Let us describe a more complete example now possible thanks to rewrite rules: exceptions [12]. Previously, one could define an exception as an axiom and merely state how it behaves against other term constructors, now one can do the following:

```
Symbol raise : ∀ (A : Type), A.
Rewrite Rules raise_rew :=
| raise (∀ (x : ?A), ?B) ?a ~> raise ?B@{x := ?a}
| if raise B as b return ?P then _ else _ ~> raise ?P@{b := raise B}
| match raise N as n return ?P with 0 => _ | S _ => _ end ~> raise ?P@{n := raise N}
| fst (raise (?A * ?B)) ~> raise ?A | snd (raise (?A * ?B)) ~> raise ?B.
```

Note that the second and third rules would currently raise a warning complaining about a potential universe inconsistency, since the return predicates $?P$ could have a universe level larger than the one `raise` expects, but this issue can only happen if one mentions that level explicitly, and we plan to introduce solutions to make this impossible (and thus make the rules safe) in the future.

A few extensions were made from the theory exposed in this paper, the first of which being the support of the full grammar of Coq terms. This includes notably primitive projections (see the rules on the product type in the example above), universe polymorphism and sort polymorphism where universe instances can be matched against.

An extension has been made so that users can make some symbols unfold fixpoints. In Coq, fixpoints are guarded to prevent infinite loops and the guard condition operates on inductive values. This means that fixpoint unfolding may only happen when the guarded argument has a head constructor. However, a user may want to add new values of an inductive type that should behave as a constructor in this regard, for instance an exception as in the example above. To this end, a flag called `unfold_fix` can be given when declaring a symbol:

```
(* Supersedes the declaration shown previously *)
#[unfold_fix] Symbol raise : ∀ (A : Type), A.
(* ... same set of rewrite rules ... *)
Eval compute in raise ℕ + 5. (* raise ℕ *)
```

In practice, metavariables can be used to bind terms on the left-hand side and mention them in the right-hand side, but they are also convenient to avoid describing explicitly sub-patterns that are “forced” and will be found by the extraction of equalities mechanism described in Section 5. To this end, we have added the `_` to produce metavariables that are not relevant to the right-hand side, as the two branches of the example above.

In this setting, another extension has been added to the type preservation criterion to mitigate the inference requirements for application and cases: while the theory requires that the pattern infers respectively a product and an inductive, it is in fact safe to allow inferring a pattern variable. This way, the user can still put `_` at places where a product or an inductive is expected, and we automatically add an equality between this implicit pattern variable and the product/inductive with “fresh” pattern variables (respectively for the domain/codomain and for the arguments to the inductive). Since we cannot generate fresh variables during type checking, these additional pattern variables have to be added to the pattern from the start, giving for application a virtual annotation $(p : \forall(x : ?A), ?B) a$, with typing rule

$$\frac{\Sigma; \Theta; \Gamma \vdash p \triangleleft \forall(x : ?A), ?B \quad \Sigma; \Theta; \Gamma \vdash a \triangleleft ?A}{\Sigma; \Theta; \Gamma \vdash (p : \forall(x : ?A), ?B) a \triangleright ?B\{x := a\}}$$

7 Related Work

Cockx et al. [3] introduce a system for rewrite rules in dependently typed λ -calculus, which forms the foundation for this work. Their system employs simpler rewrite rules (lacking higher-order rules and cumulativity) and doesn’t provide a criterion for type preservation. In this paper, we extend their setting with higher-order rewrite rules in presence of cumulativity and definitional proof-irrelevance. We also provide a decidable criterion to ensure type preservation which has been implemented in the Coq proof assistant. It is important to note that the argument by Cockx et al. regarding the consistency of the theory extended with rewrite rules, under the assumption that these rules are admissible as propositional equalities, also applies to our system. This is because their argument relies on an encoding in extensional type theory, and does not depend on the complexity or richness of the rewrite rules syntax.

In Andromeda 2, users manually describe definitional equality by providing equality judgments, which must either be extensional equalities or get interpreted as rewrite rules [1]. During this transformation, the system checks type preservation in a manner similar to ours, but since their system is more permissive, it has fewer guarantees (such as substitutivity which always holds for us) and thus needs to check equalities before applying a rewrite rule.

Felicissimo introduces a bidirectional system with rewriting [6], where rewrite rules are also checked to preserve typing. Both systems are close, but our pattern grammar is more expressive, we have fewer constraints on erased arguments (e.g., our virtual sort annotations can't be recovered from the type of the products) and we extract equalities from the typing of the pattern. The bidirectional approach has some interesting advantages such as the possibility to implement rewrite rules that would typically be non-left-linear. In the case of Coq it would require a redesign of the syntax and thus it's not clear how applicable it would be to our case.

Blanqui's decision procedure for type safety in rewrite rules [2] also checks type preservation and extracts equalities in a simpler theory, the $\lambda\Pi$ -calculus, thus without a (cumulative) hierarchy of universes or inductive types. In particular, there is no notion of subtyping and thus no need for bidirectional inference. However, this simpler setting makes it possible to extract more enforced equalities from the pattern.

8 Conclusion and Future Work

We introduced and implemented a new framework for users to define their own rewrite rules within the Coq proof assistant. Additionally, we developed a method to ensure these user-defined rules maintain type safety, guaranteeing the integrity of Coq's core functionalities like subject reduction and completeness of type-checking.

From a practical point of view, our most immediate goal is to port to Coq the confluence checker that has been written by Jesper Cockx for Agda, to improve the safety of rewrite rules. Another meaningful extension is to define a notion of justified rewrite rules, which corresponds to the mapping of existing terms to the declaration of symbols and a proof of propositional equality on those terms to rewrite rules on those symbols. For instance, `pp1us` is justified with the standard notion of addition. This way, justified rewrite rules can be considered as a safe extension instead of their current status of additional axioms.

References

- 1 Andrej Bauer and Anja Petković Komel. An extensible equality checking algorithm for dependent type theories. *Logical Methods in Computer Science*, Volume 18, Issue 1, January 2022. doi:10.46298/lmcs-18(1:17)2022.
- 2 Frédéric Blanqui. Type Safety of Rewrite Rules in Dependent Types. In *5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020)*. Schloss-Dagstuhl - Leibniz Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.FSCD.2020.13.
- 3 Jesper Cockx, Nicolas Tabareau, and Théo Winterhalter. The taming of the rew: A type theory with computational assumptions. *Proceedings of the ACM on Programming Languages*, 5(POPL):60:1–60:29, January 2021. doi:10.1145/3434341.
- 4 Thierry Coquand and Christine Paulin. Inductively defined types. In Per Martin-Löf and Grigori Mints, editors, *COLOG-88*, pages 50–66, Berlin, Heidelberg, 1990. Springer. doi:10.1007/3-540-52335-9_47.
- 5 Leonardo de Moura and Sebastian Ullrich. The Lean 4 Theorem Prover and Programming Language. In André Platzer and Geoff Sutcliffe, editors, *Automated Deduction – CADE 28*, pages

- 625–635, Cham, 2021. Springer International Publishing. doi:10.1007/978-3-030-79876-5_37.
- 6 Thiago Felicissimo. Generic bidirectional typing for dependent type theories. In Stephanie Weirich, editor, *Programming Languages and Systems*, pages 143–170, Cham, 2024. Springer Nature Switzerland. doi:10.1007/978-3-031-57262-3_6.
 - 7 Gaëtan Gilbert, Jesper Cockx, Matthieu Sozeau, and Nicolas Tabareau. Definitional proof-irrelevance without K. *Proceedings of the ACM on Programming Languages*, 3(POPL):3:1–3:28, January 2019. doi:10.1145/3290316.
 - 8 Jean-Yves Girard. *Interprétation Fonctionnelle et Élimination Des Coupures de l'arithmétique d'ordre Supérieur*. PhD thesis, Université Paris VII, 1972.
 - 9 Antonius J. C. Hurkens. A simplification of Girard's paradox. In Mariangiola Dezani-Ciancaglini and Gordon Plotkin, editors, *Typed Lambda Calculi and Applications*, pages 266–278, Berlin, Heidelberg, 1995. Springer. doi:10.1007/BFb0014058.
 - 10 Meven Lennon-Bertrand. Complete Bidirectional Typing for the Calculus of Inductive Constructions. In *DROPS-IDN/v2/Document/10.4230/LIPIcs.ITP.2021.24*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.ITP.2021.24.
 - 11 Ulf Norell. *Towards a Practical Programming Language Based on Dependent Type Theory*, volume 32. Citeseer, 2007.
 - 12 Pierre-Marie Pédrot and Nicolas Tabareau. Failure is not an option an exceptional type theory. In *ESOP 2018 - 27th European Symposium on Programming*, volume 10801 of *LNCS*, pages 245–271, Thessaloniki, Greece, April 2018. Springer. doi:10.1007/978-3-319-89884-1_9.
 - 13 Matthieu Sozeau, Abhishek Anand, Simon Boulrier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau, and Théo Winterhalter. The MetaCoq Project. *Journal of Automated Reasoning*, 2020. doi:10.1007/s10817-019-09540-0.
 - 14 Matthieu Sozeau, Simon Boulrier, Yannick Forster, Nicolas Tabareau, and Théo Winterhalter. Coq Coq correct! Verification of type checking and erasure for Coq, in Coq. *PACMPL*, 4(POPL), 2020. doi:10.1145/3371076.
 - 15 The Coq Development Team. The Coq Proof Assistant. Zenodo, June 2023. doi:10.5281/zenodo.8161141.
 - 16 Beta Ziliani and Matthieu Sozeau. A unification algorithm for Coq featuring universe polymorphism and overloading. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, pages 179–191, New York, NY, USA, August 2015. Association for Computing Machinery. doi:10.1145/2784731.2784751.

Teaching Mathematics Using Lean and Controlled Natural Language

Patrick Massot  

Université Paris-Saclay, France

Carnegie Mellon University, Pittsburgh, PA, USA

Abstract

We present Verbose Lean, a library using the flexibility of the Lean programming language and proof assistant to teach undergrad mathematics students how to read and write traditional paper proofs. After explaining our main pedagogical goals, we explain how students use the library with varying levels of assistance to write proofs that are easy to transfer to paper because they look like natural language. We then describe how teachers can customize the student experience based on their specific pedagogical goals and constraints. Finally we describe some aspects of the implementation of the library, emphasizing how new aspects of the very recently released version 4 of Lean allow a lot of flexibility that could benefit many new creative uses of a proof assistant.

2012 ACM Subject Classification Human-centered computing → Natural language interfaces; Applied computing → Interactive learning environments; Theory of computation → Logic and verification

Keywords and phrases mathematics teaching, proof assistant, controlled natural language

Digital Object Identifier 10.4230/LIPIcs.ITP.2024.27

Funding *Patrick Massot*: Partially funded by the Hoskinson center for formalized mathematics.

Acknowledgements The library described here is the result of work spread over five years and benefited from many interactions. Marie-Anne Poursat made it possible by trusting me to teach using a proof assistant. Frédéric Bourgeois and Christine Paulin-Mohring later joined me in teaching this course and discussing how to make it more useful. Our students also participated with their patience, questions and enthusiasm. Very recently, Julien Narboux, Pierre Boutry and Evmorfia-Iro Bartzia started using this library with students and made valuable comments. Simon Hudon started my Lean meta-programming education. It was then continued by Mario Carneiro and Kyle Miller who both also directly contributed crucial code that I would have been incapable of writing. Wojciech Nawrocki helped me a lot with widgets, adding several features to his ProofWidgets library for the needs of this project. More generally many people from the Mathlib community contributed indirectly by formalizing basic mathematics and creating tactics that we built on. I also benefited from conversations about teaching using a proof assistant, particularly with Heather Macbeth, Jim Portegies and Jelle Wemmenhove. Most of the work on the Lean 4 version was made possible by Jeremy Avigad's invitation to benefit from some of Charles Hoskinson's generous donation to CMU. And of course none of this would exist without the amazing work of Leonardo de Moura and Sebastian Ullrich on Lean 4.

1 Introduction

The transition from high-school mathematics to proof-based university mathematics is a well-known challenge for students. In the recent past, there have been several experiments using proof assistants to help students in this transition [1, 2, 4, 7, 12, 16, 17]. Here we really mean courses that consider the proof assistant only as an intermediate tool, not as a final goal. Note this tool is applicable to any kind of mathematics in principle, but this account and the library it describes are biased towards elementary real analysis which is used in France as the main introduction to rigorous proofs.



© Patrick Massot;

licensed under Creative Commons License CC-BY 4.0

15th International Conference on Interactive Theorem Proving (ITP 2024).

Editors: Yves Bertot, Temur Kutsia, and Michael Norrish; Article No. 27; pp. 27:1–27:19

Leibniz International Proceedings in Informatics



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

Verbose Lean¹ is a teaching library and meta library for the Lean programming language and proof assistant. Lean is due mostly to Leonardo de Moura at Microsoft Research and then Amazon Web Service and the Lean Focussed Research Organization². Our library has three main layers. The bottom one is a set of tactics (i.e., proof producing programs) mimicking the granularity of proofs on paper. The middle one is a controlled natural language syntax whose goal is to ease the transition to paper proofs, at the cost of being slightly more difficult to write. The third layer is made of mechanisms that help students to write proofs by suggesting potential next steps. All layers are customizable, even without programming knowledge, and everything exists in French and English and is translatable to other languages.

This paper is intended for mathematics teachers and for people who want to see examples of using Lean’s flexibility. It is organized roughly in order of decreasing importance. We first explain our pedagogical goals, then describe the student experience, then the teacher experience before concluding with some aspects of the implementation. That last section is more technical but could be useful to anyone interested in what can be done with Lean.

2 Main pedagogical goals

The first main goal is to train students to have a clear view of the current state of the proof: what is currently being proven, what are the current assumptions, which mathematical objects are fixed. The next pedagogical goal is to train students to automatically perform proof steps depending of the syntactic structure of the current goal. For instance, a direct proof of a universally quantified statement starts with fixing an object of the relevant type.

A basic requirement here is to make sure that every statement has a clear status: is it something that is known to be true? Something that we assume? Something that will be proven soon? We claim that this goal is much easier to achieve if there is a clear separation between stating, proving and using. For any given logical operator or quantifier, there are syntactic rules that explain how to form a mathematical statement using it together with some previously existing statements. Then there are rules that explain how to prove such a statement. Finally there are rules about how to use such a statement. This distinction seems obvious, but in practice it is very blurred, both by students and by professional mathematicians. Of course the difference is that mathematicians know what they are doing even when they are very sloppy about this distinction. We think that enforcing a clear distinction is very useful for beginners, although it can feel pedantic to more advanced people. One of the most frequent cases is failing to distinguish between stating the existence of a mathematical object satisfying some condition and fixing a witness. An example would be to say or write “since f is continuous and ε is positive, there exists δ such that...” and then refer to some δ on the following line. The other very common one is more tied to using symbols instead of text. It uses the implication symbol as an abbreviation of the word “hence” or “therefore”, and more generally mixing using an implication and stating one. An example would be writing on a blackboard “ f is polynomial $\implies f$ is continuous” instead of “ f is polynomial hence continuous”. This misuse of a symbol is extremely problematic for beginners since this alternative meaning of the symbol turns every legitimate use into nonsense, and beginners lack the expertise required to understand which meaning is used. For instance reading the definition of convergence of a function f towards a number l at some point x_0 with this interpretation for the implication symbol gives “for every positive ε ,

¹ <https://github.com/PatrickMassot/verbose-lean4>

² <https://lean-fro.org/>

there exists some positive δ such that for every x , $|x - x_0| < \delta$ hence $|f(x) - l| < \varepsilon$ ". A less ubiquitous but still common example is a confusion between stating a claim starting with a universal quantifier and beginning a proof of such a statement.

All those logical errors (or at least sloppy writing) also impact achieving a third pedagogical goal which is to train students to make a clear distinction between bound and free variables. This is very related to keeping track of what is fixed in the current state of the proof, but with the additional twist that some variable name can appear simultaneously as the name of a free variable and as a bound variable in some assumption or in the current target statement.

The next teaching goal that we want to achieve with this technology is to train students to classify proof steps into safe reversible steps that require no initiative and risky irreversible steps that require some creativity. Here irreversible means it can lead to a goal that is not provable. An example in the first category is fixing a positive number at the beginning of a proof using the definition of continuity, or extracting a witness out of an existential assumption. An example in the second category would be announcing an existential witness or specializing a universally quantified assumption to a unique object. The alternation between routine safe steps and risky creative steps is a crucial aspect of mathematical proof creation. This is not necessarily emphasized when presenting a proof on a blackboard.

On top of that there is a final goal which is to learn how to choose indirect strategies instead of simply following the syntax of the current target. Those includes stating and proving an intermediate fact, using a lemma instead of re-proving everything, and the use of the excluded middle axiom, e.g. through proofs by contradiction or contraposition.

The usual interfaces of proof assistants have many qualities that help achieving those goals. Near the beginning of the proof of sequential continuity from continuity, the proof assistant can display something like:

```
f : ℝ → ℝ
u : ℕ → ℝ
x₀ : ℝ
hu : u converges to x₀
hf : f is continuous at x₀
ε : ℝ
ε_pos : ε > 0
δ : ℝ
δ_pos : δ > 0
hδ : ∀ (x : ℝ), |x - x₀| ≤ δ ⇒ |f x - f x₀| ≤ ε
⊢ ∃ N, ∀ n ≥ N, |(f ∘ u) n - f x₀| ≤ ε
```

This display is called the *tactic state*. Most lines describe either a mathematical object that is fixed in the proof or an assumption. The last line shows the current goal.

This is already tremendously useful to students, and completely impractical to emulate on a blackboard or in print. But there are also many challenges. The most obvious one is the need to learn the syntax of the software. In order to progress in proofs, one need to use *tactics* that are commands which usually do not look like mathematics but rather like programming. This is not a crucial problem, especially with students who also learn programming in other courses. But it does take time, so this issue prevents using a proof assistant on the side of a regular course with no dedicated time. A much more serious issue is that having a proof assistant syntax that is very different from paper proofs makes it a lot harder to transfer proving skills to paper. This is true in the direct case of transcribing a proof done on the computer but also in the longer run when students try to prove things on paper without writing a formal proof first.

A second challenge is to set up the right kind of automation. Traditional paper proofs are very far from mentioning every lemma that is used. Mentioning every lemma systematically is counter-productive with respect to the goals listed above. An extreme example would be lemmas asserting the commutativity and associativity of addition and multiplication of numbers. More generally, things that are too obvious to be mentioned on paper should be done automatically by the proof assistant, whereas things that we want to see justified on paper should not. But this is an extremely vague specification. In practice it is even often inconsistent because it is pretty difficult to be consistent about what we want to see on paper. Powerful automation is also potentially problematic for students who have very slow computers with little memory, and it can make error reporting more complicated. But it is crucial for the success of that kind of use of proof assistants in teaching.

A last challenge is that most countries do not use English as their main language. This is especially relevant for very young students.

3 Using the library as a student

3.1 Tactic language

In this section we will show what Verbose Lean looks like from the point of view of student users. We will show several possible variations but it is probably unwise to show all those variations to students. The following is a typical exercise solution.

```
Exercise "Continuity implies sequential continuity"
  Given: (f : ℝ → ℝ) (u : ℕ → ℝ) (x₀ : ℝ)
  Assume: (hu : u converges to x₀) (hf : f is continuous at x₀)
  Conclusion: (f ∘ u) converges to f x₀
Proof:
  Let's prove that  $\forall \varepsilon > 0, \exists N, \forall n \geq N, |f (u n) - f x_0| \leq \varepsilon$ 
  Fix  $\varepsilon > 0$ 
  By hf applied to  $\varepsilon$  using that  $\varepsilon > 0$  we get  $\delta$  such that
     $\delta_{\text{pos}} : \delta > 0$  and  $Hf : \forall x, |x - x_0| \leq \delta \Rightarrow |f x - f x_0| \leq \varepsilon$ 
  By hu applied to  $\delta$  using that  $\delta > 0$  we get  $N$  such that
     $Hu : \forall n \geq N, |u n - x_0| \leq \delta$ 
  Let's prove that  $N$  works :  $\forall n \geq N, |f (u n) - f x_0| \leq \varepsilon$ 
  Fix  $n \geq N$ 
  By Hf applied to  $u n$  it suffices to prove  $|u n - x_0| \leq \delta$ 
  We conclude by Hu applied to  $n$  using that  $n \geq N$ 
QED
```

In this simple proof, all steps correspond directly to ordinary Lean tactics. Hence we can compare with the following code that builds exactly the same proof.

```
example (f : ℝ → ℝ) (u : ℕ → ℝ) (x₀ : ℝ) (hu : u converges to x₀)
  (hf : f is continuous at x₀) : (f ∘ u) converges to f x₀ := by
  change  $\forall \varepsilon > 0, \exists N, \forall n \geq N, |(f \circ u) n - f x_0| \leq \varepsilon$ 
  intro  $\varepsilon \varepsilon_{\text{pos}}$ 
  rcases hf  $\varepsilon \varepsilon_{\text{pos}}$  with  $\langle \delta, \delta_{\text{pos}}, Hf \rangle$ 
  rcases hu  $\delta \delta_{\text{pos}}$  with  $\langle N, Hu \rangle$ 
  use N
  intro n n_ge
  suffices  $|u n - x_0| \leq \delta$  from Hf (u n) this
  exact Hu n n_ge
```


The first difference with the default syntax of Lean is that the statement clearly distinguishes the objects, the assumptions and the conclusion. Then each proof line looks like natural mathematical language, but it is actually as rigid as any programming language. Remember the goal of this language is not to be easier to write, it is to be easier to transfer to paper.

The first line is completely optional, it only unfolds a definition. The second line shows how bounded quantifiers are handled by the library. The core logic of Lean does not involve those quantifiers. Given a predicate P , say on real numbers, the statement $\forall \varepsilon > 0, P(\varepsilon)$ is a notation for $\forall \varepsilon, \varepsilon > 0 \implies P(\varepsilon)$. In Verbose Lean, introducing a positive number can be done in one step (of course it can also be done in two steps). This generates a name `ε_pos` for the assumption that ε is positive. The next tactic, that spans the third and fourth lines above, uses that positivity but in a declarative way. However it does use the name `hf` that was assigned to the continuity assumption on f . We will refer to the approach that states claims without referring to names of assumptions as the nameless approach. A syntactic variant here would be to write

```
Since f is continuous at x0 and ε > 0 we get δ such that
  δ_pos : δ > 0 and Hf : ∀ x, |x - x0| ≤ δ ⇒ |f x - f x0| ≤ ε
```

which only list claims and does not even explicitly mention ε before mentioning its positivity.

Another important style choice is the use of backward reasoning at the end, witnessed by the words “it suffices to prove”. One could also replace the last two lines with

```
Since ∀ n ≥ N, |u n - x0| ≤ δ and n ≥ N we get h : |u n - x0| ≤ δ
Since ∀ x, |x - x0| ≤ δ → |f x - f x0| ≤ ε and |u n - x0| ≤ δ we
  conclude that |f (u n) - f x0| ≤ ε
```

which uses both the nameless approach and forward reasoning only (those two aspects are independent, we gathered them only to prevent a combinatorial explosion of examples). The nameless approach is not purely stylistic, it also involves some implicit reasoning. For instance, with the same assumptions, we could write `since ε ≥ 0` and the library would silently derive this from $\varepsilon > 0$.

The example used so far only uses two definitions and the rules of logic. It features both using and proving statements formed using quantifiers and implication. Stating a universally quantified claim on line one and starting its proof on line 2 are very distinct operations. The first one involves the quantifier symbol while the second one involves the word “Fix”.

The distinction between stating existence and extracting a witness is handled in a more subtle way. We could first state the existence using the symbols $\exists \delta$ and then have something like “Let us fix such a δ ”. Nothing prevents a teacher from implementing this syntax, but the trick of saying `we get δ such that` is a very nice compromise which distinguishes fixing a witness from merely stating existence and which stays very light to read.

The distinction between claiming an implication and using it is handled very simply. First the implication symbol is used only when stating. Then both backward and forward uses of implication mention both the implication and its premise. This applies both to the style referring to assumption names and to the nameless style.

Let us now consider another example: proving the squeeze theorem.

```
Example "The squeeze theorem."
Given: (u v w : ℕ → ℝ) (l : ℝ)
Assume: (hu : u converges to l) (hw : w converges to l)
        (h : ∀ n, u n ≤ v n) (h' : ∀ n, v n ≤ w n)
Conclusion: v converges to l
```

```

Proof:
Fix  $\varepsilon > 0$ 
Since  $u$  converges to  $l$  and  $\varepsilon > 0$  we get  $N$  such that
   $hN : \forall n \geq N, |u\ n - l| \leq \varepsilon$ 
Since  $w$  converges to  $l$  and  $\varepsilon > 0$  we get  $N'$  such that
   $hN' : \forall n \geq N', |w\ n - l| \leq \varepsilon$ 
Let's prove that  $\max N\ N'$  works :  $\forall n \geq \max N\ N', |v\ n - l| \leq \varepsilon$ 
Fix  $n \geq \max N\ N'$ 
Since  $n \geq \max N\ N'$  we get  $hn : n \geq N$  and  $hn' : n \geq N'$ 
Since  $\forall n \geq N, |u\ n - l| \leq \varepsilon$  and  $n \geq N$  we get
   $hNl : -\varepsilon \leq u\ n - l$  and  $hNd : u\ n - l \leq \varepsilon$ 
Since  $\forall n \geq N', |w\ n - l| \leq \varepsilon$  and  $n \geq N'$  we get
   $hN'l : -\varepsilon \leq w\ n - l$  and  $hN'd : w\ n - l \leq \varepsilon$ 
Let's prove that  $|v\ n - l| \leq \varepsilon$ 
Let's first prove that  $-\varepsilon \leq v\ n - l$ 
Calc  $-\varepsilon \leq u\ n - l$  by assumption
       $- \leq v\ n - l$  since  $u\ n \leq v\ n$ 
Let's now prove that  $v\ n - l \leq \varepsilon$ 
Calc  $v\ n - l \leq w\ n - l$  since  $v\ n \leq w\ n$ 
       $- \leq \varepsilon$  by assumption
QED

```

The beginning of the proof uses the same tactics as our first example. New things start with the line `Since $n \geq \max N\ N'$ we get $hn : n \geq N$ and $hn' : n \geq N'$` . Here we are using a lemma claiming that $n \geq \max(N, N')$ implies that $n \geq N$ and $n \geq N'$. But this lemma is not mentioned explicitly. The tactic saw that the claim $n \geq \max(N, N')$ is not a conjunction so it tried splitting it into the announced conclusions using so-called anonymous fact splitting lemmas. Here anonymous refers to the fact that their names do not appear in the proof script, but of course they actually do have names. The next two tactics (each spanning two lines) use the exact same mechanism using an anonymous lemma that splits an inequality with shape $|x| \leq y$ into $-y \leq x$ and $x \leq y$.

The next tactic `Let's prove that $|v\ n - l| \leq \varepsilon$` is completely optional, it recalls what is the current goal since it was never explicitly spelled out and we just went through three tactics that created new facts without changing current goal. This tactic could also have been used right after `Fix $n \geq \max N\ N'$` .

The next line is something new: `Let's first prove that $-\varepsilon \leq v\ n - l$` . This tactic can be used to start a conjunction proof. But here the current goal is not a conjunction, it is turned into a conjunction by a so-called anonymous goal splitting lemma, which happens to be the converse of the anonymous fact splitting lemma used before (but those are completely separate lemmas from the framework's point of view).

This tactic does a bit more than applying the lemma and splitting the resulting conjunction. Indeed we want to force students to announce the second part of the conjunction when the first one is proven. So instead of showing directly the second goal, the tactic state displays: `You need to announce: Let's now prove that $v\ n - l \leq \varepsilon$` and refuses³ any other tactic.

Returning to what happens during the proof of the first inequality, we see some computation introduced by the `Calc` word. This is based on the builtin Lean `calc` tactic, but the justifications are specific to our library. The first one in the example is `by assumption` which

³ We were hesitant to make this mandatory, but seeing it in the Coq waterproof project convinced us.

implicitly refers to the `hN1` assumption. A more explicit justification could be `from hN1`. What comes after the word `from` could also contain the words `applied to` and `using that` as in the third tactic of our first example. The next justification uses `since` which indicates a nameless approach: we claim that $u\ n \leq v\ n$ without explaining why; the tactic has to instantiate the assumption `h` to the free variable `n`. But there is more to it since this fact by itself is not sufficient to justify that calculation step. The tactic has to secretly invoke a lemma saying that $\forall x, y, z, x \leq y \implies x - z \leq y - z$. This is handled internally by calling the `gcongr` tactic of Carneiro and Macbeth.

Those two examples illustrate the main mechanisms that we use to get students to develop proof skills that are easier to transfer to paper than using the native Lean tactics. Of course they do not exhaust the list of tactics provided by our library. In particular there are tactics that allow to prove things by case disjunctions, using contraposition or proof by contradiction, or using the axiom of choice.

3.2 Assisted modes

The above examples can all be typed in the editor (typically VSCode to avoid teaching at the same time how to use Lean and a powerful editor such as vim or emacs). Lean then answers by updating the proof state and displaying error messages if needed. But mastering a lot of syntax is challenging, even if only one proof style is taught (for instance only the nameless variant that do not use assumption names). So Verbose Lean offers two levels of assistance.

The first level is the `help` tactic that can be used inside the proof. For instance, if the current target is `(f ∘ u) converges to f x0` as at the beginning of our first example then the `help` tactic displays:

```
Help
· The goal starts with the application of a definition.
  One can make it explicit with:
  Let's prove that  $\forall \varepsilon > 0, \exists N, \forall n \geq N, |(f \circ u)\ n - f\ x_0| \leq \varepsilon$ 
· The goal starts with " $\forall \varepsilon > 0$ "
  Hence a direct proof starts with:
  Fix  $\varepsilon > 0$ 
```

The above has two help messages and two suggestions. Clicking on a suggestion replaces the help tactic with the suggestion.

In the same example, there is a local assumption named `hu` saying that u converges to x_0 . The answer to `help hu` is

```
Help
· This assumption starts with the application of a definition.
  One can make it explicit with:
  We reformulate hu as  $\forall \varepsilon > 0, \exists N, \forall n \geq N, |u\ n - x_0| \leq \varepsilon$ 
· The assumption hu starts with " $\forall \varepsilon > 0, \exists N, \dots$ "
  One can use it with:
  By hu applied to  $\varepsilon_0$  using  $h\varepsilon_0$  we get  $N$  such that
  ( $hN : \forall n \geq N, |u\ n - x_0| \leq \varepsilon_0$ )
  where  $\varepsilon_0$  is a real number and  $h\varepsilon_0$  is a proof of the fact that  $\varepsilon_0 > 0$ 
  The names  $N$  and  $hN$  can be chosen freely among available names.
```

Using this tactic with students suggests it already does a lot to tame the syntactic complexity of our controlled natural language. One could fear that students will constantly use it instead of analysing the goal or assumptions themselves, but this was not observed.

Especially in situations where there is not much time allocated to the use of a proof assistant, one can use a more assisted mode where proofs can be assembled at least partly through clicking. In this interaction mode, students click on expressions in the tactic state and get tactics suggestions in return. This subsumes the help tactic: when clicking on the full target or on a full assumption, the suggestions that are given are the same that appear in the help command (assuming the default configuration is used). But one can also click on multiple assumptions, or on sub-expressions inside the target or inside an assumption.

For instance the example that proved sequential continuity from continuity can be done entirely by clicking. Clicking on the full goal suggests the first two lines of the proof. Then one needs to specialize the continuity assumption to the positive ε that was just introduced. This is done by clicking on the assumption and then clicking on ε . With this selection in the tactic state, one gets the following suggestions:

```
· By hf applied to  $\varepsilon$  using that  $\varepsilon > 0$  we get  $\delta$  such that ( $\delta\_pos : \delta > 0$ )
  ( $h\delta : \forall (x : \mathbb{R}), |x - x_0| \leq \delta \Rightarrow |f x - f x_0| \leq \varepsilon$ )
· By hf applied to  $\varepsilon / 2$  using that  $\varepsilon / 2 > 0$  we get  $\delta$  such that ( $\delta\_pos : \delta > 0$ )
  ( $h\delta : \forall (x : \mathbb{R}), |x - x_0| \leq \delta \Rightarrow |f x - f x_0| \leq \varepsilon / 2$ )
```

Those two suggestions have the same shape but use either ε or $\varepsilon/2$ since specializing to half a given number is a very common move in elementary analysis and the default configuration is biased towards this kind of mathematics.

In the above example, the library does not check that it will be able to automatically prove the positivity side condition that appears after `using that`. This lets students judge the different suggestions.

4 Using the library as a teacher

4.1 Basic configuration

In this section we explain various mechanisms used for configuration which do not require programming expertise from teachers (of course a lot more is possible with programming). The goal is not to document every configuration possibility – since this is not a manual – but to show the configuration mechanisms that we use. We also show how this configuration depends on specific pedagogical goals, students expertise and time constraints.

The first decision to make is how much automation, if any, is desired when implicitly using lemmas. As explained in the previous section, there are two kinds of such lemmas, depending on whether they split a given fact or the current goal. For instance lemmas in the second category can be configured using `configureGoalSplittingLemmas Iff.intro Subset.antisymm`. Listing a lot of lemmas in these commands could become very tedious. So we allow defining lemmas lists and referring to them in the configuration commands. We also pre-define some lists. Defining a list named `MyList` which contains the pre-defined list `LogicIntros` and the lemma proving set equality from double inclusion is done using

```
AnonymousGoalSplittingLemmasList MyList := LogicIntros Subset.antisymm
```

Those commands and all configuration commands are meant to be “hidden” in the teacher file. When changes are needed in the middle of an exercise file, one can use a macro. For instance `macro "switchConf" : command => `(configureGoalSplittingLemmas x)` would allow to simply write `switchConf` between two exercises to avoid a long distracting line. Note that teachers will probably want to tell students about the list of anonymous lemmas, at least informally, in order to avoid creating confusion about what needs to be justified.

Tactics can also have configuration flags. For instance, say the goal is $\neg \exists x:\mathbb{Q}, x^2 = 2$. By default, starting the proof with `Assume for contradiction H : $\exists x : \mathbb{Q}, x^2 = 2$` will lead to the error message: “The goal is a negation, there is no point in proving it by contradiction. You can directly assume $\exists x : \mathbb{Q}, x^2 = 2$ ”. Teachers who fully embrace the confusion between a direct proof of a negation and a proof by contradiction, or simply need to focus on other topics, can use `allowProvingNegationsByContradiction`.

The next thing to configure is the assisted modes (help tactic and suggestion widget). First there are commands to disable those modes for teachers who want students to write everything by hand (in an exam setting, one can simply delete the relevant file for extra safety). Assuming they are enabled, many aspects are configurable. Each help message comes from a function, and one can configure the available functions using the same kind of lists as with anonymous lemmas. For instance, in a basic course which progressively introduces different kinds of reasoning, one can disable messages suggesting a proof by contradiction in the beginning. One can also modify existing help functions with no programming knowledge by copy-pasting and editing only the text.

If the current lecture focuses on students knowing definitions then one can completely disable the help that unfolds definitions. One can also control in detail which definitions participate in unfolding suggestions. This has to be an opt-in mechanism to avoid having suggestions unfolding fundamental definitions such as the definition of real numbers or even the definition of addition of natural numbers. For instance the teacher file could contain: `configureUnfoldableDefs continuous seq_limit` assuming the teacher library defined `continuous` and `seq_limit`.

The suggestion widget is also fully configurable. Really changing its behavior requires programming. But an easy tweak is to change how to produce data from the selection. We saw that selecting a real number ε and a universally quantified assumption h does not only propose to specialize h to ε but also to $\varepsilon/2$. The configuration for this can look like:

```
dataProvider mkSelf a := a
dataProvider mkHalf a := a/2
dataProvider mkMin a b := min a b
dataProvider mkMax a b := max a b
DataProviderList CommonProviders := mkSelf mkMin mkMax

configureDataProviders {
  ℝ : [CommonProviders, mkHalf],
  ℕ : [CommonProviders] }
```

In addition to a declaration list `CommonProviders` analogous to the one we use for anonymous lemmas, there are two new kinds of micro-DSLs (domain specific languages) here. First we define four “functions” with the `dataProvider` command which features no type information at all. Those are purely syntactic objects. They only participate in creating the widget suggestions on the syntactic level. Writing meaningless functions there would of course create trouble when accepting suggestions. Finally there is a JSON-like syntax in the `configureDataProviders` that registers data providers for different types. The goal of those DSL is to allow configuring this even for teachers who basically know nothing about Lean, maybe not even enough to write a function that can perform an algebraic operation either on natural numbers or on reals.

All the configuration options mentioned so far are specific to the `Verbose` library, but of course they come on top of the usual Lean configuration. A lot of the flexibility offered by Lean out of the box is very relevant to the kind of teaching targeted by our library.

27:10 Teaching Mathematics Using Lean and Controlled Natural Language

This includes the whole parsing and elaboration pipelines. For instance Verbose Lean overwrites the notation for implication to use the double arrow symbol that is normally used in mathematics instead of a single arrow. The examples in this paper also use an infix notation for continuity and limits as in `u tendsto x`.

Overriding notation is not only about having a nice output. It also help mitigating unwanted side-effects of using type theory. For instance say we want to use the sequence of real numbers $n \mapsto 1/(n + 1)$. Using its default configuration, Mathlib may need help to understand that $1/(n + 1)$ is a real number and not a natural number. The correct interpretation can be enforced using a type ascription such as $1/(n + 1 : \mathbb{R})$. But this is distracting for students in the provided code, and failing to use such ascription in their own code can lead to inscrutable error messages. In this case it is much easier to override the meaning of the division symbol to always mean division or real numbers. One can also use a custom notation for function abstraction specialized to sequences of real numbers. For instance one can ensure `seq n ↦ ...` gets expanded to `fun n : ℕ ↦ (... : ℝ)`.

Note there is no way to completely avoid type ambiguities in the input without type ascriptions. We have seen students feeling the need to state as an intermediate fact something like $0 < 1$. Here there is no way Lean can guess whether this is meant as an inequality of real numbers or of natural numbers. And there is no way students can be aware of this issue without discussing the subtle status of the “inclusion” of \mathbb{N} into \mathbb{R} . Lean will interpret the above statement as an inequality of natural numbers and our tactics will happily prove it. But then using this intermediate fact will fail if the intended meaning was an inequality of real numbers. Note that Lean has an option, namely `pp.numeralTypes`, to always decorate literal numbers such as 0 or 1 when it displays them. This helps making the above problem easier to spot, but it does not fix the input issue and does not avoid discussing the subtlety.

4.2 Translating to a new language

The English language can be a huge barrier for undergraduate students. One can also imagine teachers who want to use a dialect of English. Verbose Lean comes with an English version and a French version. Adding a new language can be done by imitation without any knowledge of Lean programming. The process is to copy the English folder of the Verbose library and replace English words. In case of doubt, comparing the French and English versions can show the required modifications. For instance we see⁴ in the English version:

```
declare_syntax_cat maybeApplied
syntax term : maybeApplied
syntax term "applied to " term : maybeApplied
syntax term "applied to " term " using " term : maybeApplied
syntax term "applied to " term " using that " term : maybeApplied

def maybeAppliedToTerm : TSyntax `maybeApplied → MetaM (TSyntax `term)
| `(maybeApplied| $e:term) => pure e
| `(maybeApplied| $e:term applied to $x:term) => `($e $x)
| `(maybeApplied| $e:term applied to $x:term using $y) => `($e $x $y)
| `(maybeApplied| $e:term applied to $x:term using that $y) =>
  `($e $x (strongAssumption% $y))
| _ => pure default

elab "We" " conclude by " e:maybeApplied : tactic => do
  concludeTac (← maybeAppliedToTerm e)
```

⁴ The actual code has some more cases that were removed here for conciseness.

We will comment more on this code in the next section. Our point here is that we see a lot of mysterious things but understanding them is not required to write the French version:

```
declare_syntax_cat maybeAppliedFR
syntax term : maybeAppliedFR
syntax term "appliqué à " term : maybeAppliedFR
syntax term "appliqué à " term " en utilisant " term : maybeAppliedFR
syntax term "appliqué à " term " en utilisant que " term : maybeAppliedFR

def maybeAppliedFRToTerm : TSyntax `maybeAppliedFR → MetaM Term
| `(maybeAppliedFR| $e:term) => pure e
| `(maybeAppliedFR| $e:term appliqué à $x:term) => `($e $x)
| `(maybeAppliedFR| $e:term appliqué à $x:term en utilisant $y) => `($e $x $
  y)
| `(maybeAppliedFR| $e:term appliqué à $x:term en utilisant que $y) =>
  `($e $x (strongAssumption% $y))
| _ => pure default

elab "On" " conclut par " e:maybeAppliedFR : tactic => do
  concludeTac (← maybeAppliedFRToTerm e)
```

5 Some implementation mechanisms

The previous sections have been all about the pedagogical choices of the library, about how they can be tweaked by teachers and how students can use them. We now switch gears and turn to the question of the Lean mechanisms that allow all this. Many of those mechanisms are specific to Lean 4, the new family of versions of Lean that was officially released for the first time in September 2023 and puts flexibility of use in the center [5].

The flexibility of Lean as a proof assistant rests on two main pillars. The first one is that Lean is also a programming language and that almost all of Lean is implemented in Lean. This allows in particular to override a lot of what Lean is doing, even pretty deep down, but we don't really use that directly. However we certainly use Lean as a programming language here, so we need a very quick introduction to that.

Lean is a pure functional programming language. So, fundamentally, it never does anything but defining and applying functions. However Lean makes extensive use of the monad pattern together with an extremely sophisticated notation system that can make it look a lot like imperative programming, but without the bad surprises [15]. An important inspiration is Haskell here. For our purposes, one can think of a monad M as a programming environment with a well specified state that can be read or written during program execution, a well specified way it can fail or not, and a well specified way of interacting or not with the outside world (such interactions could include reading files or printing things for instance). For any type α , the type $M \alpha$ is the type of programs that can do all this and return an element of type α (when they don't throw an exception if M includes the exception throwing capability). Running such a program requires providing an initial state and actually also return the new state if the state includes writable parts.

An important example is the `CoreM` monad defined by Lean itself. It allows to describe and run programs that have read and write access to all definitions in scope, read only access to options, can fail by throwing certain kinds of exceptions, and can interact with the outside world (of course it has a more precise definition, we only give the flavor here).

27:12 Teaching Mathematics Using Lean and Controlled Natural Language

On top of this `CoreM` monad sits the `MetaM` monad which mainly adds read and write access to the meta-variable context. A meta-variable is a place-holder that can be used in particular in a partially constructed proof. For instance at the very beginning of an interactive proof of a lemma, the full proof is a single meta-variable. By itself a meta-variable only stores a unique identifier. The meta-variable context is a data structure holding in particular for each meta-variable its expected type (that would be the conclusion of the lemma in our example) and its local context (that would be the assumptions of the lemma). On top of `MetaM` sits the `TacticM` monad which describes tactics, with additional access to all the relevant goals.

The second pillar of flexibility is the existence of typed concrete syntax objects as first class citizens [14, 13]. Here an important source of inspiration is the family of LISP languages, especially modern incarnations such as Racket. This is crucial for us. First it is crucial to our translation system that the syntax of tactics is clearly separated from their implementation. It also allows the assisted modes to provide suggestions that are guaranteed to be syntactically correct, because they produce syntax objects that are then printed as strings. This is seen in the snippets above that use syntax quotations such as `(maybeApplied| $e:term)`. Syntax objects can also be turned into other syntax objects, either by macros such as the one showed in Section 4 or by functions in the library.

We will now explain part of the implementation of: `By hu applied to δ using that $\delta > 0$ we get N such that $Hu : \forall n \geq N, |u n - x_0| \leq \delta$` that we saw earlier. The place where the corresponding syntax is hooked to the tactic implementation is

```
elab "By " e:maybeApplied " we get " news:newStuff : tactic =>
do obtainTac (← maybeAppliedToTerm e) (newStuffToArray news)
```

The `elab` command is a shortcut that allows to define a syntax in the first line and immediately assign it some meaning in the second line. On the first line we see two literal strings and two variables `e` and `news`. Those variables hold syntax objects with some syntax categories that are defined in Verbose Lean. They are based in the crucial syntax category `term` which is used for syntax representing Lean expressions. The (simplified) definition of `maybeApplied` representing functions that may be applied to arguments, and then a function turning such syntax objects into terms syntax objects have been seen in the first code snippet in Section 4.2. The first line of that snippet registers our syntax category and the next four lines describe four ways to build a syntax object in this syntax category. Those four ways are very simple and combine terms and literal words.

Now we can move to the second line of our `elab` command that calls the actual tactic. The `do` keyword starts a monadic program. Here it is a `TacticM Unit` program, i.e. a program in the `TacticM` monad that returns nothing – its only purpose is to manipulate the state of this monad. The type of the three involved functions are

```
obtainTac : TSyntax `term → Array MaybeTypedIdent → TacticM Unit
maybeAppliedToTerm : TSyntax `maybeApplied → MetaM (TSyntax `term)
newStuffToArray : TSyntax `newStuff → Array MaybeTypedIdent
```

We will ignore the third function. Note that the return type of the second one does not match the type of the first input to the first function. We need a term and not only a program computing a term in the `MetaM` monad. Fortunately, a program in this monad can be used in the `TacticM` monad which extends it. And the arrow in `(← maybeAppliedToTerm e)` tells Lean to run the `maybeAppliedToTerm e` program and feed the result at this spot.

The definition of the `maybeAppliedToTerm` function was shown in Section 4.2. It is of course defined by pattern matching on the four possibilities to create a `maybeApplied` syntax object (plus a wild card possibility that is required because the syntax category

could in principle be extended after the definition of this function). Note that the last interesting case also uses the `strongAssumption\linline{(by strongAssumption : $y)}` where `strongAssumption` is one of our tactics. Of course we could have used the expanded version here but the macro is used in several other places. Hence this example is a library function turning some syntax objects into other syntax objects using pattern matching and a macro that also does such a transformation.

The result type is not directly `TSyntax `term` but a program in the `MetaM` monad. This is because the quotation mechanism includes hygiene guarantees, a mechanism preventing accidental name capture and requiring some information from the surrounding context.

Note that one could inline this function into the `obtainTac` function. But this would make the latter into a language dependent function. As we saw in Section 4.2, Verbose Lean contains a French `maybeAppliedFR` syntax category and a function to turn syntax objects in this category into terms. It then uses the exact same `obtainTac` function from the language agnostic part of the library. Hopefully this already illustrate how we put to good use the monadic meta-programming framework of Lean and, crucially, its treatment of syntax objects. Syntax objects are also used to implement all the little DSLs we saw earlier.

We now want to discuss part of the implementation of `obtainTac`. Its first task is to turn the term it got as its first argument into a Lean expression, i.e. an abstract syntax object whose type `Expr` is an inductive type whose main constructors correspond to the fundamental operations of lambda calculus: function application, function abstraction... This process is possible in the `TacticM` monad which has access to all definitions in scope as well as to the local context of the proof where this tactic is used.

Then `obtainTac` first tries to decompose the type of this expression. In our example this type is $\exists N, \forall n \geq N, |u_n - x_0| \leq \delta$ which can indeed be decomposed as a witness `N` and its property. This job of `obtainTac` is a trivial wrapper around a standard Lean tactic.

More interestingly, when such a decomposition does not make sense, the tactic will try to apply an anonymous splitting lemma. We saw already how to configure the lemmas that are tried. Now we want to discuss how this configuration is stored and updated. We saw that programs in the `CoreM` monad have access to existing declarations. More generally they have access to the so-called environment that stores declarations but also a lot more information. Lean allows to declare environment extensions storing user information for later use. In the case at hand, the stored information is simply a list of lists of declaration names. But our library also uses more interesting extensions for assisted modes.

The multilingual support for help and suggestions is based on a multilingual function dispatch framework by Mario Carneiro. Multilingual functions are first registered using the `register_endpoint` command. This gives a function that can immediately be used to define other functions. But running those functions requires implementing the endpoint in the current language, which is `en` by default but can be changed using `setLang`. For instance:

```
/-- Multilingual hello function. -/
register_endpoint hello : CoreM String

/-- Greeting function refering to our endpoint before any implementation
is defined. -/
def greet (name : String) : CoreM String :=
  return (← hello) ++ " " ++ name

#eval greet "Patrick" -- throws error: no implementation of hello found
for language en
```

27:14 Teaching Mathematics Using Lean and Controlled Natural Language

```
implement_endpoint (lang := en) hello : CoreM String := return "Hello"
implement_endpoint (lang := fr) hello : CoreM String := return "Bonjour"
#eval greet "Patrick" -- returns "Hello Patrick"
setLang fr
#eval greet "Patrick" -- returns "Bonjour Patrick"
```

Note that above example creates three declarations: `hello`, `hello.en` and `hello.fr`, but only the first one is explicitly used. The implementations in this example are silly since they do not perform anything inside the `CoreM` monad, they simply return a value without reading or writing any `CoreM` state. But the `hello` function itself, which is created by the `register_endpoint` command, crucially uses `CoreM` to fetch the relevant information from the environment after reading the current language setting.

The way this is achieved illustrates an important point about Lean's flexibility. Lean as a proof assistant has very strong soundness guarantees, and the whole proof checking is handled by its type system. This translates to the default behavior of Lean as a programming language. We saw that being a pure functional programming language does not prevent us from accessing state and having side-effects. One simply has to be honest about it by announcing in which monad we are working. Lean also allows to throw away type safety as long as we clearly announce it. And of course functions which do that cannot appear in proofs (they can participate in creating proofs, but can't appear in the end result).

A very simple version of the trick used in the multilingual framework is implemented in the example below. The `runFunctionOn` takes a string and a natural number, searches the environment for a declaration whose name is that string, then forcefully tells the Lean type system that this declaration is a function from natural numbers to natural numbers and applies it to the given number. The result is in `CoreM ℕ` rather than `ℕ` since it needs access to the environment to search for the relevant declaration and it could fail to find it so it needs to be able to throw errors – this is what happens in the second example below. But the new piece is the `unsafe` signpost in front of `def`. Indeed the declaration could be found but not with type `ℕ → ℕ`, bringing us into undefined behavior territory. This is what happens in the third example below where a function concatenating strings is found.

```
unsafe def runFunctionOn (function : String) (a : ℕ) : CoreM ℕ := do
  let myFun ← evalConst (ℕ → ℕ) (Name.mkSimple function)
  return myFun a

def foo (a : ℕ) := 2*a + 1
#eval runFunctionOn "foo" 1 -- returns 3
#eval runFunctionOn "baz" 1 -- fails with error message: unknown
  declaration baz

def bar (a : String) := a ++ a
#eval runFunctionOn "bar" 1 -- crashes Lean
```

The moral is that programming in Lean allows to do very unsafe things that completely bypass the guarantees offered by the type system, but this must be clearly flagged and cannot be used as a proof (explaining how soundness is protected is beyond the scope of this

discussion). Note that our actual multilingual dispatch is much more careful and checks that types match before calling functions so that teachers don't crash Lean when they make a type mistake in the implementation of a new help message. For performance reasons we don't want to perform this check at every function call, so we also use an extension that keeps track of a list of endpoints and type-checked implementations.

Although we insisted on our use of concrete syntax objects, we also use abstract ones, with type `Expr`, in the tactic backends. We even have a custom version `VExpr` with many more constructors that are useful when analysing goals and assumptions in assisted modes. For instance bounded quantifiers have dedicated constructors. We have a function parsing an `Expr` into a `VExpr`, hence factoring out work that many help functions would need to do. The type of help functions that analyse the goal is `MVarId → VExpr → SuggestionM Unit` where `MVarId` is used to indicate the relevant goal and the `SuggestionM` monad accumulates suggestions while providing access to the `MetaM` monad. Such functions are registered as part of the configuration by teachers, together with a pattern indicating (coarsely) which kind of goal they comment on. Calling the help tactic uses a discrimination tree to quickly locate functions with the relevant pattern and then check whether they are active in the configuration. This use of discrimination trees is not necessary until someone implements thousands of help functions, but the infrastructure is provided by Lean so it is free.

The last piece of Lean infrastructure that we want to comment on is the framework that allows us to build the suggestion widget. There are quite a few layers here. Lean implements the Language Server Protocol (LSP) with many extensions related to the so called info view which gather the tactic state display, various messages and user-defined widgets. Deep down, widgets are Javascript modules that export a React component that is displayed by the VSCode extension, can access information from Lean and modify the current document. However the `ProofWidgets` library [9] offers a powerful abstraction that allows us to ignore Javascript. In particular it features a JSX-like DSL as well as React components written in Lean and having a Lean interface. As a result, `Verbose Lean` does not contain a single line of actual Javascript or HTML. For instance the loop printing the suggestions is:

```
for ⟨linkText, newCode, range?⟩ in suggs do
  let p : MakeEditLinkProps := .ofReplaceRange doc.meta
    ⟨params.pos, params.pos⟩ (ppAndIndentNewLine curIndent newCode) range?
  children := children.push
    <li style={json% {"margin-bottom": "1rem"}}>
      <MakeEditLink
        edit={p.edit} newSelection?={p.newSelection?} title?={p.title?}>
        { .text linkText }
      </MakeEditLink>
    </li>
```

The `li` tag is directly turned into an HTML list item, whereas `MakeEditLink` refers to a `ProofWidgets` component. This component is rendered as an HTML link which, upon getting clicked, edits the proof script. All this is fully type-correct Lean code, with real-time typechecking and the expected editor support (for instance ctrl-clicking on `MakeEditLink` jumps to the relevant declaration).

The tactic state natively allows to select names or sub-expressions in the local context or the goal. It records this information as an array where each element inhabits an inductive type `Lean.SubExpr.GoalLocation` having a constructor for each kind of selection, for instance a constructor for an element of the local context, one for a sub-expression in the type of such an element, etc. This layout is not convenient for our purposes so we introduce another

datatype `SelectionInfo` which gather the same information by type of selections. We also have many functions querying this information. Each suggestion provider has type `SelectionInfo → MVarId → WidgetM Unit` analogous to the `help` function type.

6 Related work

Both the dream of using proof assistants for teaching and the work on alternative interfaces are very common. However most teaching uses focus on computer science, logic or discrete mathematics, or even on proof assistants for themselves.

One notable exception is the work of Heather Macbeth at Fordham university [8]. However her course is more focused on computations and less on reasoning, so the need for a controlled natural language is less pressing. What is common to both contexts is the need for automation that is adapted to the level of details expected from students. And indeed some of our tactics rely on tactics developed for Macbeth's course.

Even more relevant is the comparison with the Coq-Waterproof project [16] that was developed independently and shares many goals with Verbose Lean. Discussing with its authors led to several improvements in our work. One thing that we still do not have is a nice custom text editor to mix rendered comments and interactive exercises. On the other hand, we do benefit from using a very flexible proof assistant that easily allows syntactic freedom and interactive interfaces, as explained in this paper. The resulting proof scripts are closer to paper proofs and the user interaction model is richer.

Also very relevant and interesting is the Diproche system [3, 4]. Its focus on controlled natural language is even greater than in Verbose Lean. Proofs are sequences of assertions in a more flexible language. Those assertions are sent to an automated prover that complains if it cannot justify a step. The main downside is that the proof structure is much less clear.

On the topic of alternative interfaces, there are again many attempts that seem practicable only for pure logic. For instance this is the case of the Actema project [6] which proposes a drag and drop interface that is partly a more graphical version of our suggestion widget and can interface with Coq. However it seems difficult to integrate with computations as in our squeeze theorem example (which was chosen as the simplest example involving computations). And it very explicitly targets leaving no written trace at all, hence has a very different goal.

Also in the same category but explicitly targeting teaching very young university students is the $d\forall\exists$ duction project [11]. It features a graphical interface based on selecting sub-expressions and clicking buttons, with a completely invisible Lean backend. Again there is no written trace so the transfer of skills to paper is not completely clear.

Edukera [10] is another point and click interface that produces a written text. It is a web interface based on Coq. Its first main drawback is that teachers cannot write exercises or configure anything, they simply have access to a fixed set of exercises. In addition, there is no possibility to directly input text. The interface is only based on clicks and the text is purely on the output side. Also the development of Edukera stopped in 2018.

As far as we know, none of those very nice projects have multilingual support except for Edukera. Most of them are only in English, Diproche is only in German and $d\forall\exists$ duction is only in French.

7 Conclusion and future work

We end this paper with remarks about the effects of this work on students, on colleagues and on other proof assistant users, and then with remarks about future work.

Although some version of this library has been used for five years in University Paris-Saclay at Orsay, it is still a work in progress, especially since the switch to Lean 4 made it a lot more flexible. The suggestion widget in particular has not been used with students yet, and will need refinements and extensions. However the Lean 3 version, including a help tactic but no widget, has been used a lot. The move from standard Lean tactics to controlled natural language tactics seemed pretty risky to us, and was tried only because of our frustration with the difficulty to transfer skills from the computer to paper. But it has been a lot more effective than what we anticipated, and really seemed to help with the pedagogical objectives we described in Section 2. One limitation of those experiments, besides their anecdotal nature, is that we did not try to use this tool with really weak students.

Compared to other reports about the use of proof assistants in mathematics teaching, it is also worth mentioning that we met almost no resistance from colleagues or students. The only exceptions came in the early years of this experiment when we used some pure propositional logic exercises. Our students simply could not see the point of making efforts to prove tautologies. As a result, they mostly did not try and, more importantly, they lost faith in the usefulness of rigorous logical reasoning. After we removed those exercises, the problem was solved. Of course such exercises can be pertinent in other contexts.

One can wonder whether such a tactic library could be used for regular Lean input, say in Mathlib, the mathematical library of Lean. We do not believe such a use would be productive. The usual tactics of Lean are more concise and flexible, and learning the fragment corresponding to what we can do with Verbose Lean is not the most time consuming task for new users. The main difficulties rather come from switching to a formal mindset, navigating the library to find relevant definitions and lemmas, and finding the most efficient encodings of mathematical definitions and statements in Lean's type theory. The tactic language described here do not really make these things easier (and was not at all built for this purpose). More generally, past experiences with programming languages, going at least as far back as COBOL, suggest that controlled natural languages are not sufficiently efficient as a general input format. So it seems unlikely that this tactic language would significantly facilitate access to formalized mathematics for mathematicians. One could still argue that it could make formalized mathematics more accessible to readers that do not want to learn the language but still want to access precise definitions, statements and proofs. But we think that this goal is much more likely to be achieved by translating formalized mathematics to natural language a posteriori. In particular such a translation can give access to information that was automatically inferred and to proofs that were automatically generated.

Concerning future work, there are plans to rigorously assess the benefits of using this library next year with the APPAM⁵ team which includes specialists in education sciences. Error reporting is also a never-ending work in progress. Each new interface or piece of automation requires more care in case of incorrect input, and students always find new ways to trigger unexpected error messages. On the multi-lingual side, one short-term goal is to make it easier to create variants of an existing language. Another project is to offer more exercises that are ready to use or modify for teachers. Existing exercises are not yet all ported to Lean 4, and new ones should be created in different fields of elementary mathematics.

References

- 1 Jeremy Avigad. *Learning Logic and Proof with an Interactive Theorem Prover*, pages 277–290. Springer International Publishing, Cham, 2019. doi:10.1007/978-3-030-28483-1_13.

⁵ <https://appam.icube.unistra.fr/>

- 2 Evmorfia-Iro Bartzia, Emmanuel Beffara, Antoine Meyer, and Julien Narboux. Underlying theories of proof assistants and potential impact on the teaching and learning of proof. In *12th International Workshop on Theorem proving components for Educational software*, Rome, Italy, July 2023. Julien Narboux and Walther Neuper and Pedro Quaresma. URL: <https://hal.science/hal-04227823>.
- 3 Merlin Carl. Number theory and axiomatic geometry in the diproche system. In Pedro Quaresma, Walther Neuper, and João Marcos, editors, *Proceedings 9th International Workshop on Theorem Proving Components for Educational Software, ThEdu@IJCAR 2020, Paris, France, 29th June 2020*, volume 328 of *EPTCS*, pages 56–78, 2020. doi:10.4204/EPTCS.328.4.
- 4 Merlin Carl, Hinrich Lorenzen, and Michael Schmitz. Natural language proof checking in introduction to proof classes - first experiences with diproche. In João Marcos, Walther Neuper, and Pedro Quaresma, editors, *Proceedings 10th International Workshop on Theorem Proving Components for Educational Software, ThEdu@CADE 2021, (Remote) Carnegie Mellon University, Pittsburgh, PA, United States, 11 July 2021*, volume 354 of *EPTCS*, pages 59–70, 2021. doi:10.4204/EPTCS.354.5.
- 5 Leonardo de Moura and Sebastian Ullrich. The Lean 4 theorem prover and programming language. In André Platzer and Geoff Sutcliffe, editors, *Automated Deduction - CADE 28 - 28th International Conference on Automated Deduction, Virtual Event, July 12-15, 2021, Proceedings*, volume 12699 of *Lecture Notes in Computer Science*, pages 625–635. Springer, 2021. doi:10.1007/978-3-030-79876-5_37.
- 6 Pablo Donato, Pierre-Yves Strub, and Benjamin Werner. A drag-and-drop proof tactic. In Andrei Popescu and Steve Zdancewic, editors, *CPP '22: 11th ACM SIGPLAN International Conference on Certified Programs and Proofs, Philadelphia, PA, USA, January 17 - 18, 2022*, pages 197–209. ACM, 2022. doi:10.1145/3497775.3503692.
- 7 Marie Kerjean, Frédéric Le Roux, Patrick Massot, Micaela Mayero, Zoé Mesnil, Simon Modeste, Julien Narboux, and Pierre Rousselin. Utilisation des assistants de preuves pour l'enseignement en L1. In *Gazette de la SMF*, volume 174, Octobre 2022.
- 8 Heather Macbeth. The mechanics of proof. <https://hrmacbeth.github.io/math2001/>.
- 9 Wojciech Nawrocki, Edward W. Ayers, and Gabriel Ebner. An extensible user interface for Lean 4. In Adam Naumowicz and René Thiemann, editors, *14th International Conference on Interactive Theorem Proving, ITP 2023, July 31 to August 4, 2023, Białystok, Poland*, volume 268 of *LIPICs*, pages 24:1–24:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPICs.ITP.2023.24.
- 10 Benoît Rognier. Edukera. <https://edukera.com/>.
- 11 Frédéric Le Roux. D \forall Eduction. <https://perso.imj-prg.fr/frederic-leroux/dforallEduction/>.
- 12 Athina Thoma and Paola Iannone. Learning about proof with the theorem prover Lean: the abundant numbers task. *International Journal of Research in Undergraduate Mathematics Education*, 8(1):64–93, April 2022. doi:10.1007/s40753-021-00140-1.
- 13 Sebastian Ullrich. *An Extensible Theorem Proving Frontend*. PhD thesis, Karlsruhe Institute of Technology, Germany, 2023. URL: <https://nbn-resolving.org/urn:nbn:de:101:1-2023080204582480933072>, doi:10.5445/IR/1000161074.
- 14 Sebastian Ullrich and Leonardo de Moura. Beyond notations: Hygienic macro expansion for theorem proving languages. *Log. Methods Comput. Sci.*, 18(2), 2022. doi:10.46298/LMCS-18(2:1)2022.
- 15 Sebastian Ullrich and Leonardo de Moura. 'do' unchained: embracing local imperativity in a purely functional language (functional pearl). *Proc. ACM Program. Lang.*, 6(ICFP):512–539, 2022. doi:10.1145/3547640.
- 16 Jelle Wemmenhove, Thijs Beurskens, Sean McCarren, Jan Moraal, David Tuin, and Jim Portegies. Waterproof: educational software for learning how to write mathematical proofs, 2022. arXiv:arXiv:2211.13513.

- 17 Xiaoheng Yan and Gila Hanna. Using the Lean interactive theorem prover in undergraduate mathematics. *International Journal of Mathematical Education in Science and Technology*, 0(0):1–15, 2023. doi:10.1080/0020739X.2023.2227191.

Lean Formalization of Completeness Proof for Coalition Logic with Common Knowledge

Kai Obendrauf  

Department of Computer Science, Vrije Universiteit Amsterdam, The Netherlands

Anne Baanen   

Department of Computer Science, Vrije Universiteit Amsterdam, The Netherlands

Patrick Koopmann   

Department of Computer Science, Vrije Universiteit Amsterdam, The Netherlands

Vera Stebletsova  

Department of Computer Science, Vrije Universiteit Amsterdam, The Netherlands

Abstract

Coalition Logic (CL) is a well-known formalism for reasoning about the strategic abilities of groups of agents in multi-agent systems. Coalition Logic with Common Knowledge (CLC) extends CL with operators from epistemic logics, and thus with the ability to model the individual and common knowledge of agents. We have formalized the syntax and semantics of both logics in the interactive theorem prover Lean 4, and used it to prove soundness and completeness of its axiomatization. Our formalization uses the type class system to generalize over different aspects of CLC, thus allowing us to reuse some of to prove properties in related logics such as CL and CLK (CL with individual knowledge).

2012 ACM Subject Classification Security and privacy → Logic and verification; Mathematics of computing → Mathematical software; Computer systems organization → Dependable and fault-tolerant systems and networks

Keywords and phrases Multi-agent systems, Coalition Logic, Epistemic Logic, common knowledge, completeness, formal methods, Lean prover

Digital Object Identifier 10.4230/LIPIcs.ITP.2024.28

Related Version *Extended Version*: <https://zenodo.org/records/12582709>

Supplementary Material

Software (Source Code): <https://github.com/kaiobendrauf/cl-lean> [20]
archived at [swh:1:dir:5679444e6dc3b26cd7f1c54786bff9be89541c19](https://sw.haskell.org/dir/5679444e6dc3b26cd7f1c54786bff9be89541c19)

Funding *Anne Baanen*: NWO Vidi grant No. 016.Vidi.189.037, Lean Forward.

1 Introduction

Computers rarely work in isolation, rather they constantly interact with both human users and other devices. Such interconnected systems can range from household Internet of Things (IoT) devices, working towards creating a useful digital home for a user [2], to safety-critical systems for metros that need to account for multiple trains [15]. Correctly designing and verifying such systems is an important goal of research in Artificial Intelligence, specifically in the field of Multi Agent Systems (MAS) [11, 13, 27]. The large number of agents and simultaneous goals involved in these interactions make them highly complex. Furthermore, computers in such systems must often operate with imperfect information [26], for instance because they have limited input about the external environment [13]. It can therefore be difficult to maintain an overview of whether a system has been correctly programmed to always meet its requirements, highlighting the need for formal modelling and programmatic verification [27].



© Kai Obendrauf, Anne Baanen, Patrick Koopmann, and Vera Stebletsova;
licensed under Creative Commons License CC-BY 4.0

15th International Conference on Interactive Theorem Proving (ITP 2024).

Editors: Yves Bertot, Temur Kutsia, and Michael Norrish; Article No. 28; pp. 28:1–28:18

Leibniz International Proceedings in Informatics



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

In this paper we focus on Coalition Logic with Common Knowledge (CLC) to model such systems and their requirements. Coalition Logic (CL) was introduced by Marc Pauly in 2002 [22] for reasoning about abilities of agents, and is a popular logic in MAS research [1]. CL introduces an effectivity operator, which describes whether some group of agents is effective for ensuring some outcome, regardless of the actions of other agents. CL was later extended by Ågotnes and Alechina [1] into CLC by adding operators from epistemic logic for individual and common knowledge.

The current paper aims to build a foundation for CLC formalizations for MAS by investigating how CLC can be defined and reasoned over using the Lean prover [8]. Lean is an interactive proof assistant based on dependent type theory, and its mathematical library Mathlib [25] is a rapidly growing community-driven project that we made thankful use of. In this work, we use Lean to formalize the syntax and semantics of CLC and formalize the soundness and completeness theorem together with the finite model property of CLC as given by Ågotnes and Alechina [1]. Formalizing these proofs allows us to check that the syntax and semantics of CLC defined in Lean relate to one another as expected. Additionally, doing so demonstrates that these definitions can be used in nontrivial proofs about CLC.

Since we closely follow the Ågotnes and Alechina proof in our formalization, we will focus on the larger scale proof engineering aspects and show only relevant excerpts of these proofs. A full, **sorry-free**, version of our code is available online at <https://github.com/kaiobendrauf/c1-lean>. The formalization of CLC is part of a larger work [21], where the entire proofs and their formalization are given in as much detail as possible. This larger work [21] also formalizes soundness and completeness of CL. Although the current paper focuses on CLC, we give special attention to lemmas and definitions that are also used in the completeness proof for CL. Thus we illustrate the ways in which we prioritize generalizability and reusability in our Lean implementation. The intention of this design choice is to make our formalization easier for future work to extend.

In the following sections we give a brief overview of existing work formalizing logics related to CLC in Section 2 and an overview of the Lean prover and its mathematical library in Section 3. We give a detailed description of the syntax, semantics and axiomatic system of CLC in Section 4. We describe our definition of CLC in Lean in Section 5. Our formalization of the soundness of CLC is described in some detail in Section 6. Section 7 notes a method of making our formalization reusable for other logics. Finally, using these definitions we will prove in Lean that CLC is complete in Section 8.

2 Related work

The formalization of modal logics in proof assistants is an active area of research. To our knowledge, CLC has not yet been formalized in any proof assistant, however our work builds on related work on formalizing Epistemic Logics (EL) and CL. We start by describing work on CL. Nalon et al. [18] present a prototype automated reasoning tool for CL, based on a sound, complete, and terminating resolution-based calculus for CL in SWI-Prolog. On the other hand, Baston and Capretta [5] propose how to formalize the relation between strategic games and the effectivity operator in CL. These works provide support that CL can be defined and reasoned with in proof assistants. However, to the best of our knowledge, at this point in time there are no current works that formalize completeness of CL in any proof assistant, nor has any project formalized CL in Lean.

In contrast there are several existing formalizations of EL both in Lean [6, 17, 19] and other proof assistants [7, 9, 16]. In Lean, the first of these is the completeness proof of EL (S5) by Bentzen [6]. Following this, both Neeley [19] and Li [17] formalized completeness

again, but with different approaches, showing how flexibly such proofs can be implemented in Lean. We will, when possible, defer to these existing works on formalizing EL in Lean for guidance on implementation choices. Most often, we use ideas by Neeley [19] as she uses the same type of proof as Ågotnes and Alechina [1] while being particularly detailed about her design decisions. Furthermore, this work formalizes several logics, thus we know the design decisions generalize to multiple types of logic.

3 The Lean prover and Mathlib

We used the Lean theorem prover [8] in our formalization. Lean is an interactive theorem prover based on the Calculus of Inductive Constructions, featuring proof irrelevance, quotient types and classical reasoning. These features are used ubiquitously throughout the flagship mathematical library for Lean, Mathlib [25], which we used as a starting point for our own formalization. An introduction to Lean can be found at [3].

A characteristic aspect of Mathlib is its use of *typeclasses* to organize mathematical theories. Lean’s typeclass system extends the class mechanism introduced for operator overloading in Haskell [28], and are used to associate types with both operators and axioms about these operators. Moreover, the typeclass system permits extending structures, so that, for example, any theorem declared for a `Monoid M` will automatically apply to a type `G` for which a `Group G` instance exists. The typeclass system is invoked by placing parameters to declarations between square brackets. An *instance synthesis* algorithm is used to supply values for these parameters automatically, through a variation on depth-first search [23].

In 2023, Mathlib was ported from Lean 3 to the newly released Lean 4, a port that required substantial changes in notation and design choices. Our project was originally written for Lean 3 and after the proofs were completed, we ported it to Lean 4. The code we present in our paper is an abridged version of the Lean 4-compatible source code, using Lean version 4.4.0-rc1 and Mathlib commit 98fe17fd. Although this paper omits many proof steps for presentational purposes, in our accompanying formalization all proofs are complete and `sorry`-free.

4 Coalition Logic with Common Knowledge

We recall the syntax and semantics of CLC, as well as the axiomatization, following Ågotnes and Alechina [1], in their work extending CL [22].

Based on a finite, non-empty set N of *agents*, and a set Φ_0 of *atomic propositions*, CLC formulas are constructed using the usual propositional logic operators, the CL *effectivity* operator $[G]$, where $G \subseteq N$, and two epistemic operators: K_i for individual knowledge, where $i \in N$, and C_G for common knowledge. Formally, CLC formulas are defined by the following BNF grammar:

$$\varphi := \perp \mid p \mid \varphi \wedge \varphi \mid \varphi \rightarrow \varphi \mid [G]\varphi \mid K_i\varphi \mid C_G\varphi$$

where $p \in \Phi_0$, $G \subseteq N$ and $i \in N$. We note that our syntax here is slightly different from that of Ågotnes and Alechina [1], as we allow the case when $G = \emptyset$ for the $C_G\varphi$ operator. Additionally, based on Neeley [19], we use a non-minimal set of propositional operators as this simplifies our proofs in Lean.

$[G]\varphi$ expresses the *effectivity* of a coalition to achieve φ . Intuitively, $[G]\varphi$ can be read as “coalition group G can ensure φ , regardless of the actions of agents not in the coalition”. K_i expresses the knowledge of an agent $i \in N$. Thus, $K_i\varphi$ can be intuitively read as “agent i

knows φ ". This individual knowledge can be extended to groups via the derived operator E_G , using the conjunction of individual knowledge. Specifically, for $G \subseteq N$, the notation $E_G\varphi$ is defined as $E_G\varphi := \bigwedge_{i \in G} K_i\varphi$ and reads as "everyone in group G knows φ ". $C_G\varphi$ expresses that group G has common knowledge of φ . Intuitively this can be read as "everyone in group G knows φ , and they all know that they all know φ , and they all know that they all know that they all know φ and so on".

The semantics of CLC is based on *epistemic coalition* frames and models. An epistemic coalition model contains an *epistemic accessibility relation* \sim_i for each agent $i \in N$. These are equivalence relations that model what each agent knows. Specifically, if $(s, t) \in \sim_i$ for some agent i , written as $s \sim_i t$, then agent i cannot differentiate state s and state t .

Additionally epistemic coalition frames and models contain an *effectivity structure* E which represents the effectivity of coalitions. Given a non-empty set S of states, E maps a state and subset of N to a set of subsets of S , i.e. $E : S \rightarrow \mathcal{P}(N) \rightarrow \mathcal{P}(\mathcal{P}(S))$. Note that, given some state $s \in S$ and set of agents $G \subseteq N$, $E(s)(G)$ denotes a set of sets of states. Intuitively, if $X \in E(s)(G)$, the coalition G must have some joint strategy in state s such that, no matter the strategy of agents not in the coalition, we are guaranteed to end up in some $t \in X$. In this way the effectivity structure models the ability of coalitions to ensure some (sets of) outcomes while abstracting away specific actions and strategies. In order adequately model a coalition's effectivity we require specific properties to hold, which are collectively defined by the concept of *true playability*.

► **Definition 1** (True Playability). *A truly playable effectivity structure is an effectivity structure E such that for any state s , $E(s)$ meets the following 6 conditions [1, Section 2.1].*

1. $E(s)$ is live: for every $G \subseteq N$, $\emptyset \notin E(s)(G)$
2. $E(s)$ is safe: for every $G \subseteq N$, $S \in E(s)(G)$
3. $E(s)$ is N -maximal: for every $X \subseteq S$, if $(S \setminus X) \notin E(s)(\emptyset)$, then $X \in E(s)(N)$
4. $E(s)$ is outcome monotonic: for every $G \subseteq N$ and $X, Y \subseteq S$, if $X \in E(s)(G)$ and $X \subseteq Y$, then also $Y \in E(s)(G)$
5. $E(s)$ is superadditive: for all $C, D \subseteq N$ where $C \cap D = \emptyset$, and all $X, Y \subseteq S$, if $X \in E(s)(C)$ and $Y \in E(s)(D)$, then $X \cap Y \in E(s)(C \cup D)$
6. $E(s)(\emptyset)$ is principal: there exists an $X \in E(s)(\emptyset)$ such that for every $Y \in E(s)(\emptyset)$, we have $X \subseteq Y$.

We have now everything to define epistemic coalition frames and models formally.

► **Definition 2.** *An epistemic coalition frame is a tuple $F = (S, E, \{\sim_i : i \in N\})$, where*

- S is a non-empty set of states,
- $E : S \rightarrow (\mathcal{P}(N) \rightarrow \mathcal{P}(\mathcal{P}(S)))$ is a truly playable effectivity structure, and
- $\sim_i \subseteq S \times S$ is an equivalence relation, the epistemic accessibility relation over S for agent i .

► **Definition 3.** *An epistemic coalition model is a tuple $M = (F, V)$, where:*

- F is an epistemic coalition frame, and
- $V : \Phi_0 \rightarrow \mathcal{P}(S)$ is the usual valuation function, assigning to each $p \in \Phi_0$ some set of states $V(p) \subseteq S$.

Based on an epistemic coalition model $M = (F, V)$, where $F = (S, E, \{\sim_i : i \in N\})$, and some state $s \in S$, we can now define what it means for a CLC formula ϕ to be true in s (written as $M, s \models \phi$). Truth of $[G]\varphi$ relates to the effectivity structure: if in state s group G is effective in bringing about φ , then G must be able to restrict the possible next states to some set containing only states where φ is true. $K_i\varphi$ relates intuitively to the \sim_i relation: if

■ **Table 1** Axiomatization of CLC.

(Prop)	Prop. tautologies	(K)	$\vdash K_i(\varphi \rightarrow \psi) \rightarrow (K_i\varphi \rightarrow K_i\psi)$	(T)	$\vdash K_i\varphi \rightarrow \varphi$
(4)	$\vdash K_i\varphi \rightarrow K_iK_i\varphi$	(5)	$\vdash \neg K_i\varphi \rightarrow K_i\neg K_i\varphi$	(C)	$\vdash C_G\varphi \rightarrow E_G(\varphi \wedge C_G\varphi)$
(\perp)	$\vdash \neg[G]\perp$	(\top)	$\vdash [G]\top$	(N)	$\vdash (\neg[\emptyset]\neg\varphi) \rightarrow [N]\varphi$
(S)	$\vdash ([G]\varphi \wedge [F]\psi) \rightarrow [G \cup F](\varphi \wedge \psi)$, if $G \cap F = \emptyset$				
(MP)	$\vdash \varphi, \varphi \rightarrow \psi \Rightarrow \vdash \psi$	(RN)	$\vdash \varphi \Rightarrow \vdash K_i\varphi$	(Eq)	$\vdash \varphi \leftrightarrow \psi \Rightarrow \vdash [G]\varphi \leftrightarrow [G]\psi$
(RC)	$\vdash \psi \rightarrow E_G(\varphi \wedge \psi) \Rightarrow \vdash \psi \rightarrow C_G\varphi$				

agent i knows φ in state s , then φ must be true in all states that i cannot distinguish from s . The operator $C_G\varphi$ is a little more complex, as here we need to consider paths through epistemic relations. For readability, we write $(s, t) \in (\bigcup_{i \in G} \sim_i)^*$ as $s \approx_G t$. If group G has common knowledge of φ in state s , then φ must be true in all states t such that $s \approx_G t$. Truth in M, s now defined as follows.

$M, s \not\models \perp$	
$M, s \models p$	iff $p \in \Phi_0$ and $s \in V(p)$
$M, s \models \varphi \wedge \psi$	iff $M, s \models \varphi$ and $M, s \models \psi$
$M, s \models \varphi \rightarrow \psi$	iff $M, s \models \varphi \Rightarrow M, s \models \psi$
$M, s \models [G]\varphi$	iff $\{s \in S \mid M, s \models \varphi\} \in E(s)(G)$
$M, s \models K_i\varphi$	iff $\forall t \in S, s \sim_i t \Rightarrow M, t \models \varphi$
$M, s \models C_G\varphi$	iff $\forall t \in S, s \approx_G t \Rightarrow M, t \models \varphi$

As usual, φ is *valid* in a model ($M \models \varphi$) if it is true in every state of the model and is *globally valid* ($\models \varphi$) if it is valid in all models.

The axiomatization of CLC can be seen in Table 1.

5 Formalizing the Syntax and Semantics in Lean

To formalize the syntax of CLC in Lean, we use a deep embedding, which allows us to prove metatheoretical results about the logic such as soundness and completeness [14, 19]. Thus, in Lean, the language of CLC formulas is defined as an inductive type, meaning the smallest type closed under the operators `bot`, `var`, `and`, `imp`, `eff`, `K` and `C`.

```
inductive formCLC (agents : Type) : Type
| bot
  : formCLC agents
| var (n : Nat)
  : formCLC agents
| and (φ ψ : formCLC agents)
  : formCLC agents
| imp (φ ψ : formCLC agents)
  : formCLC agents
| eff (G : Set agents) (φ : formCLC agents)
  : formCLC agents
| K (a : agents) (φ : formCLC agents)
  : formCLC agents
| C (G : Set agents) (φ : formCLC agents)
  : formCLC agents
```

The inductive type is parameterized over an arbitrary type `agents`. At this point we do not require that only finitely many agents appear in the formula. Instead, we will apply this assumption only to those theorems whose proofs require it, guided by the automated proof checking done by Lean.

28:6 Formalizing Completeness of CLC in Lean

To define the semantics, we first define effectivity structures:

```
def effectivity_struct (agents states : Type) :=
  states → Set agents → Set (Set states)
```

To represent a playable effectivity structure, we create a 6-tuple to link the effectivity function itself to the 5 playability requirements. Thus, we need to store tuples of a certain shape, which in Lean we do using a `structure` data type:

```
structure truly_playable_effectivity_struct (agents states : Type) :=
  (E      : effectivity_struct agents states)
  (liveness : ∀ s : states, ∀ G : Set agents, ∅ ∉ E s G)
  (safety  : ∀ s : states, ∀ G : Set agents, univ ∈ E s G)
  (N_max   : ∀ s : states, ∀ X : Set states, Xc ∉ E s ∅ → X ∈ E s univ)
  (mono    : ∀ s : states, ∀ G : Set agents, ∀ X Y : Set states,
             X ⊆ Y → X ∈ E s G → Y ∈ E s G)
  (superadd : ∀ s : states, ∀ G F : Set agents, ∀ X Y : Set states,
             X ∈ E s G → Y ∈ E s F → G ∩ F = ∅ →
             X ∩ Y ∈ E s (G ∪ F))
  (principal_E_s_empty : ∀ s, ∃ X, X ∈ E s ∅ ∧ ∀ Y, Y ∈ E s ∅ → X ⊆ Y)
```

Comparing the semantics defined in Section 4, we can see a particular difference in the treatment of sets: where informally we write $X \in E(s)(N)$, Lean writes $X \in E\ s\ univ$. Since Lean is based on type theory, it distinguishes `agents : Type` from its universal set `univ : Set agents`. Apart from this distinction, the conditions translate straightforwardly.

Epistemic coalition frames and models are then defined as follows:

```
structure frameECL (agents : Type) :=
  (states      : Type)
  (hs         : Nonempty states)
  (E          : truly_playable_effectivity_struct agents states)
  (rel        : agents → states → Set states)
  (rfl       : ∀ i s, s ∈ rel i s)
  (sym       : ∀ i s t, t ∈ rel i s → s ∈ rel i t)
  (trans     : ∀ i s t u, t ∈ rel i s → u ∈ rel i t → u ∈ rel i s)
```

```
structure modelECL (agents : Type) :=
  (f : frameECL agents)
  (v : ℕ → Set f.states)
```

To encode semantic entailment, we first formalize the *common knowledge path* recursively defined predicate that we call a `C_path`.

```
inductive C_path {agents : Type} {m : modelECL agents} (G : Set agents) :
  m.f.states → m.f.states → Prop
| done (hi : i ∈ G) (hst : t ∈ m.f.rel i s) : C_path G s t
| next (hi : i ∈ G) (hsu : u ∈ m.f.rel i s) (ih : C_path G u t) :
  C_path G s t
```

Intuitively, we say given a coalition G that there is a path from state s to state t if we can give, for some $n \geq 1$, some list i_0, i_1, \dots, i_n of agents in G , as well as some list u_1, u_2, \dots, u_n of states, such that $s \sim_{i_0} u_1, u_1 \sim_{i_1} u_2, \dots, u_n \sim_{i_n} t$. $s \approx_G t$ then means that there is a

C_path from s to t , where every agent in the list of agents is also in G . From here, defining semantic entailment is straightforward, so we show only the non-propositional cases:

```
def s_entails_CLC {agents : Type} (m : modeleCL agents) (s : m.f.states) :
  formCLC agents → Prop
...
| (.[G] φ)   => {t : m.f.states | s_entails_CLC m t φ} ∈ m.f.E.E s G
| (.K i φ)   => ∀ t : m.f.states, t ∈ m.f.rel i s → s_entails_CLC m t φ
| (.C G φ)   => ∀ t : m.f.states, C_path G s t → s_entails_CLC m t φ
```

6 Formalizing Soundness

The axiomatization of CLC (Table 1) is defined as an inductive predicate, that is, as an inductively defined proposition [4]. An inductive predicate is defined as the smallest predicate closed under a set of proof steps. Thus, an inductive predicate contains all proofs constructed from a finite tree of proof steps. This mirrors how the set of formulas provable in an axiomatic system is the smallest set closed under rule applications. The translation to Lean is thus very straightforward and omitted here. Before we come to the more challenging proof of completeness of this system, we prove its soundness.

► **Theorem 4** (Soundness of CLC [1, Lemma 1]). $\forall \varphi, \vdash \varphi \Rightarrow \models \varphi$

Despite the proof itself being simple, translating it into Lean is not entirely straightforward. We prove this theorem by structural induction on the proof of $\vdash \varphi$. Most of the cases can be proven directly from the given axiom. Note that axioms (\perp), (\top), (**N**), (**M**) and (**S**) relate directly to the first five true playability requirements, and axioms (**T**), (**4**) and (**5**) relate to the fact that epistemic relations are equivalence relations.

The cases (**C**) and (**RC**) are a little more complex, as they involve the C_G operator. To show that a formula of the form $C_G\varphi$ is true, we need to reason about the common knowledge relation \approx_G . More specifically, in Lean we have to look for C_paths between states. To illustrate how this is done, we look closer at the case for Axiom (**C**). Given $M, s \models C_G\varphi$, we need to show $M, s \models E_G(\varphi \wedge C_G\varphi)$, which gives the following goal after simplifying:

```
h : M, s ⊨ C G φ
hi : i ∈ G
hts : t ∈ M.f.rel i s
⊢ M, t ⊨ φ ∧ C G φ
```

For the first half of the conjunction, we apply the hypothesis h , so it remains to prove that $C_path\ G\ s\ t$ holds. In this case the path will have length one, so that the constructor $C_path.done$ applies, and our existing hypothesis $hts : t \in M.f.rel\ i\ s$ concludes this case.

For the second half of the conjunction, we get the following goal after simplification:

```
h : M, s ⊨ C G φ
hi : i ∈ G
hts : t ∈ M.f.rel i s
htu : C_path G t u
⊢ M, u ⊨ φ
```

Intuitively for any state u such that $t \approx_G u$, we must show $M, u \models \varphi$. Again we apply h , leaving us to show $C_path\ G\ s\ u$ which must hold when we extend $C_path\ G\ t\ u$ by first using agent i to pass from s to t . This corresponds to the $C_path.next$ constructor of C_path , using the hypotheses hts and htu to discharge the remaining goals.

7 Creating reusable definitions in Lean

Before tackling the completeness proof for CLC, we note that the proof relies in large part on lemmas and definitions taken from Pauly’s completeness for CL [22]. In paper proofs such reuse is trivial, but in Lean lemmas and definitions only apply to the syntax they are defined on, since the syntax and proof system for each logic form a distinct inductive type. Our formalization therefore gives special attention to reusability using the *typeclass* system of Lean, to limit the need for redundant copies of code for each logic. Specifically, we make use of the fact that one logic commonly extends another by adding new operators and axioms. In Lean we define a `class` for some logic in such a way that all extensions of that logic are an `instance` of that `class`. We can then construct definitions and proofs in Lean that apply to any logic that is an instance of that `class`. Doing so allows our Lean results to be reused across different logics.

We start by creating a typeclass for logics whose syntax extends that of propositional logic. More precisely, an instance of `Pformula form`

```
class Pformula (form : Type) :=
  (bot : form)
  (var : ℕ → form)
  (and : form → form → form)
  (imp : form → form → form)
```

We also introduce notation for formulas: \perp , \wedge , \rightarrow , \top , \neg , \vee , \iff . Next, we demonstrate that the language of CLC formulas `formCLC` extends propositional logic by registering an `instance`.

```
instance formulaCLC {agents : Type} : Pformula (formCLC agents) :=
  { bot := formCLC.bot,
    var := formCLC.var,
    and := formCLC.and,
    imp := formCLC.imp, }
```

We can then make our formula constructions generic over all syntaxes that have a `Pformula` instance, and Lean will automatically infer this instance when applying these constructions. For instance, the following definition gives the conjunction of a finite list of formulas.

```
def finite_conjunction {form : Type} [Pformula form] : List form → form
| [] := ⊤
| (f :: fs) := f ∧ finite_conjunction fs
```

Since all provable propositional formulas are also provable in logics that extend propositional logic, we also introduce a `class` `Pformula_ax (form : Type) [Pformula form]` that denotes the existence of a provability predicate \vdash such that $\vdash \varphi$ holds for all formulas φ provable by the axioms of propositional logic.

We create three more typeclasses relevant to CLC. Logics (extending) CL are instances of `class` `CLformula (agents : outParam Type) (form : Type) [Pformula_ax form]`, which specifies the additional operator and axioms associated with CL. Note that this typeclass inherits from `Pformula_ax` as CL extends propositional logic. Similarly we introduce `class` `Kformula (agents : outParam Type) (form : Type) [Pformula_ax form]` representing logics that extend propositional logic with individual knowledge. Lastly we create a typeclass for logics with common knowledge, which must therefore also contain individual

knowledge. This typeclass must therefore inherit from both `Pformula_ax` and `Kformula`:
`Cformula (agents : outParam Type) [hN : Fintype agents] (form : Type)
 [pf : Pformula_ax form] [kf : Kformula agents form].`

8 Formalizing Completeness

We begin by sketching the completeness proof for CLC [1, Corollary 1], which is based on a canonical model construction. For each consistent formula, we create a finite model for which that formula is true at some state. We focus on finite models because the 6th true playability condition, that $E(s)(\emptyset)$ is principal, is always met in finite models [12]. Doing so simplifies the proof that the effectivity structure in these models is truly playable. In the process we demonstrate that CLC has the finite model property.

We create such a finite model by first creating a single infinite canonical coalition model where every consistent formula is true in some state. This canonical coalition model is defined analogously to an epistemic coalition model, but without epistemic relations, and where the effectivity structure only meets the first 5 true playability conditions. Then, given some consistent formula φ , we filter the canonical coalition model and add epistemic relations to form a finite epistemic coalition model for which φ is true in some state.

8.1 Formalizing the canonical coalition model

We start by building the canonical coalition model. We define $M^C := (F^C, V^C)$, where $F^C := (S^C, E^C)$ as follows:

- S^C is the set of all maximal CLC-consistent sets of formulas.
- E^C is the playable effectivity structure:
 - $X \in E^C(s)(N)$ iff $\forall \varphi, \tilde{\varphi} \subseteq X^c \rightarrow [\emptyset]\varphi \notin s$, where $\tilde{\varphi} := \{t \in S^C \mid \varphi \in t\}$
 - $X \in E^C(s)(G)$ iff $\exists \varphi, \tilde{\varphi} \subseteq X \wedge [G]\varphi \in s$, when $G \neq N$
- V^C is the usual valuation function : $s \in V^C(p)$ iff $p \in s$.

A playable effectivity structure must meet the first 5 true playability conditions. A set Σ of formulas is consistent iff there are no $\sigma_1, \dots, \sigma_n \in \Sigma$ such that $\vdash (\sigma_1 \wedge \dots \wedge \sigma_n) \rightarrow \perp$. The proof that M^C is indeed a coalition model is analogous to the proof by Pauly [22, Lemma 5.2] for CL. In Lean, we use our generic classes for propositional logic and CL to define a canonical coalition model for any logic that extends CL, so long as that logics axiomatic system is consistent (as required by the hypotheses `hnpr : $\neg \vdash (\perp : \text{form})$`):

```
def canonical_model_CL [Nonempty agents]
  [Pformula_ax form] [CLformula agents form]
  (hnpr :  $\neg \vdash (\perp : \text{form})$ ) : modelCL agents
```

Note that this definition includes the proof that the defined effectivity structure is playable.

8.2 Filtering the canonical model

Given some φ , we filter S^C into a finite set of states S^f . We will prove the properties of S^C transfer to S^f and shows it enjoys some additional properties essential to constructing a playable model. We achieve this by creating a finite closure $cl(\varphi)$, defined as the smallest set satisfying the following:

1. For any $\psi \in cl(\varphi)$, all subformulas of ψ are also contained in $cl(\varphi)$.
2. For any $\psi \in cl(\varphi)$, if ψ is not of the form $\neg\chi$, then $\neg\psi \in cl(\varphi)$.
 ($cl(\varphi)$ is thus closed under single negations.)

28:10 Formalizing Completeness of CLC in Lean

3. If $C_G\varphi \in cl(\varphi)$, then for all $i \in G$, $K_i C_G\varphi \in cl(\varphi)$.
4. If $[G]\varphi \in cl(\varphi)$, then $C_G[G]\varphi \in cl(\varphi)$.

This definition is adjusted slightly compared to the work by Ågotnes and Alechina [1], as we allow the formula $C_\emptyset\psi$, and thus do not need to consider the case $G = \emptyset$ separately. Additionally, we change the first requirement such that all subformulas of any $\psi \in cl(\varphi)$ are contained in the closure, rather than just subformulas of φ . This change is needed to prove the truth lemma, where we will perform induction on an arbitrary $\psi \in cl(\varphi)$. For Ågotnes and Alechina [1] this adjusted requirement is already met when $cl(\varphi)$ contains all subformulas of φ , because their syntax is defined from different base operators. The closure definition thus illustrates that small implementation choices early in the formalization process can have unintended effects later in the proof that may not be immediately obvious. Luckily, the interactive environment of a theorem prover made the consequences of this change clear, and made the necessary changes easy to implement and test.

The set $cl(\varphi)$ can be built recursively on the structure of φ , and this is also how we define it in Lean. For instance, for the case $cl(C_G\psi)$, the closure must include $cl(\psi) \cup \{C_G\psi, \neg(C_G\psi)\} \cup \{K_i C_G\psi : i \in G\} \cup \{\neg(K_i C_G\psi) : i \in G\}$. Note that the sets $\{K_i C_G[G]\psi : i \in G\}$ and $\{\neg(K_i C_G\psi) : i \in G\}$ are finite, because G is finite. In Lean we define the union of these two sets as follows:

```

noncomputable def cl_C {agents : Type} [Fintype agents] (G : Set agents)
  (φ : formCLC agents) : Finset (formCLC agents) :=
  Finset.image (fun i => K i (C G φ)) (toFinset G) ∪
  Finset.image (fun i => (¬ K i (C G φ))) (toFinset G)

```

In addition to defining a set, the above definition also guarantees that the set is finite, using the `Finset` datatype. We create this resulting `Finset` by first mapping the set of agents G from a `Set` to a `Finset`. Lean can infer this is possible, because N is finite, as indicated by the hypothesis `[Fintype agents]`. Then we can take the image of G as desired.

In Lean, we then need to prove that our closure $cl(\phi)$, defined recursively on the structure of the formula ϕ indeed meets the four requirements described above. To do so, we first define a subformula as an inductive proposition with cases for each operator. For instance we define two cases for the \wedge -operator: `and_left` $\{\varphi \psi\} : \text{subformula } \varphi (\varphi \wedge \psi)$ and `and_right` $\{\varphi \psi\} : \text{subformula } \psi (\varphi \wedge \psi)$. Additionally, we add two cases for the requirements that our sub-formula definition must be reflexive and transitive. Given this definition, we tackle the four proofs about the closure. Although in a paper proof all four requirements are trivially met by definition of the closure, in Lean this is only the case for the last two. The first two requirements both need inductive proofs on φ where for every case we iteratively consider all possible $\psi \in cl(\varphi)$. For instance if $\varphi = \chi_l \wedge \chi_r$, we consider the cases where $\psi = \chi_l \wedge \chi_r$, $\psi = \neg(\chi_l \wedge \chi_r)$, $\psi \in cl(\chi_l)$ and $\psi \in cl(\chi_r)$. These proofs are not difficult, but considering each case creates long and tedious proofs.

Now that we have defined the closure, given some φ , we can filter M^C through $cl(\varphi)$, to construct a finite model $M^f := (F^f, V^f)$, where $F^f := (S^f, E^f, \{\sim_i^f : i \in N\})$. We construct M^f as follows:

$$\begin{array}{ll}
S^f & := \{(s^f) \mid s \in S^C\}, & \text{where } s^f := (s \cap cl(\varphi)) \\
E^f & := \begin{array}{l} X \in E^f(s)(N) \\ X \in E^f(s)(G \subset N) \end{array} & \begin{array}{l} \text{iff } \exists t \in S^C, s^f = t^f \text{ and } \widetilde{\phi}_X \in E^C(t)(N) \\ \text{iff } \forall t \in S^C, s^f = t^f \Rightarrow \widetilde{\phi}_X \in E^C(t)(G) \end{array} \\
\sim_i^f & := (s^f) \sim_i^f (t^f) & \text{iff } \{\varphi \mid K_i \varphi \in s^f\} = \{\varphi \mid K_i \varphi \in t^f\} \\
V^f & := s \in V(p) & \text{iff } p \in s,
\end{array}$$

where $\phi_X := \bigvee_{s^f \in X} \phi^{s^f}$ is the disjunction of a set of filtered states, $\phi^{s^f} := \bigwedge_{\psi \in s^f} \psi$ is the

conjunction of the formulas in a filtered state, and $\tilde{\psi} := \{t \in S^C \mid \psi \in t\}$. Note that S^f is finite because $cl(\varphi)$ is, and that \sim_i^f is an equivalence relation by definition. For this model we will use the notation $s^f \approx_G^f t^f := (s^f, t^f) \in (\bigcup_{i \in G} \sim_i^f)^*$ for the common knowledge path.

These definitions can be translated quite directly into Lean, although it might not look so direct, due to again having to distinguish between sets and finite sets in Lean. Thus, to define S^f in Lean, we start with $cl(\varphi)$, as this is a `Finset`. We take the powerset of $cl(\varphi)$, which Lean knows must also be finite. This finite powerset is filtered with `Finset.filter` to include only those elements s^f for which there exists some $s \in S^C$ such that $s^f = s \cap cl(\varphi)$. In order to check $s^f = s \cap cl(\varphi)$, we need both to be of the same data type and therefore convert both to sets. Finally, we pair each state s^f with a proof that it is produced by the filter, using `Finset.attach`.

```
def S_f {agents form : Type} (m : modelCL agents) [SetLike m.f.states form]
  (cl : form → Finset (form)) (φ : form) : Type :=
  Finset.attach (Finset.filter
    (λ sf => ∃ s : m.f.states, {x | x ∈ cl φ ∧ x ∈ s} = {x | x ∈ sf}))
    (Finset.powerset (cl φ))
```

Note that we do impose strong requirements on the model in the definition of S^f , so long as the states contain a set of formulas, as enforced by the hypothesis `[SetLike m.f.states form]`. Doing so allows us to keep our definition simpler and more generic, by removing the need for hypotheses needed to create our canonical model (for instance that N is nonempty).

Next we define the subformulas ϕ_X and ϕ^{s^f} which are needed to define E^f :

```
variable {agents form : Type} [Pformula form]
  {m : modelCL agents} [SetLike m.f.states form]
  {cl : form → Finset (form)} {φ : form}

noncomputable def phi_s_f (sf : S_f m cl φ) : form :=
  finite_conjunction (Finset.toList (sf.1))

noncomputable def phi_X_list : List (S_f m cl φ) → List form
  | List.nil => List.nil
  | (sf :: ss) => ((phi_s_f sf) :: phi_X_list ss)

noncomputable def phi_X_finset (X : Finset (S_f m cl φ)) : form :=
  finite_disjunction (phi_X_list (Finset.toList X))

noncomputable def phi_X_set (X : Set (S_f m cl φ)) : form :=
  phi_X_finset (Finite.toFinset (Set.toFinite X))
```

Here the `variable` statement adds the hypotheses to each subsequent declaration. Defining ϕ^{s^f} in Lean (`phi_s_f`) is as simple as converting our finite set to a list (putting the elements in an arbitrary order) and then creating a conjunction from that list. We then define ϕ_X in several steps. First we define a function `phi_X_list` to map X to $\{\phi^{s^f} : s^f \in X\}$. Next, we define ϕ_X for finite sets, as we can convert that finite set to a list, map it to formulas with `phi_X_list`, and then return the disjunction of that mapped list. Lastly, for the `Set` datatype, we define ϕ_X by converting to a `Finset`, which we can do because $X \subseteq S^f$ is a set of a finite type.

Although it would suffice logically to work with a `List` of filtered states, we provide the definition `phi_X_set` in higher generality for two reasons. Firstly, this approach more closely matches the definitions in the paper proof. Secondly, the definition should intuitively

not depend on a choice of order on the states, so we make this independence explicit in the datatypes. Splitting this definition up into three parts may seem to add complexity. However, it allows us to define lemmas about each data type, thereby breaking proofs down into smaller steps. We can prove lemmas more easily for a list, which is ordered, finite, and allows induction. Then, it is easy to show that if some lemma holds for a list, it must work for a list created from a (finite) set. Keeping track of the converted datatypes and how they relate to one another within a single lemma is non-trivial (and not always possible) in Lean. Thus in Lean we often prove some result about ϕ_X across three lemmas, one for each datatype: `Set`, `Finset` and `List`.

Given our definition(s) for ϕ_X , it is straightforward to define E^f , and then our whole model M^f . We thus omit these Lean translations.

8.3 Playability of the filtered canonical model

We prove that M^f meets the requirements for being a CLC model. We have to show that for an arbitrary state s^f in the filtered model, $E^f(s^f)$ is truly playable [1, Proposition 1]. This proof relies on the fact that E^f is defined from E^c . We are therefore able to exploit the fact that the first five true playability conditions hold in E^c to prove that they must also hold in E^f . In Lean we really benefit from our generic typeclasses here, as our proofs that E^c meets those playability conditions are written to hold for any logic that extends CL. Recall that the final true playability condition must hold in E^f because M^f is finite [12].

To formalize this proof, we first expand the proof by Ågnotes and Alechina, into a proof with similar levels of detail to a Lean formalization. We aim for a level of detail such that each step in our extended paper proof translates roughly into one step in Lean, possibly with some reshaping. To illustrate this procedure and the level of detail required we present our extended paper proof for Condition 3 of true playability (Definition 1), which was the most interesting to formalize in Lean.

$E^f(s^f)$ is N -maximal (for every $X \subseteq S^f$, if $(S^f \setminus X) \notin E^f(s^f)(\emptyset)$, then $X \in E^f(s^f)(N)$) is shown by the following sequence of proof steps:

1. Pick some $X \subseteq S^f$ such that $X^c = (S^f \setminus X) \notin E^f(s^f)(\emptyset)$.
2. $\neg(X^c \in E^f(s^f)(\emptyset))$, from Step 1.
3. $\neg(\forall t \in S^c : s^f = t^f \Rightarrow \widetilde{\phi_{X^c}} \in E^c(t)(\emptyset))$, from Step 2 and by definition of E^f .
4. $\exists t \in S^c : s^f = t^f$ and $\widetilde{\phi_{X^c}} \notin E^c(t)(\emptyset)$, from Step 3.
5. $\vdash \phi_{X^c} \leftrightarrow \neg\phi_X$, because $\vdash \phi_{S^f}$ and $\forall s, t \in S^{c'}, s^f \neq t^f \Rightarrow \vdash \phi^{s^f} \rightarrow \neg\phi^{t^f}$.
6. $\exists t \in S^c, s^f = t^f$ and $\neg\widetilde{\phi_X} \notin E^c(t)(\emptyset)$, from Step 4 and 5.
7. $\exists t \in S^c, s^f = t^f$ and $(\widetilde{\phi_X})^c \notin E^c(t)(\emptyset)$, from Step 6, because all $s \in S^c$ are maximally consistent.
8. $\exists t \in S^c, s^f = t^f$, and $\widetilde{\phi_X} \in E^c(t)(N)$, provided $s = t$, from Step 7, because $E^c(t)$ is N -maximal: $(\widetilde{\phi_X})^c \notin E^c(t)(\emptyset) \Rightarrow \widetilde{\phi_X} \in E^c(t)(N)$
9. $X \in E^f(s^f)(N)$, from Step 8, by definition of E^f .

In this expanded proof the only step that is not straightforward to formalize in Lean is Step 5. We elaborate on the process of formalizing this step as it is a good illustration of working with our Lean definition(s) of ϕ_X . For space we do not expand on the proofs that $\vdash \phi_{S^f}$ and $\forall s, t \in S^{c'}, s^f \neq t^f \Rightarrow \vdash \phi^{s^f} \rightarrow \neg\phi^{t^f}$. In Lean these proofs are given in the lemmas `univ_disjunct_provability` and `unique_s_f` respectively. To show $\vdash \phi_{X^c} \leftrightarrow \neg\phi_X$, in the \Leftarrow direction we first use a lemma defined elsewhere called `phi_X_set_disjunct_of_disjuncts`, which proves that $\vdash (\neg\phi_X \rightarrow \phi_Y) \Leftrightarrow \vdash (\phi_{X \cup Y})$, to change the goal to $\vdash \phi_{X \cup X^c}$. This lemma is trivial on paper by definition of ϕ_X , but requires

unfolding the definitions and their respective datatype in Lean. Next we change the goal to $\vdash \phi_{S^f}$, with the lemma `union_compl_self`, because the union of a set and its complement is the universe. Lastly, we can use `univ_disjunct_provability` to prove this goal:

```
apply (phi_X_set_disjunct_of_disjuncts _ _).mpr
rw [union_compl_self X]
apply univ_disjunct_provability
```

For the \Rightarrow direction, we cannot so immediately apply the relevant lemma `unique_s_f`, as this lemma refers to single elements (filtered states) in our sets X and X^c . We will eventually use an inductive proof to consider a single element in X . In Lean we will therefore need to work with a `list` datatype and will need to unfold our definitions of ϕ_X accordingly. We create a lemma per data type: `phi_X_set_unique`, `phi_X_finset_unique` and `phi_X_list_unique`, which show the slightly generalized result that $\vdash \phi_X \rightarrow \neg \phi_Y$ holds for any disjoint sets $X, Y \subseteq S^f$. Our actual proof simply applies this first lemma:

```
apply phi_X_set_unique hcl (compl_inter_self X)
```

where `compl_inter_self` is a lemma proving that a set and its complement are disjoint, and `hcl` is a proof that our closure is closed under single negations. The need to pass this condition of our closure forward highlights how verification makes explicit exactly when and for which purpose specific hypotheses are used. In this case we will eventually pass this hypothesis to the lemma `unique_s_f`.

The interesting work is within `phi_X_list_unique`. We have converted X into `sfs : List (S_f M cl φ)`, where `S_f M cl φ` corresponds to S^f for the canonical model M and Y into `tfs : List (S_f M cl φ)`. The proof is first inductive on X . The case when X is empty is trivial because ϕ_{\emptyset} is defined as \perp in Lean. Thus we unfold the definitions `phi_X_list` and `finite_disjunction`, and then use `explosion`, which represents the propositional lemma $\forall \psi, \vdash \perp \rightarrow \psi$.

```
induction' sfs with sf sfs ihsfs generalizing tfs
· -- sfs = []
  simp only [phi_X_list, finite_disjunction, explosion]
```

So let `sfs` contain at least one element `sf` at the head of the list, and call the rest of the list `sfs'`. Then we can split $\vdash (\phi^{s^f} \vee \phi_{X'}) \rightarrow \neg \phi_Y$ into two cases, where the latter follows from the induction hypothesis `ihsfs`. Note that the `sorry` keyword in the snippet below indicates a proof omitted from the paper for presentation purposes; the omitted proof is included below and in the full formalization source code.

```
· -- sfs = sf :: sfs'
  simp only [phi_X_list, finite_disjunction]
  apply or_cases
  --  $\vdash \phi_{sf} \rightarrow \neg \phi_{tfs}$ 
  · sorry -- Proof included below
  --  $\vdash \phi_{sfs'} \rightarrow \neg \phi_{tfs}$ 
  · apply ihsfs
    apply List.disjoint_of_disjoint_cons_left hdis
```

Here the lemma `List.disjoint_of_disjoint_cons_left` shows that `sfs'` and `tfs` must be disjoint when `sfs` and `tfs` are disjoint (represented by `hdis` in Lean).

28:14 Formalizing Completeness of CLC in Lean

For the case $\vdash \phi^{s^f} \rightarrow \neg \phi_Y$, we perform induction on Y (\mathbf{tfs}). Again here the base case of an empty list holds by propositional logic, so assume \mathbf{tfs} contains at least one element \mathbf{tf} at the head of the list, and call the rest of the list \mathbf{tfs}' . We now look at the contrapositive of our goal: $\vdash (\phi^{t^f} \vee \bigvee_{u^f \in \mathbf{tfs}'} \phi^{u^f}) \rightarrow \neg \phi^{s^f}$. Again we have two cases. For the former we can apply `lemma unique_s_f` where we prove $s^f \neq t^f$ by contradiction, based on the disjointness of both lists. The latter case is solved with the (new) induction hypothesis (`ih \mathbf{tfs}`).

```

· induction' tfs with tf tfs ihtfs
  · simp only [phi_X_list, finite_disjunction]
    exact mp _ _ (p1 _ _) iden -- applying propositional lemmas
  · simp [finite_disjunction] at *
    -- contrapositive
    refine contrapos.mp (cut dne (or_cases ?_ ?_))
    -- ⊢ phi tf → ¬ phi sf
    · apply unique_s_f hcl
      by_contra h
      simp only [h] at hdis
    -- ⊢ phi tfs' → ¬ phi sf (proved with ihtfs and propositional lemmas)
    · rw [←contrapos]
      exact cut dne (ihtfs hdis.2.1 hdis.2.2)

```

8.4 Truth lemma

Next we show that in the filtered canonical model all formulas contained in a state are also true in that state. Recall that M^f is the model created when filtering M^C through $cl(\varphi)$.

► **Lemma 5** (CLC Truth Lemma [1, Theorem 1]). *For all $s \in S^C$ and $\psi \in cl(\varphi)$, we have $M^f, s^f \models \psi$ iff $\psi \in s^f$.*

This proof is by induction on ψ . For space reasons we include only the proof for $C_G\psi$: $M^f, s^f \models C_G\psi$ iff $C_G\psi \in s^f$, and specifically the \Leftarrow direction, as this was the most interesting to formalize. Given $C_G\psi \in s^f$, and some state t^f such that $s^f \approx_G^f t^f$, we need to show $M^f, t^f \models \psi$. This proof is inductive on the common knowledge path from s^f to t^f . Thus, the details of this proof depend on how exactly we defined the common knowledge path in Lean.

Let the length of a common knowledge path be the number of states in the path between our first state (s^f) and our last state (t^f). In this case we may describe a common knowledge path from s^f to t^f as $\langle s^f, \sim_{i_0}^f, u_1^f, \sim_{i_1}^f, u_2^f, \dots, u_n^f, \sim_{i_n}^f, t^f \rangle$, such that $s_0^f \sim_{i_0}^f u_1^f, u_1^f \sim_{i_1}^f u_2^f, \dots, u_n^f \sim_{i_n}^f t^f$ and $\{i_0, i_1, \dots, i_n\} \subseteq G$. We will perform induction on the length n of this path.

For the base case of our inductive proof, let $n = 0$. Thus, we consider a path $\langle s^f, \sim_{i_0}^f, t^f \rangle$, matching the base case of our Lean implementation:

```
| done (hi : i ∈ G) (hst : t ∈ m.f.rel i s) : C_path G s t
```

Thus we need to prove that $M, t \models \psi$, given that $s^f \sim_{i_0}^f t^f$, for some $i_0 \in G$. This base case differs from a more traditional inductive proof on a common knowledge path, like the proof by Ågotnes and Alechina [1], where the base case is simply the starting state, with the path being $\langle s^f \rangle$. Note that this is equivalent to our base case with the additional assumption that $s^f = t^f$, as we must by reflexivity have $s^f \sim_{i_0}^f s^f$.

Next our inductive step needs to match our recursive case in Lean:

```
| next (hi : i ∈ G) (hsu : u ∈ m.f.rel i s) (ih : C_path G u t) :
  C_path G s t
```

Here we build the path recursively from the front: so when looking at the path from s^f to t^f , we consider first the individual knowledge relation from s^f to the second state in the path. Then we recursively define the rest of the path from the second state to t^f . Our inductive step must match this format. Let the first state between s^f and t^f be u^f , where $s^f \sim_i^f u^f$ for some $i \in G$, and let the common knowledge path for group G of length n be $\langle u^f, \sim_{i_0}^f, u_1^f, \sim_{i_1}^f, u_2^f, \dots, u_n^f, \sim_{i_n}^f, t^f \rangle$. The inductive hypothesis states that if $C_G\psi \in u^f$, then $M^f, t^f \models \psi$. Again, this approach to the inductive step is different from the more usual inductive proof on a common knowledge path by Ågotnes and Alechina [1]. In their case the inductive step splits a path of length $n + 1$ into a path from the starting state (s^f) to the n th state in the path, and a single knowledge relation from the n th to the end state (t^f).

Note that for our inductive proof on the common knowledge path, both in the base case and in the inductive case we need to prove something (that ψ holds in the base case, that it contains $C_G\psi$ for the inductive step) about a state (t^f for the base case, w^f for the inductive step) which is connected from s^f by an individual knowledge relation for some agent in G . Thus we now show that for any state w^f , where there is a relation $s^f \sim_j^f w^f$ for some $j \in G$, we must have both $M^f, w^f \models \psi$ and $C_G\psi \in w^f$. From $C_G\psi \in s^f$ we must have $K_j(C_G\psi) \in s^f$ by definition of $cl(C_G\psi)$, propositional logic, and axioms **(C)**, **(K)** and **(RN)**. Thus by definition of \sim_i^f we must also have $K_j(C_G\psi) \in w^f$. Then we must also have $C_G\psi \in w^f$ by axiom **(T)**. Hereby we have proven $M^f, t^f \models \psi$ for the inductive step in our proof. Additionally, from $C_G\psi \in w^f$, we know $\psi \in w^f$, by axioms **(T)**, **(C)**, **(K)** and **(RN)**. Note that by the inductive hypothesis for the truth lemma ($\forall s \in S^C, M^f, s^f \models \psi \leftrightarrow \psi \in s^f$), we must then also have $M^f, w^f \models \psi$. Therefore we have proven $M^f, t^f \models \psi$ for the base case in our proof.

8.5 Finalizing the completeness proof

It remains to prove the final theorem:

► **Theorem 6** (Completeness of CLC [1, Corollary 1]). $\forall \varphi, \models \varphi \Rightarrow \vdash \varphi$

We prove the contrapositive by showing that every formula not provable by CLC is not globally valid: $\not\vdash \varphi \Rightarrow \not\models \varphi$. From $\neg \vdash \varphi$ we know that $\{\neg\varphi\}$ must be a consistent set. By Lindenbaum's lemma [24] the set can thus be extended into some maximally consistent set Σ that is equal to some state $s \in S^C$. Note that when filtered through $cl(\varphi)$, we will still have $\neg\varphi \in s^f$. By Lemma 5 $\neg\varphi$ is true in that filtered state, and thus φ is not. Thus φ is not globally valid.

We have thus verified the proof theory and model theory of CLC relate to each other as expected by proving both soundness and completeness. All Lean lemmas and definitions about (filtered) canonical model construction can be reused to prove that CL and CLK are also sound and complete (see [21] for details). For CL, as mentioned previously, this is done by proving the truth lemma for the canonical coalition model for CL. For Coalition Logic with individual knowledge (CLK), the proofs are analogous to the proofs presented here, omitting any parts related to common knowledge.

9 Conclusion and Discussion

In this paper, we have described the successful implementation of soundness and completeness proofs for CLC in Lean. Our project consists of approximately 6,000 lines of code. Of these, approximately 300 lines are specific to Coalition Logic (CL), 700 are specific to Coalition Logic with Knowledge operators (CLK), and 1,100 to Coalition Logic with Common knowledge

operators (CLC). The remaining almost 4,000 lines are shared between the three. In addition, we make extensive use of the Lean mathematical library Mathlib. We will not mention a De Bruijn factor for our development, as there is no direct comparison possible between the scope of our work and any of the relevant papers.

Much of the complexity of our formalization comes from the need to deal with finiteness in Lean. To access properties of finiteness in Lean, we needed to use specific data types. This is most notable in our formalization of ϕ_X , where we create three different definitions for when X is a `Set`, a `Finset` (finite set) and a `List`. Of the three mentioned data types only the `List` is ordered in Lean (in our case, when converting from a (finite) set the order is arbitrary) and therefore allows us to iterate over elements. However when translating some (finite) set into a list we often need to keep track of relevant properties about the initial `Set`. For instance, we may need to remember that our resulting `List` contains no repeating elements. We are therefore often required to create separate lemmas for each data type, and manually pass such information forward. These translations consequently add a lot of work. However, each individual step was relatively simple with the existing Mathlib library [25]. Additionally, some of these challenges are likely exacerbated by our goal to keep our Lean proofs reasonably similar to their respective paper proofs. For instance, in our formalization we define finite conjunctions and disjunctions recursively. However to show a finite conjunction is provable or is contained in some state, we simply need to show that all conjuncts are provable or are contained within that state. Similarly for finite disjunctions we aim to show that one disjunct is provable or contained within the state. Thus a deeper embedding using Lean’s native \forall and \exists quantifiers may have been more natural.

Another difficulty with formalization is that there are many trivial lemmas that need detailed proofs in Lean, which makes formalization cumbersome and time-consuming. This is especially notable with the lemmas about the finite closure (cl), for instance that it is closed under single negations. Despite being trivial by our definition of cl , the proof in Lean is long because of how many cases need to be considered. This highlights the need for continued work on increasing automation in Lean. Specifically, these long but trivial inductive proofs would be ideal candidates for better automation.

Despite these challenges, one of the main advantages of formalizing this proof is that it required us to be precise about exactly when we were using hypotheses and assumptions. In our case, this led to us easily showing that the completeness proof for CLC described by Ågotnes and Alechina [1] also holds if we extend the syntax to also allow formulas of the form $C_{\emptyset}\varphi$. Programmatic formalization lends itself well to these tests of generalization: it automates the work of re-checking an entire proof every time a hypothesis is slightly changed or removed [4, 10].

Aside from dealing with the nature of formalization itself, one of the goals of our research was to allow for reuse of lemmas and definitions across different logics. To this end we introduced Lean `classes` for each logics syntax and axiomatic system. Importantly, we were able to define the canonical model M^C for these classes, such that the model can be built for CL and any of its extensions. Additionally we provided a large number of lemmas defined using our classes for propositional logic, CL, CLK and CLC. We hope that an increasing library of these kinds of proofs can aid future research into formalizing modal logics, especially work on formalizing the other types of Epistemic Coalition Logic described by Ågotnes and Alechina [1].

We note, however, that we did not add semantics to our class definitions. This choice was made as the semantics are only used in inductive proofs. We could not use classes for inductive proofs, as they act as minimum requirements for the syntax and proof system of

the logic. However, each individual case in an inductive proof could be separated into its own lemma if the semantics was added to the generic classes. Future work could thus look into expanding our classes and creating such generic proofs. Even more interesting would be to define the logics in such a way that we can use generic data structures for inductive proofs.

References

- 1 Thomas Ågotnes and Natasha Alechina. Coalition logic with individual, distributed and common knowledge. *Journal of Logic and Computation*, 29(7):1041–1069, 2019. doi:10.1093/logcom/exv085.
- 2 Mussab Alaa, Aos Alaa Zaidan, Bilal Bahaa Zaidan, Mohammed Talal, and Miss Laiha Mat Kiah. A review of smart home applications based on internet of things. *Journal of Network and Computer Applications*, 97:48–65, 2017. doi:10.1016/j.jnca.2017.08.017.
- 3 Jeremy Avigad, Leonardo de Moura, Soonho Kong, and Sebastian Ullrich. Theorem proving in lean 4. Accessed 2024-01-18. URL: https://lean-lang.org/theorem_proving_in_lean4/.
- 4 Anne Baanen, Alexander Bentkamp, Jasmin Blanchette, Johannes Hölzl, and Jannis Limperg. *The Hitchhiker’s Guide to Logical Verification*. Vrije Universiteit Amsterdam, 2021 edition, 2021.
- 5 Colm Baston and Venanzio Capretta. Game forms for coalition effectivity functions. In *TYPES 2019*, pages 26–27, 2019. URL: <https://nottingham-repository.worktribe.com/output/2681068>.
- 6 Bruno Bentzen. A Henkin-style completeness proof for the modal logic S5. In *Lecture Notes in Computer Science*, pages 459–467. Springer International Publishing, 2021. doi:10.1007/978-3-030-89391-0_25.
- 7 Lubor Budaj. Formalization of modal logic S5 in the Coq proof assistant, 2022. Bachelor’s Thesis, University of Groningen. URL: https://fse.studenttheses.ub.rug.nl/28482/1/BSc_Thesis_final.pdf.
- 8 Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The Lean Theorem Prover (System Description). In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 9195, pages 378–388. Springer, 2015. doi:10.1007/978-3-319-21401-6_26.
- 9 Asta Halkjær From. Formalized soundness and completeness of epistemic logic. In Alexandra Silva, Renata Wassermann, and Ruy de Queiroz, editors, *Logic, Language, Information, and Computation*, pages 1–15, Cham, 2021. Springer International Publishing.
- 10 Herman Geuvers. Proof assistants: History, ideas and future. *Sadhana*, 34:3–25, 2009.
- 11 Balaji Parasumanna Gokulan and Dipti Srinivasan. An introduction to multi-agent systems. In Dipti Srinivasan and Lakhmi C. Jain, editors, *Innovations in Multi-Agent Systems and Applications*, pages 1–27. Springer, Berlin, Heidelberg, 2010. doi:10.1007/978-3-642-14435-6_1.
- 12 Valentin Goranko, Wojciech Jamroga, and Paolo Turrini. Strategic games and truly playable effectivity functions. *Autonomous Agents and Multi-Agent Systems*, 26(2):288–314, March 2013. doi:10.1007/s10458-012-9192-y.
- 13 Frans C. A. Groen, Matthijs T. J. Spaan, Jelle R. Kok, and Gregor Pavlin. Real world multi-agent systems: Information sharing, coordination and planning. In *TbiLLC ’05*, volume 4363 of *LNCS*, pages 154–165, Cham, 2007. Springer. doi:10.1007/978-3-540-75144-1_12.
- 14 Joni Helin. Combining deep and shallow embeddings. *Electronic Notes in Theoretical Computer Science*, 164(2):61–79, 2006. doi:10.1016/j.entcs.2006.10.005.
- 15 Thierry Lecomte, Thierry Servat, and Guilhem Pouzancre. Formal methods in safety-critical railway systems. In *10th Brazilian symposium on formal methods*, pages 29–31, August 2007.
- 16 Pierre Lescanne and Jérôme Poussegur. Dynamic logic of common knowledge in a proof assistant. (preprint), 2007. doi:10.48550/arXiv.0712.3146.
- 17 Jiayu Li. Formalization of pal-s5 in proof assistant. (preprint), 2020. doi:10.48550/arXiv.2012.09388.

- 18 Cláudia Nalon, Lan Zhang, Clare Dixon, and Ullrich Hustadt. A resolution prover for coalition logic. In Fabio Mogavero, Aniello Murano, and Moshe Y. Vardi, editors, *Proceedings 2nd International Workshop on Strategic Reasoning, SR 2014, Grenoble, France, April 5-6, 2014*, volume 146 of *EPTCS*, pages 65–73, 2014. doi:10.4204/EPTCS.146.9.
- 19 Paula Neeley. A formalization of dynamic epistemic logic. Master’s thesis, Carnegie Mellon University, 2021.
- 20 Kai Obendrauf. CL-Lean Formalization. Software, swhId: swh:1:dir:5679444e6dc3b26cd7f1c54786bff9be89541c19 (visited on 2024-07-08). URL: <https://github.com/kaiobendrauf/cl-lean>.
- 21 Kai Obendrauf. Formalizing completeness proofs for coalition logic with and without common knowledge in Lean. Master’s thesis, Vrije Universiteit Amsterdam, 2023. doi:10.5281/zenodo.12582708.
- 22 Marc Pauly. A modal logic for coalitional power in games. *Journal of Logic and Computation*, 12(1):149–166, 2002. doi:10.1093/logcom/12.1.149.
- 23 Daniel Selsam, Sebastian Ullrich, and Leonardo de Moura. Tabled typeclass resolution. *CoRR*, abs/2001.04301, 2020. arXiv:2001.04301.
- 24 Alfred Tarski. Über einige fundamentale Begriffe der Metamathematik. *Sprawozdania z Posiedzeń Towarzystwa Naukowego Warszawskiego. Wydział III*, 23:22–29, 1930.
- 25 The mathlib Community. The Lean mathematical library. In *CPP 2020*, pages 367–381, New York, NY, USA, 2020. ACM. doi:10.1145/3372885.3373824.
- 26 Johan Van Benthem. Games in dynamic-epistemic logic. *Bulletin of Economic Research*, 53(4):219–248, 2001. doi:10.1111/1467-8586.00133.
- 27 Wiebe van der Hoek and Michael J. Wooldridge. Logics for multiagent systems. *AI Mag.*, 33(3):92–105, 2012. doi:10.1609/aimag.v33i3.2427.
- 28 Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *POPL ’89, Principles of Programming Languages*, pages 60–76. ACM, 1989. doi:10.1145/75277.75283.

Conway Normal Form: Bridging Approaches for Comprehensive Formalization of Surreal Numbers

Karol Pał  

University of Białystok, Poland

Cezary Kaliszyk  

University of Melbourne, Australia

University of Innsbruck, Austria

Abstract

The proper class of Conway’s surreal numbers forms a rich totally ordered algebraically closed field with many arithmetic and algebraic properties close to those of real numbers, the ordinals, and infinitesimal numbers. In this paper, we formalize the construction of Conway’s numbers in Mizar using two approaches and propose a bridge between them, aiming to combine their advantages for efficient formalization. By replacing transfinite induction-recursion with transfinite induction, we streamline their construction. Additionally, we introduce a method to merge proofs from both approaches using global choice, facilitating formal proof. We demonstrate that surreal numbers form a field, including the square root, and that they encompass subsets such as reals, ordinals, and powers of ω . We combined Conway’s work with Ehrlich’s generalization to formally prove Conway’s Normal Form, paving the way for many formal developments in surreal number theory.

2012 ACM Subject Classification Theory of computation → Interactive proof systems

Keywords and phrases Surreal numbers, Conway normal form, Mizar

Digital Object Identifier 10.4230/LIPIcs.ITP.2024.29

Supplementary Material *Software*: <http://cl-informatik.uibk.ac.at/cek/itp2024/>

Funding Supported by ERC PoC grant no. 101156734 *FormalWeb3* and Cost action CA20111 EPN.

1 Introduction

Surreal Numbers, developed by John Conway [5], are fascinating for several reasons. Their construction relies on two intuitively simple recursive definitions: as new numbers are built, an order relation on the numbers is extended. They form a totally ordered algebraically closed field, denoted by \mathbb{No} that contains (up to isomorphism) the reals, the ordinals, infinitesimal numbers, as well as great ones, for example $\frac{1}{\omega}$, $\sqrt{\omega}$, $\omega^{\omega^{\omega}}$. Despite their intuitively simple definition, they form a proper class of numbers inductively while simultaneously defining an order on the class recursively. In the seventies, when they were first considered, mathematics was not ready for such definitions, considering them unsure or even unsafe. Even Conway [5] wondered if such definitions were meaningful and later referred to their construction as “remarkable”.

Fortunately, the informal concept intrigued several mathematicians and gave rise to work on the foundations of such concepts [7, 11, 23]. Conway saw the possibility of defining a model for surreal numbers in Neumann-Bernays-Gödel (NBG) set theory with global choice. This has later been completed by Ehrlich [8]. There are also several more detailed proofs of the properties of surreal numbers as a field; however, the more formal ones only cover their simpler properties [1, 11, 23]. Among these, Schleicher’s work [22] has been essential for our formalization.

There are two approaches to defining surreal numbers. The first approach is closer to Conway’s convention: it begins by considering the quotient of surreal numbers with respect to the equivalence relation (defined by $x \leq y \wedge y \leq x$) and proceeds to demonstrate that it forms



© Karol Pał and Cezary Kaliszyk;

licensed under Creative Commons License CC-BY 4.0

15th International Conference on Interactive Theorem Proving (ITP 2024).

Editors: Yves Bertot, Temur Kutsia, and Michael Norrish; Article No. 29; pp. 29:1–29:18

Leibniz International Proceedings in Informatics



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

a pre-order, employing switching between representatives of the equivalence classes. On the other hand, the second approach defines the class of surreal numbers using a tree-theoretic definition [8]. Both approaches have their advantages and disadvantages, with the former allowing for the free selection of equivalence class representatives, while the latter ensures the uniqueness of elements. This means that formalization of certain proofs is more intricate and challenging in one approach compared to the other.

In this paper, we define a bridge that allows us to combine formal proofs about surreal number in the general sense with the tree-theoretic proofs. This allows us to efficiently formalize a large number of properties of surreal numbers. In particular:

- We propose an easier approach to constructing the general surreal numbers where transfinite induction-recursion is replaced by transfinite induction.
- We propose an approach to conveniently combine the general approach proofs with tree-theoretic approach using global choice.
- We show that this is a convenient representation by proving that the surreals form a field including the square root.
- We show that reals, ordinals, and powers of ω are subsets of the surreals.
- We combine the Conway Normal Form proof skeleton [5] with Erlich's generalization of Conway's theory of surreal numbers [8] expressed in NBG set theory to prove it in Tarski-Grothendieck set theory [4].
- We present the details of the formalization in the Tarski-Grothendieck set theory formalized in the Mizar proof system. The same approach can be used to work effectively with surreal numbers in systems that only support transfinite induction, e.g. Isabelle/ZF [18] and MetaMath [15].
- We proved a large number of surreal number properties needed for the above results. This amounts to 335 proved top-level Mizar theorems totaling 1099 KB. The parts of the formalization corresponding to the proofs that surreals form a ring is already in the Mizar library [19, 20, 21].

To our knowledge, this is the most in-depth formalization of surreal numbers today.

2 Mizar

The Mizar proof system operates within the framework of classical first-order logic, augmented with limited second-order schemes requiring explicit instantiation by users [3, 10]. Within that logic, Mizar introduces the axioms of Tarski-Grothendieck set theory [2], which extends ZFC by incorporating Tarski's Axiom A. This axiom implies the Axiom of Choice (AC) [4] and enables the existence of arbitrarily large strongly inaccessible cardinals, thus providing models of ZFC and circumventing the necessity for proper classes in certain formalizations.

Unlike traditional foundational type systems, Mizar treats types as first-order predicates, supplemented with automation implemented through user-programmable Horn clauses. These clauses facilitate the propagation of various properties, known as adjectives, throughout Mizar terms in a bottom-up fashion [12]. The fact that an object X satisfies the type predicate t is written $x \text{ be } t$. Mizar adopts a Jaśkowski-style natural deduction approach, complemented by a fast and type-aware refutational first-order prover with several extensions [9] known as the Mizar obvious inference **by**.

The Mizar system is accompanied by the Mizar Mathematical Library (MML), a large corpus of formal mathematics, that among many other topics includes formalizations of reals and ordinals that we will use in this work.

Mizar allows the definition of several kinds of meta-level objects. We briefly revisit the definition of meta-level functions by **means**, as we will utilize them several times throughout the paper. This mechanism enables the definition of a function based on a predicate. The predicate can then reference the object being defined using the special keyword **it** within the definition body. This methodology closely resembles defining functions using the choice operator found in other proof systems. However, unlike the direct use of the choice operator, function definitions by **means** in Mizar permit the specification of additional conditions and require the explicit declaration of the result type. Two key proof obligations accompany such definitions: the proof of existence and the proof of uniqueness of the function's result. We illustrate the syntax of function definitions by **means** in Mizar through an example. For two sets X and Y of type **set**, the MML defines a meta-level function (keyword **func**) union, denoted as $X \cup Y$. This function returns a set (return type indicated after the \rightarrow keyword). The semantics of the function (following the **means** keyword) are given by the predicate stating that elements belong to the union if they are in any of its arguments. While the Mizar input syntax for universal and existential quantifiers is **for** and **ex**, respectively, we present them using more standard quantifier symbols in this paper:

```
let X, Y be set;
func X ∪ Y → set means
  ∀x be set. x ∈ it ⇔ (x ∈ X ∨ x ∈ Y);
```

Each meta-function definition requires showing that the defined object exists and is unique. For the details of these proofs see the formalization.

Meta-level predicates and types (as mentioned above types are just predicates) are defined in an analogous way, with the only exception that the keyword **func** is replaced by **pred** and **attr** respectively.

One of the restrictions we will consider in this paper is that, in Mizar (as well as in similar systems based on set theory), everything is considered to be a set, in accordance with Tarski's first axiom. Additionally, any set is an element of a set in the von Neumann hierarchy of sets. However, this does not imply that reasoning about certain classes is impossible, as meta-level functions and predicates (in Mizar also attributes/types) are not sets. This means that, according to the grammar, x **is set** can only be written when x is a term. Consequently, we can define a type such as **Ordinal** even if there is no set of ordinals. Similarly, we can define meta-level functions on a type whose elements form a proper class, for example, a successor function of the type **Ordinal** \rightarrow **Ordinal**. In Mizar (as in Isabelle/ZF or Megalodon), we can even quantify over such meta-level functions and predicates using second-order logic, but only with the universal quantifier.

3 Introduction to Surreal Numbers

Conway introduced the surreal numbers using two interleaving definitions: The way to build a new surreal number relies on two sets of surreal numbers, for which appropriate ordering constraints hold. And the way to check if two numbers are related in the order relies on checking the relation for the underlying sets of numbers. More precisely:

Concept: If L, R are any two sets of numbers, and no member of R is \leq any member of L , then there is a number $\{L \mid R\}$. All numbers are constructed in this way.

Comparison: If $x = \{L \mid R\}$, $x' = \{L' \mid R'\}$, then $x \leq x'$ if and only if $x' \not\leq$ any member of L and no member of R is $\leq x'$.

We introduce several notations that allow describing the various properties and proofs more concisely. We write $L \ll R$ iff for each $x \in L$ and $y \in R$, $y \not\leq x$. We introduce the relation $x \approx y$ to denote $x \leq y \wedge y \leq x$. It is easy to see that it is an equivalence relation.

Note, that several works quoted in the introduction Sec. 1 use the same symbol for identity $=$ and equivalence \approx of surreal numbers. As we aim to formalize these using interactive proof systems we will precisely separate the two. We also follow Conway's original notation $\{L \mid R\}$ for pairs $\langle L, R \rangle$. Let $x = \{L \mid R\}$ a surreal number, we use L_x and R_x to refer to the left L and the right component R of x , respectively.

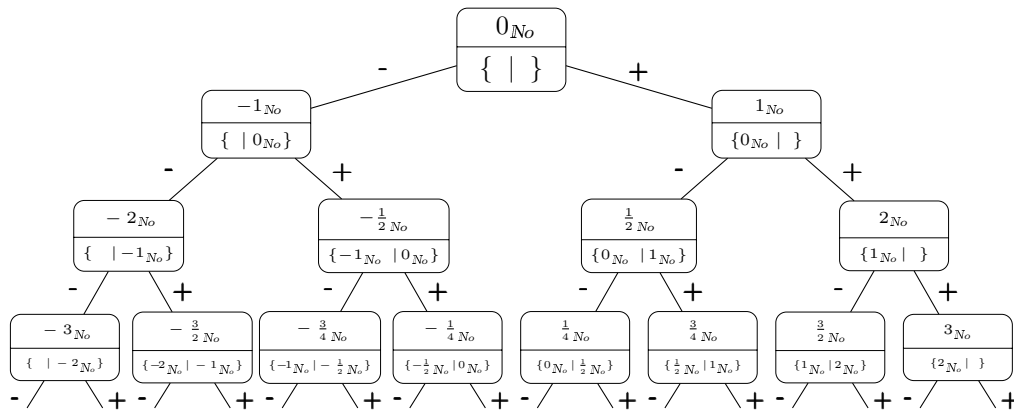
Conway constructs surreal numbers in so-called days indexed by ordinals. He starts by defining the first number, denoted 0_{No} , as the pair $\{ \mid \}$ ($:= \langle \emptyset, \emptyset \rangle$). Note that the relation $\emptyset \ll \emptyset$ obviously holds. This number is then used to initialize day zero as the only number present at this stage. In the next iteration, Day 1, we could consider three more pairs $-1_{No} := \{ \mid 0_{No} \}$, $1_{No} := \{ 0_{No} \mid \}$, and $\{ 0_{No} \mid 0_{No} \}$. The last one of those is not a number since already in Day 0 we can prove that $0_{No} \leq 0_{No}$, and relations between numbers are preserved across days. Generally, Day α is defined by all surreal numbers x for which $L_x, R_x \subseteq \bigcup_{\beta < \alpha} \text{Day } \beta$ and $L_x \ll R_x$. Additionally, we introduce the concept of the birthday of a surreal number x , denoted by $\mathfrak{b}x$, i.e, the smallest ordinal α such that $x \in \text{Day } \alpha$.

To show the main differences between Conway's approach and the tree-theoretic one, that we address in our contribution, we present the construction of Day 2. When generating the numbers present in Day 2, we can place any of the numbers already in Day 1 (i.e., -1_{No} , 0_{No} , 1_{No}) in the left and in the right set. This gives $(2^3)^2 = 64$ candidates for new numbers x . Only 20 of these numbers satisfy the criterion $L_x \ll R_x$. Note, that checking this criterion requires knowing the ordering on all the numbers in the preceding Day 1. In general, this number grows exponentially, that is given n different numbers, there are $(n+2)2^{n-1}$ new ones that satisfy the comparison criterion. These 20 numbers are different, however, not all are different in the quotient \approx . There are, in fact, only 4 new numbers, namely -2_{No} , $-\frac{1}{2}_{No}$, $\frac{1}{2}_{No}$, 2_{No} (see Fig. 1). This is because some of the newly generated numbers are equivalent in the \approx sense to each other, e.g., $\frac{1}{2}_{No} := \{ 0_{No} \mid 1_{No} \} \approx \{ -1_{No}, 0_{No} \mid 1_{No} \}$, and some are equivalent to already existing ones, e.g., $0_{No} = \{ \mid \} \approx \{ -1_{No} \mid \} \approx \{ \mid 1_{No} \} \approx \{ -1_{No} \mid 1_{No} \}$. More generally, in Day n for any $n \in \mathbb{N}$ there are $2n$ new numbers.

We next need to characterize the comparison relation on the new numbers. To do this for the new numbers in Day α , it is not sufficient to directly (without recursion) use the order for the previous days Day β for $\beta < \alpha$. As the tree has depth α , we need to perform up to α steps of recursion in order to use the information from previous days. Indeed, to justify that $\{ -1_{No} \mid 1_{No} \} \leq \{ 0_{No} \mid 1_{No} \}$ (compare with $0_{No} \leq \frac{1}{2}_{No}$) we need to show $-1_{No} \leq \{ 0_{No} \mid 1_{No} \} \wedge \{ -1_{No} \mid 1_{No} \} \leq 1_{No}$, and that happens because $-1_{No} \leq 1_{No}$ which we know from Day 1.

For any surreal number x in Fig. 1, we can always construct two new numbers in the \approx sense using $x_1 = \{ L_x \cup \{x\} \mid R_x \}$ and $x_2 = \{ L_x \mid R_x \cup \{x\} \}$ to represent them. Naturally, if x is youngest w.r.t. \approx , then $\mathfrak{b}x_1 = \mathfrak{b}x_2 = 1 + \mathfrak{b}x$. Note, that the new numbers in Day α , where α is a limit ordinal, cannot have a direct predecessor. They are created as cuts, similar to Dedekind reals. Nevertheless, if $\mathfrak{b}x$ is not a limit ordinal and x is youngest w.r.t. \approx , we can construct y that corresponds to the direct predecessor of x in the \approx sense, for which $x \approx \{ L_y \cup \{y\} \mid R_y \}$ or $x \approx \{ L_y \mid R_y \cup \{y\} \}$. This allows interpreting the equivalence classes of the \approx relation, as the class of all possible maps from an ordinal (including limit ordinals) to the set $\{+, -\}$, where $+$ and $-$ correspond to these two alternatives. This is the foundation of the tree-theoretic approach. As ordinals can be thought of as sequences, we can use the standard lexicographic order, with $- \prec \text{undefined} \prec +$. As the sequences are of different length for different ordinals, the *undefined* come up outside of the domain of the maps: If x is a subsequence of length α of the sequence y then the first index where they differ is $\alpha + 1$. At that index $x(\alpha + 1)$ is undefined while $y(\alpha + 1)$ has a value.

In the tree-theoretic approach, the comparison is defined as below. We will not analyse it in our work, but it helps compare the approaches.



■ **Figure 1** The relations between the numbers created in the first four days. The bottom part of each node give a representative of the equivalence class w.r.t. the \approx relation. The edge labels $+$ and $-$ correspond to the tree-theoretic interpretation of surreal numbers.

► **Definition 1** (tree-theoretic comparison). *Let x, y be maps from an ordinal to $\{+, -\}$, that represent surreal numbers in the tree-theoretic approach. Suppose that $x \neq y$ and α is the smallest ordinal where $x(\alpha) \neq y(\alpha)$. Then $x < y$ if and only if*

$$(x(\alpha) = - \wedge y(\alpha) \text{ is undefined}) \vee (x(\alpha) = - \wedge y(\alpha) = +) \vee (x(\alpha) \text{ is undefined} \wedge y(\alpha) = +). \quad (1)$$

The construction of surreal numbers in this approach is significantly easier to formalize [16]. We can even define the negation operator $-x$ by exchanging $+$ and $-$ in any map x . However, the construction of the remaining field operations becomes much more involved. In order to define $x \star y$, it is needed to build a kind of bridge to the Conway approach. This starts with some representation \bar{x}, \bar{y} , for which we define $\bar{x} \star \bar{y}$ using the Conway method. Finally, one needs to show the existence of a map for $x \star y$, rather than use direct recursion. A similar approach is required to prove all the field properties. This is much more involved than in the Conway approach, where we can freely represent numbers by their different representatives in their \approx class.

4 Formal Set-theoretic Construction of Surreal Numbers

In the previous Section, we pointed out that the surreal numbers are not a set and their ordering relation \leq cannot be a set. However, the restriction of this relation to any particular day is a set. To work with such sets, we will index the order α using a relation Ord that is a set. The notation $x \leq_{Ord} y$ simply means that $\langle x, y \rangle \in Ord$ and $L \ll_{Ord} R$ means $\forall l \in L. \forall r \in R. \langle r, l \rangle \notin Ord$. Remember, that constructing the surreal numbers in Day 2 proceeded in two steps: First the candidates were selected using the ordering on Day 1 surreals; subsequently the order in Day 2 was computed. In the construction, as well as in the uniqueness proof, we need to modify the Ord relations with a given set of candidates to construct Day α . For this, we define the sets of pairs $Games_\alpha$ for any ordinal α as follows (\mathcal{P} stands for powerset):

$$Games_\alpha = \mathcal{P} \left(\bigcup_{\beta < \alpha} Games_\beta \right) \times \mathcal{P} \left(\bigcup_{\beta < \alpha} Games_\beta \right). \quad (2)$$

29:6 Conway Normal Form for the Formalization of Surreal Numbers

Clearly, $\text{Games } 0 = \{\emptyset\} \times \{\emptyset\} = \{\langle \emptyset, \emptyset \rangle\}$ and $\text{Day } \alpha \subseteq \text{Games } \alpha$. Now we can define $\text{Day}_{\text{Ord}} \alpha$ even if Ord does not satisfy the *Comparison* condition as follows:

$$\text{Day}_{\text{Ord}} \alpha = \{x \in \text{Games } \alpha \mid L_x \subseteq \bigcup_{\beta < \alpha} \text{Day}_{\text{Ord}} \beta \wedge R_x \subseteq \bigcup_{\beta < \alpha} \text{Day}_{\text{Ord}} \beta \wedge L_x \ll_{\text{Ord}} R_x\}. \quad (3)$$

The recursive definition relies on a very complicated recursion scheme that combines unions over all previous ordinals. As this is very hard to express in several systems, including Mizar, we use a helper sequence S .

► **Definition 2** ($\text{Day}_{\text{Ord}} \alpha$). *Let Ord be relation, α be an ordinal and a α -length sequence S that satisfies:*

$$S(\beta) = \{x \in \text{Games } \beta \mid L_x \subseteq \bigcup_{\gamma < \beta} S(\gamma) \wedge R_x \subseteq \bigcup_{\gamma < \beta} S(\gamma) \wedge L_x \ll_{\text{Ord}} R_x\} \quad (4)$$

for any ordinal $\beta \leq \alpha$. Then $\text{Day}_{\text{Ord}} \alpha = S(\alpha)$.

We give the formal definition of $\text{Day}_{\text{Ord}} \alpha$ in Mizar and explain several used concepts below:

```
let  $\alpha$  be Ordinal,  $\text{Ord}$  be Relation;
func Day( $\text{Ord}, \alpha$ ) → Subset of Games  $\alpha$  means
   $\exists S$  be Sequence. it =  $S.\alpha \wedge \text{dom } S = \text{succ } \alpha \wedge (\forall \beta$  be Ordinal.  $\beta \in \text{succ } \alpha \Rightarrow$ 
     $S.\beta = \{x \text{ where } x \text{ is Element of Games } \beta: L_x \subseteq \text{union rng } (S|_{\beta}) \wedge R_x \subseteq \text{union rng } (S|_{\beta}) \wedge L_x \ll_{\text{Ord}} R_x\});$ 
```

- The length of S is α (equivalently $\text{dom } S = \text{succ } \alpha$),
- $S.\beta$ is the set-theoretic function application, corresponding to $\text{Day}_{\text{Ord}} \beta$ for $\beta \in \text{succ } \alpha$ ($\beta < \alpha$).
- $\text{union rng } (S|_{\beta})$ is the union of the values of the sequence S restricted to the ordinal β which corresponds to $\bigcup_{\gamma < \beta} \text{Day}_{\text{Ord}} \gamma$,
- it refers to the defined object, equal to $S.\alpha$ which also is a subset of $\text{Games } \alpha$.

The definition implies that $\text{Day}_{\text{Ord}} \alpha \subseteq \text{Day}_{\text{Ord}} \beta$ if $\alpha \leq \beta$, but it is also possible to use transfinite induction to show a “monotonicity”-like property:

► **Lemma 3.** *Let Ord be a relation, α be an ordinal and $x \in \text{Games } \alpha$ such that $x \notin \text{Day}_{\text{Ord}} \alpha$. Then for all ordinals β , $x \notin \text{Day}_{\text{Ord}} \beta$.*

We also restrict the concept of birthday of a surreal number x to a relation Ord .

► **Definition 4.** *Let Ord be relation, x be element of $\text{Day}_{\text{Ord}} \beta$ for some ordinal β . Then the birthday of a object x with respect to a relation Ord is an ordinal α that satisfies two conditions:*

- $o \in \text{Day}_{\text{Ord}} \alpha$,
- $\forall \beta. (x \in \text{Day}_{\text{Ord}} \beta \rightarrow \alpha \leq \beta)$.

```
assume  $\exists \beta$  be Ordinal.  $x \in \text{Day}(\text{Ord}, \beta)$ ;
func born( $\text{Ord}, x$ ) → Ordinal means
   $x \in \text{Day}(\text{Ord}, \text{it}) \wedge (\forall \beta$  be Ordinal.  $x \in \text{Day}(\text{Ord}, \beta) \Rightarrow \text{it} \subseteq \beta)$ ;
```

Even if we construct the Ord relation that satisfies the *Comparison* condition, we cannot use that single relation for all days. Indeed, each day $\text{Day}_{\text{Ord}} \alpha$ corresponds to at least one new number $\{\bigcup_{\beta < \alpha} \text{Day}_{\text{Ord}} \beta \mid\}$ and $\langle 0_{\text{No}}, \{\bigcup_{\beta < \alpha} \text{Day}_{\text{Ord}} \beta \mid\} \rangle \in \text{Ord}$. As such, we can only assume that Ord satisfies the *Comparison* condition on some set. At this point, we could already talk about the surreal numbers, but only until a certain birthday, after which we would have to modify the relation.

Conway, uses a special induction over n -tuples of arguments (referred to as Conway's induction) when constructing the surreal numbers, their order, the operations, as well as when proving their properties. The induction is similar to \in -induction, where the truth of $P(x_1, x_2, \dots, x_n)$ follows from the truth of P for all modified tuples x_1, x_2, \dots, x_n where at least one x_i is replaced by its left or right component. Unfortunately, some proofs require changing the order of the elements in a sequence with additional properties. This has already been observed by Mamane [14]: In his Coq formalization he refers to such induction arguments as *permuting inductions*. Such an induction could be expressed as a transfinite induction over sums of \mathbf{b} applied to these arguments, however, standard ordinal sum is not symmetric nor monotonous on its both arguments.

In our formalization, we tackle this problem by using the natural Hessenberg sum of ordinals. This is a variant of ordinal sum that is symmetric and monotonic on both arguments. In fact, Hessenberg invented his ordinal sum, inspired by surreal numbers and the use of Cantor Normal Form¹, but it became a concept in mathematics in general and has already been formalized [13]. In our work, wherever possible, we use subsets of Cartesian product in the construction, but full Hessenberg sum is necessary for example in the definitions of the arithmetic operation.

► **Definition 5** (Prod^C and Prod^O). *Let Ord be a relation and α, β be ordinals. Then we define two subsets of the Cartesian product $\text{Day}_{\text{Ord}}\alpha \times \text{Day}_{\text{Ord}}\alpha$ as follows:*

$$\text{Prod}_{\text{Ord}}^C(\alpha, \beta) = \{\langle x, y \rangle \mid x, y \in \text{Day}_{\text{Ord}}\alpha \wedge ((\mathbf{b}_{\text{Ord}} x < \alpha \wedge \mathbf{b}_{\text{Ord}} y < \alpha) \vee (\mathbf{b}_{\text{Ord}} x = \alpha \wedge \mathbf{b}_{\text{Ord}} y \leq \beta) \vee (\mathbf{b}_{\text{Ord}} x \leq \alpha \wedge \mathbf{b}_{\text{Ord}} y = \beta))\} \quad (5)$$

$$\text{Prod}_{\text{Ord}}^O(\alpha, \beta) = \{\langle x, y \rangle \mid x, y \in \text{Day}_{\text{Ord}}\alpha \wedge ((\mathbf{b}_{\text{Ord}} x < \alpha \wedge \mathbf{b}_{\text{Ord}} y < \alpha) \vee (\mathbf{b}_{\text{Ord}} x = \alpha \wedge \mathbf{b}_{\text{Ord}} y < \beta) \vee (\mathbf{b}_{\text{Ord}} x < \alpha \wedge \mathbf{b}_{\text{Ord}} y = \beta))\} \quad (6)$$

The O and C superscripts are used, since the concepts are somewhat similar to open and closed intervals respectively. Observe two properties of Prod^C and Prod^O : $\text{Prod}_{\text{Ord}}^C(\alpha, \alpha) = \text{Day}_{\text{Ord}}\alpha \times \text{Day}_{\text{Ord}}\alpha$ and $\bigcup_{\gamma < \beta} \text{Prod}_{\text{Ord}}^C(\alpha, \gamma) = \text{Prod}_{\text{Ord}}^O(\alpha, \beta)$.

As we already discussed, the order relation on surreals \leq is too big to be a set, so reasoning about it is complicated. For this reason, we will consider its subsets that are sets. A restriction of the \leq relation to any set, will be a subset of $\text{Day}\alpha \times \text{Day}\alpha$ for some α , so we introduce a natural restriction:

► **Definition 6** (Almost No-order). *A relation Ord is an almost No-order if $\text{Ord} \subseteq \text{Day}_{\text{Ord}}\alpha \times \text{Day}_{\text{Ord}}\alpha$ for some ordinal α .*

► **Definition 7.** *Let A be a set. A relation Ord preserves the Comparison condition on A (written $\text{Comp}(\text{Ord}, A)$) if and only if*

$$\forall x. \forall y. x \leq_{\text{Ord}} y \Leftrightarrow L_x \ll_{\text{Ord}} \{y\} \wedge \{x\} \ll_{\text{Ord}} \{y\}. \quad (7)$$

One of the crucial properties of our formalization is that we can complete the proofs using only (transfinite) induction, without requiring any complex techniques available only in selected systems (e.g. we do not use induction-recursion or complicated recursion schemes). Apart from the construction we here show the first proof done this way in full. The following Theorem 8 gives a form of uniqueness of the order on the surreal numbers, uses two applications of (transfinite) induction. Many proofs in our formalization use two inductions in a similar way.

¹ Cantor Normal Form is the unique representation of any ordinal x as $n_1\omega^{\alpha_1} + n_2\omega^{\alpha_2} + \dots + n_k\omega^{\alpha_k}$ for some $k \in \mathbb{N}$, where $\{n_i\}_{i=0}^k$ is a sequence of positive naturals and $\{\alpha_i\}_{i=0}^k$ a decreasing ordinal sequence.

► **Theorem 8.** *Let R, S be relation. The following facts hold.*

1. *if $R \cap \bigcup_{\gamma < \alpha} \text{Games } \gamma = S \cap \bigcup_{\gamma < \alpha} \text{Games } \gamma$ where α is any ordinal, then:*
 - $\text{Day}(R, \alpha) = \text{Day}(S, \alpha)$,
 - $\forall x \in \text{Day}(R, \alpha). \mathfrak{b}_R(a) = \mathfrak{b}_S(a)$,
 - $\forall \beta. R \cap \text{Prod}_R^O(\alpha, \beta) = S \cap \text{Prod}_S^O(\alpha, \beta) \wedge R \cap \text{Prod}_R^C(\alpha, \beta) = S \cap \text{Prod}_S^C(\alpha, \beta)$.
2. *if R, S are almost No-order, $\text{Comp}(R, \text{Prod}_R^C(\alpha, \beta))$, $\text{Comp}(S, \text{Prod}_S^C(\alpha, \beta))$ then $R \cap \text{Prod}_R^C(\alpha, \beta) = S \cap \text{Prod}_S^C(\alpha, \beta)$.*

Proof. Fact 1 is proved by transfinite induction. Let $x = \{L_x \mid R_x\} \in \text{Day}_R \alpha$. Every element of $L_x \cup R_x$ is a member of $\text{Day}_R \beta$ for some $\beta < \alpha$, so by induction hypothesis it is also a member of $\text{Day}_S \beta$. Similarly, $L_x \ll_S R_x$ (equivalently $\forall l \in L_x. \forall r \in R_x. l \not\leq_S r$) is a consequence of $L_x \ll_R R_x$ and $R \cap \bigcup_{\gamma < \alpha} \text{Games } \gamma = S \cap \bigcup_{\gamma < \alpha} \text{Games } \gamma$, therefore $x \in \text{Day}_S \alpha$. Since $\forall \beta \leq \alpha. \text{Day}_R \beta = \text{Day}_S \beta$ the remaining part of Fact 1 is straightforward.

To show Fact 2, first consider the following subclaim. If R, S are almost No-orders, then

$$\begin{aligned} \forall \alpha. \forall \beta. \beta \leq \alpha \wedge \text{Comp}(R, \text{Prod}_R^C(\alpha, \beta)) \wedge \text{Comp}(S, \text{Prod}_S^C(\alpha, \beta)) \wedge \\ R \cap \text{Prod}_R^O(\alpha, \beta) = S \cap \text{Prod}_S^O(\alpha, \beta) \Rightarrow \\ R \cap (\text{Prod}_R^C(\alpha, \beta) \setminus \text{Prod}_R^O(\alpha, \beta)) \subseteq S \cap (\text{Prod}_S^C(\alpha, \beta) \setminus \text{Prod}_S^O(\alpha, \beta)) \quad (8) \end{aligned}$$

Let β, α such that $\beta \leq \alpha$, $\text{Comp}(R, \text{Prod}_R^C(\alpha, \beta))$, $\text{Comp}(S, \text{Prod}_S^C(\alpha, \beta))$ and

$$R \cap \text{Prod}_R^O(\alpha, \beta) = S \cap \text{Prod}_S^O(\alpha, \beta). \quad (9)$$

Let x, y such that $\langle x, y \rangle \in R \cap (\text{Prod}_R^C(\alpha, \beta) \setminus \text{Prod}_R^O(\alpha, \beta))$. Then, by definition 5, there are only two possible cases: $\mathfrak{b}_R x = \alpha \wedge \mathfrak{b}_R y = \beta$ or $\mathfrak{b}_R x = \beta \wedge \mathfrak{b}_R y = \alpha$. Without loss of generality we assume that $\mathfrak{b}_R x = \alpha \wedge \mathfrak{b}_R y = \beta$. By Lemma 3 we know that $R \cap \bigcup_{\gamma < \alpha} \text{Games } \gamma = S \cap \bigcup_{\gamma < \alpha} \text{Games } \gamma$, hence $\langle x, y \rangle \in \text{Prod}_S^C(\alpha, \beta) \setminus \text{Prod}_S^O(\alpha, \beta)$, $\mathfrak{b}_S x = \alpha$, $\mathfrak{b}_S y = \beta$. It remains to prove $\langle x, y \rangle \in S$. Since $\text{Comp}(S, \text{Prod}_S^C(\alpha, \beta))$, this is equivalent to $L_x \ll_S \{y\} \wedge \{x\} \ll_S R_y$.

To prove the first conjunct, suppose contrary to our claim, that $y \leq_S l$ for some $l \in L_x$. Then either $\beta < \alpha$ or $\beta = \alpha$. In both cases we get $\langle y, l \rangle \in S \cap \text{Prod}_S^O(\alpha, \beta)$ since $\mathfrak{b}_S l < \mathfrak{b}_S x$. Hence $y \leq_R l$ by (9) contrary to $L_x \ll_R \{y\}$ (by $\langle x, y \rangle \in R$). To show the second conjunct, $\{x\} \ll_S R_y$, suppose contrary to our claim, that $r \leq_S x$ for some $r \in R_y$. Then $\mathfrak{b}_S r < \mathfrak{b}_S y$, $\langle r, x \rangle \in S \cap \text{Prod}_S^O(\alpha, \beta)$, and finally $r \leq_R x$ by (9) contradicting $x \ll_R \{y\}$ (by $\langle x, y \rangle \in R$).

We can now easily infer $R \cap \text{Prod}_R^C(\alpha, \beta) = S \cap \text{Prod}_S^C(\alpha, \beta)$ from $R \cap \text{Prod}_R^O(\alpha, \beta) = S \cap \text{Prod}_S^O(\alpha, \beta)$. Next, using transfinite induction we can simplify the assumption, obtaining

$$\begin{aligned} \forall \alpha. \forall \beta. \beta \leq \alpha \wedge \text{Comp}(R, \text{Prod}_R^C(\alpha, \beta)) \wedge \text{Comp}(S, \text{Prod}_S^C(\alpha, \beta)) \wedge \\ R \cap \text{Prod}_R^O(\alpha, 0) = S \cap \text{Prod}_S^O(\alpha, 0) \Rightarrow R \cap \text{Prod}_R^C(\alpha, \beta) = S \cap \text{Prod}_S^C(\alpha, \beta) \quad (10) \end{aligned}$$

with the help of the equation $\bigcup_{\gamma < \beta} \text{Prod}_{Ord}^C(\alpha, \gamma) = \text{Prod}_{Ord}^O(\alpha, \beta)$. A second use of transfinite induction together with the equation $\bigcup_{\beta < \alpha} \text{Prod}_{Ord}^C(\beta, \beta) = \text{Prod}_{Ord}^O(\alpha, 0)$ completes the proof of fact 2. ◀

The above Theorem 8 allows defining the order using its selected properties. We only show the main step in the construction, the remaining part are two applications of transfinite induction, analogously to what we did in the proof of the second part of the lemma 8.

► **Theorem 9.** *Let α, β ordinals, R be relation such that $\text{Comp}(R, \text{Prod}_R^O(\alpha, \beta))$ and $R \subseteq \text{Prod}_R^O(\alpha, \beta)$. Then*

$$S := R \cup \{\langle x, y \rangle \mid x, y \in \text{Day}_R \alpha \wedge ((\mathfrak{b}_R x = \alpha \wedge \mathfrak{b}_R y = \beta) \vee (\mathfrak{b}_R x = \beta \wedge \mathfrak{b}_R y = \alpha)) \wedge L_x \ll_R \{y\} \wedge \{x\} \ll_R R_y\} \quad (11)$$

satisfies $\text{Comp}(S, \text{Prod}_S^C(\alpha, \beta))$ and $S \subseteq \text{Prod}_S^C(\alpha, \beta)$.

We can now define the order relation. Proving that it is a function (its existence and uniqueness) relies on the properties of $\text{Comp}(\text{Ord}, \text{Prod}_{\text{Ord}}^C(\alpha, \alpha))$ and $\text{Ord} \subseteq \text{Prod}_{\text{Ord}}^C(\alpha, \alpha)$:

```
let  $\alpha$  be Ordinal;
func  $\text{No}^\alpha \rightarrow \text{Relation}$  means :: SURREAL0: def 12
  it preserves  $\_No\_Comparison\_on$  [:Day(it, $\alpha$ ),Day(it, $\alpha$ ):]  $\wedge$  it  $\subseteq$  [:Day(it, $\alpha$ ),Day(it, $\alpha$ ):];
```

For a given α , this relation is still a set with all usual restrictions of sets. However, in the formalization we can consider a different relation for each particular day. We define the type surreal as the members of at least one day of the form $\text{Day}_{\text{No}^\alpha} \alpha$. Similarly, we can define the order $x \leq y$ (as a predicate and not a set-theoretic relation) as true when $\langle x, y \rangle$ is an element of at least one No^α as follows:

```
let  $x$  be object;
attr  $x$  is surreal means
   $\exists \alpha$  be Ordinal.  $x \in \text{Day}_{\text{No}^\alpha} \alpha$ ;

let  $x, y$  be surreal object;
pred  $x \leq y$  means
   $\exists \alpha$  be Ordinal.  $x \leq_{\text{No}^\alpha} y$ ;
```

This gives us the Concept and Comparison properties, that encapsulate the constructed surreals. With just the help of standard (transfinite) induction, we created a theoretical heaven, where the properties of Conway numbers are satisfied. In the next section, we will also introduce the canonical representation [8] that links this to the tree-theoretic approach.

5 The Surreal Numbers as a Field

For a surreal number x , Conway introduces the somewhat confusing concept of a *typical member* of L_x and R_x denoted by x^L and x^R , respectively. We will use these only to define how functions affect the left and right parts of a surreal number. More formally $f(x^L)$ will denote $\{f(y) \mid y \in L_x\}$ (analogous for x^R in R_x).²

With this notation, the unary negation can be simply written as $-x = \{-x^R \mid -x^L\}$ and unfolds to the full $-x = \{\{-x_r \mid x_r \in R_x\} \mid \{-x_l \mid x_l \in L_x\}\}$. We write the definitions of the remaining operations only using the typical member notation: $x + y = \{x^L + y, x + y^L \mid x^R + y, x + y^R\}$, $x \cdot y = \{x^L \cdot y + x \cdot y^L - x^L \cdot y^L, x^R \cdot y + x \cdot y^R - x^R \cdot y^R \mid x^L \cdot y + x \cdot y^R - x^L \cdot y^R, x^R \cdot y + x \cdot y^L - x^R \cdot y^L\}$ where the comma corresponds to the different possible ways how elements are constructed, formally corresponding to a union.

The definition of any arithmetic operation on the surreal numbers would require some complicated recursion principle, a different one for each operation. To avoid this, all arithmetic operations defined in our formalization are introduced in three phases. To define operation \star (such as $+$, $-$, \cdot , $^{-1}$) we first define \star_α on a specific domain of surreal numbers restricted to $\text{Day } \alpha$. We will denote this domain by X_α so that we can uniformly cover unary and binary operations. In particular, X_α will be a subset of surreal numbers for unary \star and a set of pairs for binary operations. Subsequently, we prove that the application of the operator \star_α on surreal arguments is also surreal. For this proof we recursively use the properties of \star_β for $\beta < \alpha$ as well as for \star_α . Finally, we can define \star as \star_α , where α is determined by the arguments given to \star .

² Conway also uses typical members in more ambiguous ways, i.e., $x = \{x^L \mid x^R\}$. We will avoid these in this paper.

29:10 Conway Normal Form for the Formalization of Surreal Numbers

The second step of each of the definitions is typically most involved and often requires proving several additional properties. For example, consider the proof that multiplication restricted to X_α preserves the surreal type (the definition of multiplication relies on addition, so this is of course done after addition is defined and its basic properties are proved). The formal goal is: $\forall x. \forall y. \langle x, y \rangle \in X_\alpha \Rightarrow x \cdot_\alpha y$ is surreal. For this, we only have to show that $x \cdot_\alpha y$ satisfies the *Concept* condition and (following Conway's work for \cdot) requires the following four properties of \cdot_β for any $\beta < \alpha$:

- $\forall x. \forall y. \langle x, y \rangle \in X_\beta \Rightarrow x \cdot_\beta y$ is surreal,
 - $\forall x. \forall y. \langle x, y \rangle \in X_\beta \Rightarrow x \cdot_\beta y = y \cdot_\beta x$,
 - $\forall x_1. \forall x_2. \forall y. \langle x_1, y \rangle \in X_\beta \wedge \langle x_2, y \rangle \in X_\beta \wedge x_1 \approx x_2 \Rightarrow x_1 \cdot_\beta y \approx x_2 \cdot_\beta y$,
 - $\forall x_1. \forall x_2. \forall y_1. \forall y_2. \dots \wedge x_1 < x_2 \wedge y_1 < y_2 \Rightarrow x_1 \cdot_\beta y_2 + x_2 \cdot_\beta y_1 < x_1 \cdot_\beta y_1 + x_2 \cdot_\beta y_2$ ³,
- necessary in the proof for \cdot_α (proof by induction over β of the conjunction of these properties).

In the final step of each definition, that is to define \star based on \star_α , we need to specify the X_α on which it is defined. For the unary operations $-$, $^{-1}$ (but also the square root $\sqrt{\cdot}$, unique element $\mathfrak{U}_{\text{niq}}$ for each class $[x]_{\approx}$ introduced in Section 6, and $\omega \cdot$ in Section 7) we can define \star_α on the set $X_\alpha := \text{Day } \alpha$. For binary operations, we introduce $X_\alpha := \Delta^\alpha$, i.e., the set of pairs $\langle x, y \rangle$ where $\mathfrak{b}x \oplus \mathfrak{b}y \leq \alpha$ and \oplus is the natural Hessenberg sum of ordinals. The ordinals are not a set, so we use the same trick as in Definition 2, namely a monotonously increasing sequence of functions (they need to be monotonous, that is only add new pairs to the set-theoretic functions in order to preserve consistency). The sequence $\{\text{Day } \alpha\}$ is naturally increasing, and $\{\Delta^\alpha\}$ is monotonously increasing since \oplus is monotonous. With this, we can show that the sequence of functions \star_α is increasing w.r.t. set inclusion. We call this property \subseteq -monotone. More precisely, a sequence of functions S is \subseteq -monotone iff $\forall \alpha \in \text{domain}(S). \forall \beta \in \text{domain}(S). \beta \leq \alpha \Rightarrow S(\beta) \subseteq S(\alpha)$. As a consequence $\bigcup_{\beta < \alpha} \star_\beta$ is a function defined on $\bigcup_{\beta < \alpha} X_\beta$. It already has the necessary properties, but we still have to show that the results of the function are surreal numbers. The above shows the main stages of the proposed approach to defining functions on the surreal numbers. The initial construction does not know the properties of the results; subsequently these are proved; and finally we show that the result is surreal.

In order to efficiently proceed with the formalization, we prove a second-order theorem (referred to as a scheme in Mizar) that will allow us to efficiently define all these operations. The scheme constructs a sequence, where the definition has access to all previous elements.

► **Theorem 10.** *Let X be a function from ordinals that returns an arbitrary set, and H a binary function (the first argument is an arbitrary set but the second must be a \subseteq -monotone sequence of functions) such that*

- $\forall S$ be \subseteq -monotone sequence of functions. $(\forall \alpha \in \text{domain}(S). \text{domain}(S(\alpha)) = X(\alpha)) \Rightarrow (\forall \alpha \in \text{domain}(S). \forall x \in \text{domain}(S(\alpha)). H(x, S) = H(x, S|_\alpha))$,
- $\forall \alpha. \forall \beta. \beta \leq \alpha \Rightarrow X(\beta) \subseteq X(\alpha)$.

Then for every ordinal θ , there exists a unique \subseteq -monotone sequence S of functions of length θ where $\forall \alpha \in \text{domain}(S). \text{domain}(S(\alpha)) = X(\alpha) \wedge (\forall x \in X(\alpha). S(\alpha)(x) = H(x, S))$.

Recall, that $-x = \{-x^R \mid -x^L\}$, or more precisely $-_\alpha(x) = \{(\bigcup_{\beta < \alpha} -\beta) \setminus R_x \mid (\bigcup_{\beta < \alpha} -\beta) \setminus L_x\}$, where the image of a set is denoted by \setminus . Thus $-_\alpha(x)$ depends on x and $\bigcup_{\beta < \alpha} -\beta$. However, we define $-_\alpha(x)$ as $H(x, \{-\beta\}_{\beta \leq \alpha})$ to be able to access the whole sequence and not just its union. Additionally, the constraint $x \in X(\alpha) \Rightarrow H(x, S) = H(x, S|_\alpha)$ intuitively means that we expect $-_\alpha(x) = H(x, \{-\beta\}_{\beta \leq \alpha}) = H(x, \{-\beta\}_{\beta \leq \theta})$.

³ where \dots is the union of assumptions of the form $\langle x_i, y_j \rangle \in X_\beta$ for every occurrence of $x_i \cdot_\beta y_j$.

We show the details of the definition of $-x$ and $x+y$ using this theorem. For $-x$, we simply use $\theta = \mathfrak{b} x$, $X(\alpha) = \text{Day } \alpha$, $H(o, S) = \langle (\bigcup \text{rng } S) \setminus R_o, (\bigcup \text{rng } S) \setminus L_o \rangle$. Finally, define $-x$ to be $-\mathfrak{b} x$. To define addition $x+y$, we use $\theta = \mathfrak{b} x \oplus \mathfrak{b} y$, $X(\alpha) = \sqsupset^\alpha$, $H(o, S) = \langle (\bigcup \text{rng } S) \setminus ((L_{L_o} \times \{R_o\}) \cup (\{L_o\} \times L_{R_o})), (\bigcup \text{rng } S) \setminus ((R_{L_o} \times \{R_o\}) \cup (\{L_o\} \times R_{R_o})) \rangle$ and define $x+y$ as $x + \mathfrak{b} x \oplus \mathfrak{b} y$. To clarify the second case, notice that o as a member of the triangle operator is a pair $o = \langle a, b \rangle$ for some surreal a, b , so the expression H can be represented as $\langle + \setminus ((L_a \times \{b\}) \cup (\{a\} \times L_b)), + \setminus (R_a \times \{b\}) \cup (\{a\} \times R_b) \rangle$ equal $\{L_a + b, a + L_b \mid R_a + b, a + R_b\}$. This is well-defined and reduces to Conway's definition of $a + b$.

After defining all the operations, we show that their results are surreal numbers. Like Schleicher [22], we show the properties of the operations alongside the properties of the order. We covered all of Schleicher's chapter 3 [22], additionally making use of some more detailed proofs found in Grimm [11] and Tondering [23]. We finally formally show that \mathbb{No} has all the properties of an ordered field:

$$\begin{aligned} x + y &= y + x & x + 0_{\mathbb{No}} &= x & x \cdot (y + z) &\approx x \cdot y + x \cdot z \\ x \cdot y &= y \cdot x & x \cdot 1_{\mathbb{No}} &= x & (x \cdot y) \cdot z &\approx x \cdot (y \cdot z) \\ (x + y) + z &= x + (y + z) & x + (-x) &\approx 0_{\mathbb{No}} & x \not\approx 0_{\mathbb{No}} \rightarrow x \cdot x^{-1} &\approx 1_{\mathbb{No}} \end{aligned} \tag{12}$$

The inverse operation deserves special attention. Conway uses changing the equivalence class representative and hides a secondary recursion. Let $x > 0_{\mathbb{No}}$. Then $x \approx p := \{0_{\mathbb{No}}, x^L \mid x^R\}$ where x^L, x^R are restricted to positive typical members. More formally, p is constructed by eliminating the non-positive elements from L_x, R_y and adding $0_{\mathbb{No}}$ in its left component. Then Conway defines $y := x^{-1}$ using a strongly informal recursive property, that expresses the transitive closure of subsequent generations of typical members of y as follows:

$$y = \{0_{\mathbb{No}}, \frac{1_{\mathbb{No}} + (p^R - p)y^L}{p^R}, \frac{1_{\mathbb{No}} + (p^L - p)y^R}{p^L} \mid \frac{1_{\mathbb{No}} + (p^L - p)y^L}{p^L}, \frac{1_{\mathbb{No}} + (p^R - p)y^R}{p^R}\} \tag{13}$$

where p_L, p_R are reserved only for positive typical members⁴. We imitate Schleicher and Stoll's proof [22], formalizing using transitive closures of subsequent right and left closures of typical members. For details, see the `SURREALI.miz` formalization.

A somewhat similar approach for highly-recursive definitions has been used to define the square root. According to Conway [5], Clive Bach defined the square root of a non-negative number as follows:

$$\sqrt{x} = y = \{\sqrt{x^L}, \frac{x + y^L \cdot y^R}{y^L + y^R} \mid \sqrt{x^R}, \frac{x + y^L \cdot y^{L\bullet}}{y^L + y^{L\bullet}}, \frac{x + y^R \cdot y^{R\bullet}}{y^R + y^{R\bullet}}\} \tag{14}$$

where x^L, x^R are non-negative typical members of x , and $y^L, y^{L\bullet}, y^R, y^{R\bullet}$ are options of y chosen so that no one of the three denominators is zero. Conway leaves the correctness proof to the reader [5].

We formally define this in two steps. First, we designate the initial sets $\mathbf{L} := \{\sqrt{x^L} \mid 0_{\mathbb{No}} \leq x^L \in L_x\}$, $\mathbf{R} := \{\sqrt{x^R} \mid 0 \leq x^R \in R_x\}$, and then we perform a transitive closure. We focus on the second step. For the square root definition, we introduce a helper definition S .

⁴ To show the intricacy of the definition, consider $x = 5_{\mathbb{No}} \approx p := \{0_{\mathbb{No}}, 4_{\mathbb{No}} \mid \}$ where there is no p_R and $p_L = 4_{\mathbb{No}}$. Since $y = \{0_{\mathbb{No}}, \dots \mid \dots\}$ we start by $y_L = 0_{\mathbb{No}}$ and obtain a new $y_R = \frac{1_{\mathbb{No}} + (4_{\mathbb{No}} - 5_{\mathbb{No}}) \cdot 1_{\mathbb{No}}}{4_{\mathbb{No}}} = \frac{1}{4}_{\mathbb{No}}$. Then $\frac{1_{\mathbb{No}} + (4_{\mathbb{No}} - 5_{\mathbb{No}}) \cdot \frac{1}{4}_{\mathbb{No}}}{4_{\mathbb{No}}} = \frac{3}{16}_{\mathbb{No}}$ is a new y_L , and $\frac{1_{\mathbb{No}} + (4_{\mathbb{No}} - 5_{\mathbb{No}}) \cdot \frac{3}{16}_{\mathbb{No}}}{4_{\mathbb{No}}} = \frac{13}{64}_{\mathbb{No}}$ is a new y_R and so on, creating $y = \{0_{\mathbb{No}}, \frac{3}{16}_{\mathbb{No}}, \frac{51}{256}_{\mathbb{No}}, \dots \mid \frac{1}{4}_{\mathbb{No}}, \frac{13}{64}_{\mathbb{No}}, \frac{205}{1024}_{\mathbb{No}}, \dots\}$.

► **Definition 11** (Square root). *Let x be a surreal numbers, L, R be surreal number sets. We define $S(x, A, B)$ to be $\{\frac{x+a \cdot b}{a+b} \mid a \in A \wedge b \in B \wedge 0_{No} \not\approx a+b\}$. Consider the sequences $\{L_n\}_{n \in \mathbb{N}}$, $\{R_n\}_{n \in \mathbb{N}}$ defined as follows:*

$$L_0 = L, \quad R_0 = R, \quad L_{n+1} = L_n \cup S(x, L^n, R_n), \quad R_{n+1} = R_n \cup S(x, L_n, L_n) \cup S(x, R_n, R_n). \quad (15)$$

Then, we can define $\sqrt{x, L, R} := \{\bigcup_{n \in \mathbb{N}} L_n \mid \bigcup_{n \in \mathbb{N}} R_n\}$.

With the above definition, we can apply Theorem 10 using $\theta = \mathfrak{b} x$, $X(\alpha) = \text{Day } \alpha$,

$$H(o, S) = \sqrt{o, \{(\bigcup \text{rng } S)(o^L) \mid 0_{No} \leq o^L \in L_o\}, \{(\bigcup \text{rng } S)o^R \mid 0_{No} \leq o^R \in R_o\}} \quad (16)$$

to obtain the final function \sqrt{x} .

We have proved the usual properties of the square root, such as $\sqrt{x \cdot x} \approx x$ for non-negative surreals and $\sqrt{x^{-1}} \approx \sqrt{x}^{-1}$ for positive surreals. The definition proposed by Clive Bach can even be applied even to negative surreal numbers. As expected, it does not behave well for these: rather than give us the surcomplex numbers we instead show that different representatives of the same equivalence class of a negative number give different square roots. Indeed, consider an arbitrary positive x . Then $-1_{No} \approx y = \{ \mid (\sqrt{x \cdot x + 1_{No}} - x) \cdot (\sqrt{x \cdot x + 1_{No}} - x) \}$ but $\sqrt{-1_{No}} = -1_{No}$ and $\sqrt{y} < -x$. With this, for any negative number x we can construct a number $\approx -1_{No}$, whose square root is less than x .

6 Reals and Ordinals as Subsets of Surreal

Conway [5] showed that the real numbers are a subset of No without focusing on their construction. Starting with a construction of *integers (that include the *naturals) and inverse, an x would be called *real if $x \approx \{x - \frac{1_{No}}{n_{No}} \mid x + \frac{1_{No}}{n_{No}}\}_{0 < n}$ and $-k_{No} < x < k_{No}$ for some natural k . This was later restricted to dyadic numbers. Grimm [11] directly constructed the dyadic numbers and defined a bijection from *real surreal into real.

We formalize these constructions, additionally showing the set inclusions (similar to the MML's property $\mathbb{N} \subseteq \mathbb{Z} \subseteq \mathbb{Q} \subseteq \mathbb{R}$ but with dyadic numbers \mathbb{D} and ordinals):

$$s_{\mathbb{Z}}(i) = \begin{cases} 0_{No} & \text{if } i = 0, \\ \{s_{\mathbb{Z}}(i-1) \mid \} & \text{if } i > 0, \\ \{ \mid s_{\mathbb{Z}}(i+1) \} & \text{if } i < 0, \end{cases} \quad s_{\mathbb{D}}(d) = \begin{cases} s_{\mathbb{Z}}(d) & \text{if } d \in \mathbb{Z}, \\ \{s_{\mathbb{D}}(\frac{j}{2^p}) \mid s_{\mathbb{D}}(\frac{j+1}{2^p})\} & \text{if } d = \frac{2j+1}{2^{p+1}} \text{ for} \\ & \text{some } j \in \mathbb{Z}, p \in \mathbb{N}. \end{cases} \quad (17)$$

and construct $s_{\mathbb{R}}(r)$ which selects the \approx equivalence class representative of the number, such that $\{s_{\mathbb{D}}(\frac{\lceil r \cdot (2^n) - 1 \rceil}{2^n}) \mid s_{\mathbb{D}}(\frac{\lfloor r \cdot (2^n) + 1 \rfloor}{2^n})\}_{n \in \mathbb{N}}$, $s_{\mathbb{R}}|_{\mathbb{D}} = s_{\mathbb{D}}$, $s_{\mathbb{D}}|_{\mathbb{Z}} = s_{\mathbb{Z}}$, and the basic operations $+$, \cdot are preserved. We discuss only two most important points: the set-theoretic definition of $s_{\mathbb{D}} : \mathbb{D} \mapsto \text{Day } \omega$ and the choice operator.

We denote the set of dyadic numbers of the form $\frac{j}{2^n}$ (where $j \in \mathbb{Z}$) as \mathbb{D}_n . Observe that the sequence $\mathbb{D}_0 = \mathbb{Z}$, \mathbb{D}_n is increasing and $\bigcup_{n \in \mathbb{N}} \mathbb{D}_n = \mathbb{D}$. Then, for any n we define a *recursive operator* $I_n : (\text{Day } \omega \oplus n)^{\mathbb{D}_n} \mapsto (\text{Day } \omega \oplus (n+1))^{\mathbb{D}_{n+1}}$ which extends the domain of \mathbb{D}_n to \mathbb{D}_{n+1} , assigning $\mathbb{D}_{n+1} \setminus \mathbb{D}_n$ values according to (17). It easily follows that $\{a \mid b\} \in \text{Day } \omega \oplus (n+1)$ if $a, b \in \text{Day } \omega \oplus n$. Then using $s_{\mathbb{Z}}$ on \mathbb{D}_0 with MML's fixed point combinator `NAT_1:sch 11` we construct $s_{\mathbb{D}}$ and prove that the values belong to $\bigcup_{n \in \mathbb{N}} \text{Day } n$.

We now want to obtain $s_{\mathbb{R}}|_{\mathbb{D}} = s_{\mathbb{D}}$. To choose a representative of the equivalence class $[x]$ for a given x we can use "gluing" $s_{\mathbb{R}}(r) = s_{\mathbb{D}}(r)$ for dyadic r and use global choice otherwise (taking special care, since choosing c from $[x]_{\approx}$ is a proper class). Using an adaptation of *Scott's trick*, we can expect c to be the youngest (in the sense of \mathfrak{b}). We therefore can replace the proper class $[x]_{\approx}$ by the set $\{y \in \text{Day } \mathfrak{b} x : y \approx x\}$. However, this set still can

have several elements, so in the first place we aim to reduce the cardinality of L_c, R_c and the cardinality of their union. Using the Hessenberg sum, we can minimize all three. For this, we will use $\overline{L_c} \oplus \overline{R_c}$ instead of $\overline{L_c} \cup \overline{R_c}$ and use properties of \approx . Finally, we would like the globally selected c to be suitable. By suitable, we mean minimal w.r.t. \mathfrak{b} as well as having only suitable elements in L_c, R_c . For this, we again use a transfinite sequence, the last element of which has only suitably selected elements of Day α

```

let  $\alpha$  be Ordinal;
func  $\mathfrak{U}_{niq\_op} \alpha \rightarrow$  Sequence means :: SURREALO: def 29
  dom it = succ  $\alpha \wedge \forall \beta$  be Ordinal.  $\beta \in$  succ  $\alpha \Rightarrow$  (it. $\beta \subseteq$  Day  $\beta \wedge$ 
    ( $\forall x$  be Surreal.  $x \in$  it. $\beta \Leftrightarrow$  ( $x \in$  union rng (it| $\beta$ )  $\vee$  ( $\beta =$  born_eq  $x \wedge$ 
       $\exists Y$  be non empty surreal-membered set .
         $Y =$  born_eq_set  $x \cap$  made_of union rng (it| $\beta$ )  $\wedge x =$  the  $Y$ -smallest Surreal));

```

where **the** is the global choice operator, **born_eq** x is the \mathfrak{b} of youngest surreal that is $\approx x$, **born_eq_set** x is the set of youngest surreal that is $\approx x$, **made_of** X is the set of such surreals that their both left and right members belong to X , and Y -smallest means that it has the smallest cardinality, that is $\overline{L_c} \oplus \overline{R_c}$.

The definition implicitly assumes that **born_eq_set** $x \cap$ **made_of** \bigcup rng (it| β) is non-empty because Y is non-empty. In consequence, Y -smallest Surreal is also non-empty and we can use global choice. Finally, we can use transfinite induction to select a unique element for each class $[x]_{\approx}$, denoted $\mathfrak{U}_{niq}x$.

It is important to notice, that the properties that need to be proved about the globally selected numbers (such as youngest, smallest cardinality, suitable member) must be proved by simultaneous induction, just like it was the case with the correctness of multiplication proofs in Section 5. We call the type of such elements **uSurreal**. This type is crucial for the definition of $s_{\mathbb{R}}$, as values of $s_{\mathbb{D}}$ are **uSurreal**, and before Day ω there are no more **uSurreal**. This means that $s_{\mathbb{R}}$ can be defined using \mathfrak{U}_{niq} without “gluing”.

Next, we define * ordinal numbers to be all the surreal numbers x , for which $R_x = \emptyset$ (following Conway). Subsequently, again using a transfinite sequence, we define the operator s_{On} from ordinals to * ordinal. We additionally apply \mathfrak{U}_{niq} to it, so that the values are **uSurreal** (this is justified, as we constructed \mathfrak{U}_{niq} to be * ordinal on * ordinal). The proposed construction of **uSurreal** builds a bridge that helps us formally combine proofs about the surreal numbers in the general sense with the proofs that use the tree-theoretic definition that uses real and ordinal. This will be important for several results in the next section.

7 Conway Normal Form (CNF)

Conway has partitioned \mathbb{N} using a property similar to Archimedeaness. Under this partition, the surreals behave somewhat similarly to a vector space with arbitrary (ordinal-indexed) dimensions. The Conway Normal Form theorem will give a concept akin to coordinates in that space. These are all weak analogies, however, they give a hint as to why CNF is important for surreal numbers.

The most common ordered field, the real numbers, has the Archimedean property, i.e., for all $x, y \in \mathbb{R}^+$, $x < n \cdot y$ for some $n \in \mathbb{N}$. However, this is not true for infinite cardinals, so the Archimedean-like partition of surreal is defined somewhat differently (remember that $s_{\mathbb{Z}}$ is the natural embedding of integers into surreal, as defined in the previous Section):

► **Definition 12** (commensurate, infinitely less). *Let x, y be surreal numbers. We say x, y are commensurate if $x < s_{\mathbb{Z}}(n) \cdot y \wedge y < s_{\mathbb{Z}}(m) \cdot x$ for some $n, m \in \mathbb{N}$. We say x is infinitely less than y and write $x <^{\infty} y$ if $x \cdot s_{\mathbb{Z}}(n) < y$ for all $n \in \mathbb{N} \setminus \{0\}$.*

29:14 Conway Normal Form for the Formalization of Surreal Numbers

As the definition of commensurate numbers only makes sense for positive numbers, we will refer to x, y as *commensurate in absolute terms* if $|x|, |y|$ are commensurate, where $|\cdot|$ is the standard absolute value. (Similarly $<^\infty$ is defined in general, but only used for non-negative numbers.) We next define the *power of ω* (also called ω -map [5]) as follows:

► **Definition 13** (ω^x). *Let x be a surreal. Then the x power of ω , written ω^x is defined as:*

$$\omega^x = \{0_{\mathbb{N}o}, s_{\mathbb{R}}(r) \cdot \omega^{xL} \mid s_{\mathbb{R}}(s) \cdot \omega^{xR}\} \quad (18)$$

where r, s range over all positive reals and $\omega = s_{\mathbb{O}n}(\omega)$.

To define the function formally, we again use Theorem 10 with $\theta = \mathfrak{b} x$, $X(\alpha) = \text{Day } \alpha$, and

$$H(o, S) = \{\{0_{\mathbb{N}o}\} \cup \{(\bigcup \text{rng } S)(o^L) \cdot s_{\mathbb{R}}(r) \mid o^L \in L_o \wedge r \in \mathbb{R}^+\}, \\ \{(\bigcup \text{rng } S)(o^R) \cdot s_{\mathbb{R}}(s) \mid o^R \in R_o \wedge s \in \mathbb{R}^+\}\}. \quad (19)$$

Note, that ω^x is different from exponentiation (the differences are subtle, see [5, page 38]), but many of its properties are similar: $\omega^0 = 1_{\mathbb{N}o}$, $\omega^{x+y} = \omega^x \cdot \omega^y$, and $\omega^x \leq \omega^y$ if $x \leq y$ and additionally $\omega^x <^\infty \omega^y$ if $x < y$.

The underlying idea for Conway's Normal Form of $x \not\approx 0_{\mathbb{N}o}$ is the observation that using the power of ω we can determine a unique leader ω^y commensurate in absolute terms with x :

► **Theorem 14.** *Let $x \not\approx 0_{\mathbb{N}o}$. Then there exists a unique y being uSurreal such that $r \in \mathbb{R} \setminus \{0\}$ for which $|x - s_{\mathbb{R}}(r) \cdot \omega^y| <^\infty \omega^y$.*

Notice that when $x_1, x_2 \not\approx 0_{\mathbb{N}o}$ have the same leader ω^y then $|x_1|, |x_2|$ are commensurate. Additionally ω^y is the leader for $x_1 \cdot s_{\mathbb{R}}(r)$, $x_1 + x_2$ for arbitrary $r \in \mathbb{R} \setminus \{0\}$.

Fix $x \not\approx 0_{\mathbb{N}o}$. We can *approximate* x using powers of ω , i.e., $x = s_{\mathbb{R}}(r_0) \cdot \omega^{y_0} + x_1$ and $x_1 <^\infty \omega^{y_0}$. Further, if $x_1 \not\approx 0_{\mathbb{N}o}$, we obtain a better approximation by a sum: $x = s_{\mathbb{R}}(r_0) \cdot \omega^{y_0} + s_{\mathbb{R}}(r_1) \cdot \omega^{y_1} + x_2$, where $x_2 <^\infty \omega^{y_1} <^\infty \omega^{y_0}$ and so forth. As a consequence, x can be represented in a form (already close to Cantor Normal Form):

$$x = s_{\mathbb{R}}(r_0) \cdot \omega^{y_0} + s_{\mathbb{R}}(r_1) \cdot \omega^{y_1} + s_{\mathbb{R}}(r_2) \cdot \omega^{y_2} + \dots s_{\mathbb{R}}(r_{k-1}) \cdot \omega^{y_{k-1}} + x_k. \quad (20)$$

Unfortunately, a finite number of iterations does not guarantee $x_k \approx 0_{\mathbb{N}o}$. As such, an infinite sum will be necessary (infinite in the ordinal sense, so not just ω). To state the CNF theorem, we must define this sum formally. Conway assumes its existence, but the formal proof of its convergence is actually more involved than the CNF proof. We adapt Erlich's approach [8]:

► **Definition 15** (θ -term). *Let x be a surreal, α, α' be ordinals. Let $\mathbf{r} := \{r_\beta\}_{\beta \leq \alpha}$ be a sequence of non-zero reals, $\mathbf{y} := \{y_\beta\}_{\beta \leq \alpha}$ be a decreasing sequence of surreals, and $\mathbf{s} := \{s_\beta\}_{\beta \leq \alpha'}$ be a sequence of surreals where $\alpha \leq \alpha'$ (i.e., \mathbf{s} can be longer than \mathbf{r}, \mathbf{y}). We call x the $(\theta, \mathbf{s}, \mathbf{y}, \mathbf{r})$ -term if $|x - (s_\theta + s_{\mathbb{R}}(r_\theta) \cdot \omega^{y_\theta})| <^\infty \omega^{y_\theta}$ where $\theta \leq \alpha$. Additionally, we write $x \in \bigcap_{\theta, \mathbf{s}, \mathbf{y}, \mathbf{r}} \theta$ if $\theta \leq \alpha$ and x is $(\gamma, \mathbf{s}, \mathbf{y}, \mathbf{r})$ -term for arbitrary $\gamma < \theta$.*

Note, that for any $a \leq b \leq c$, if $a \in \bigcap_{\theta, \mathbf{s}, \mathbf{y}, \mathbf{r}} \theta$ and $c \in \bigcap_{\theta, \mathbf{s}, \mathbf{y}, \mathbf{r}} \theta$ then $b \in \bigcap_{\theta, \mathbf{s}, \mathbf{y}, \mathbf{r}} \theta$. Conway calls classes with this property *convex*.

Now we can formally express Conway's sentence *the simplest number whose β -term is $r_\beta \cdot \omega^{y_\beta}$* in Erlich's way [8]. We say that a triple $\mathbf{s}, \mathbf{y}, \mathbf{r}$ is simplest on β if

- $s_\beta = 0_{\mathbb{N}o}$ for $\beta = 0$,
- if $0 < \beta$ holds: s_β is uSurreal, $s_\beta \in \bigcap_{\beta, \mathbf{s}, \mathbf{y}, \mathbf{r}} \theta$ and for every uSurreal $a \neq s_\beta$, if $b \in \bigcap_{\beta, \mathbf{s}, \mathbf{y}, \mathbf{r}} \theta$ then $\mathfrak{b} s_\beta < \mathfrak{b} a$.

The expression $\bigcap_{\beta, \mathbf{s}, \mathbf{y}, \mathbf{r}} \theta$ depends on all s_γ for $\gamma < \beta$, so we can use it to specify s_β .

Let θ be an ordinal. Consider, as previously, $\{\mathbf{r}_\beta\}_{\beta < \theta}$, $\{\mathbf{y}_\beta\}_{\beta < \theta}$ and let $\{\mathbf{s}_\beta\}_{\beta \leq \theta}$ be a sequence of $\mathbf{uSurreal}$ where additionally the triple $\mathbf{s}, \mathbf{y}, \mathbf{r}$ is simplest on β for $\beta \leq \theta$. Then $\sum_{\beta < \theta} \omega^{\mathbf{y}_\beta} \cdot \mathbf{r}_\beta$ is defined to be \mathbf{s}_θ [8, 5]. As usual, to construct a suitable θ -long sequence \mathbf{s} we apply transfinite induction, first showing the existence of a suitable β -long sequence for $\beta \leq \theta$. Indeed, using the β -step assumption we can construct a suitable sequence $\mathbf{p} := \{p_\gamma\}_{\gamma < \beta}$ and extend it by the assignment $p_\beta = \mathfrak{U}_{\text{inf}} e$ for some $e \in \bigcap_{\beta, \mathbf{p}, \mathbf{y}, \mathbf{r}} e$. The existence of such e is the key problem: If β is a limit ordinal, i.e., the sequence $\{p_\gamma\}_{\gamma < \beta}$ does not have a last element⁵. Conway introduced it highly informally [5]. Erlich [8] does it formally using an approach where the class of every e where $e \in \bigcap_{\beta, \mathbf{p}, \mathbf{y}, \mathbf{r}} e$ corresponds to a non-empty intersection of the descending transfinite sequence of convex subclasses of \mathbf{No} . This assumes a stronger foundation and is not possible in Mizar (nor Isabelle/ZF or Metamath). Indeed, his surreal numbers with lexicographic order are full, equivalently complete or equivalently every nested sequence $\{I_\gamma\}_{\gamma < \beta}$ of non-empty convex subclasses has a non-empty intersection (see Theorem 4 in [8]). We cannot do this for $\gamma < \beta$ when β is a limit ordinal. To solve this in standard set theory, we defined two somewhat complicated sequences $\{l_\gamma\}_{\gamma < \beta}$, $\{u_\gamma\}_{\gamma < \beta}$, defined by, $l_\gamma := p_{1+\gamma} + (\mathbf{s}_\mathbb{R}(\mathbf{r}_{1+\gamma}) - 1_{\mathbf{No}}) \cdot \omega^{\mathbf{y}_{1+\gamma}}$, $u_\gamma := p_{1+\gamma} + (\mathbf{s}_\mathbb{R}(\mathbf{r}_{1+\gamma}) + 1_{\mathbf{No}}) \cdot \omega^{\mathbf{y}_{1+\gamma}}$ for $\gamma < \beta$ that in contrast to \mathbf{p} are monotonous (increasing and decreasing, respectively) and $\{\bigcup_{\gamma < \beta} \{l_\beta\} \mid \bigcup_{\gamma < \beta} \{u_\beta\}\}$ is a member of the intersection.

With these, the formalization of the following theorem is attainable. It says that the approximation of x in ω way can be performed at most $\mathfrak{b} x$ times, since their \mathfrak{b} -s of subsequent partial approximating sums give an increasing sequence bounded by $\mathfrak{b} x$.

► **Theorem 16** (Conway's Normal Form Theorem, Mizar ID: SURREALC:100,102). *For every surreal x there exists a unique $\{\mathbf{r}_\beta\}_{\beta < \theta}$ sequence of non-zero real, $\{\mathbf{y}_\beta\}_{\beta < \theta}$ decreasing sequence of $\mathbf{uSurreal}$ such that $x \approx \sum_{\beta < \theta} \omega^{\mathbf{y}_\beta} \cdot \mathbf{s}_\mathbb{R}(\mathbf{r}_\beta)$. Moreover \mathfrak{b} of the sum $\leq \mathfrak{b} x$.*

Conway's Normal Form allows characterizing any x using a sum of two transfinite sequences \mathbf{r}, \mathbf{y} . Rather than a regular sum, it is interpreted more as a Hahn-Mal'cev-Neumann infinite series sum. This characterization is key to the further formalization of Conway [5], in particular it will allow constructing the n^{th} -root of x , showing that odd-degree polynomials have roots, characterizing omnific surreal integers and further results as discussed in the conclusion, Section 9.

8 Related Work

There are several formalization pertinent to surreal numbers in various systems. Mamane's formalization in Coq [14], Obua's in Isabelle/HOLZF [17], and Carneiro, Morrison, and Nakade's⁶ in Lean follow an approach closer to Conway. All three avoid induction-recursion by starting with games. Mamane motivates his work as a "stress-test" of Coq in the formalization of a very set-theoretic definition. He proved that surreal numbers form a commutative ring (without associativity), and without *permuting induction*, that was only formulated in the article. Without this induction, the formalization needs to cover 2^n cases corresponding to the edges of an n -dimensional cube. Our work deal with this using Prod^O , Prod^C inductively to cover the cartesian product.

⁵ We focus on the case of β being the limit ordinal, where $\gamma < \beta \iff 1+\gamma < \beta$.

⁶ https://github.com/leanprover-community/mathlib/blob/master/src/set_theory/surreal/

Obua's work focuses on the development of the infrastructure for surreal numbers. The formalization only reaches the fact that surreals form an additive group. Similarly, the Lean formalization defines surreal numbers with addition and show that they form a commutative group. It also includes an embedding of ordinals into surreal, a manually defined halving operator and an embedding of dyadic numbers into surreal.

Induction-recursion, as studied in intuitionistic type theory [6], could allow a more direct definition of the surreal numbers. However, we are not aware of any formalizations of the surreal numbers that make use of induction-recursion.

Nittka followed the tree-theoretic approach in Mizar [16]. After showing the involutiveness of minus, further definitions and properties became too involved in this approach in Mizar and made us abandon this method. With the `uSurreal` obtained in our formalization, it is possible to easier continue with that approach.

The largest formalization of surreal numbers focusing solely on the tree-theoretic approach has been developed in the Megalodon proof system⁷. Without the use of permuting induction, it shows that surreals form a field and defines the square root. Megalodon stands out as the only system where integers, reals and ordinals used throughout the system are carved out of the surreal numbers, rather than being added on top. In comparison to our work, the Megalodon formalization lacks the Conway Normal Form theorem and the theorems and definitions leading up to it. Additionally, we formalized morphisms between the MML numbers and Conway numbers, enabling transfer of theorems between them. The Megalodon formalization of the inverse and square root operations has been completed based on the ones done in our Mizar formalizations, demonstrating the adaptability of our approach to other systems.

9 Conclusion

We formalized a large number of properties of surreal numbers in the Mizar proof assistant system. We initially focused on Conway's approach to introduce the concept, which simplifies the definitions of arithmetic operations, and then showed the equivalence of our approach to the tree-theoretic approach. For this, we built a bridge that allows joining the proofs in both approaches (`uSurreal`) and using it, we were able to reach Conway's Normal Form. Due to the relatively weak foundations of Mizar (first, there is no induction-recursion; second, reasoning must be explicitly conducted on sets rather than classes), we believe that our approach can be useful for other formal systems.

CNF is crucial for further formalization of Conway's results [5]. Future work includes a formalization of n^{th} -root of x . We are considering two approaches. First, to combine the use of Kruskal-Gonshor exponential function with logarithms. Alternatively, a more direct use of CNF, following [5], is possible. With the n^{th} -root of x , one can show that \mathbb{No} is algebraically real-closed, i.e., that odd-degree polynomials have roots. CNF is also needed to characterize omnific surreal integers (i.e., surreals that satisfy $x \approx \{x - 1 \mid x + 1\}$). It is then possible to show that every surreal number can be represented as the quotient of two omnific integers. This can lead to the formalization of surcomplexes. Finally, CNF is fundamental for further works on s-hierarchical ordered field \mathbb{No} [7, 8].

⁷ <http://grid01.ciirc.cvut.cz/~chad/100thms/100thms.html>

References

- 1 Maan T. Alabdullah, Essam El-Seidy, and Neveen S. Morcos. On numbers and games. *International Journal of Scientific & Engineering Research*, 11, February 2022.
- 2 Grzegorz Bancerek, Czesław Byliński, Adam Grabowski, Artur Korniłowicz, Roman Matuszewski, Adam Naumowicz, and Karol Pąk. The role of the Mizar Mathematical Library for interactive proof development in Mizar. *Journal of Automated Reasoning*, 2017. doi:10.1007/s10817-017-9440-6.
- 3 Grzegorz Bancerek, Czesław Byliński, Adam Grabowski, Artur Korniłowicz, Roman Matuszewski, Adam Naumowicz, Karol Pąk, and Josef Urban. Mizar: State-of-the-art and Beyond. In Manfred Kerber, Jacques Carette, Cezary Kaliszyk, Florian Rabe, and Volker Sorge, editors, *Intelligent Computer Mathematics - International Conference, CICM 2015*, volume 9150 of *LNCS*, pages 261–279. Springer, 2015. doi:10.1007/978-3-319-20615-8_17.
- 4 Chad E. Brown and Karol Pąk. A tale of two set theories. In Cezary Kaliszyk, Edwin C. Brady, Andrea Kohlhasse, and Claudio Sacerdoti Coen, editors, *Intelligent Computer Mathematics - 12th International Conference, CICM 2019*, volume 11617 of *LNCS*, pages 44–60. Springer, 2019. doi:10.1007/978-3-030-23250-4_4.
- 5 John H. Conway. *On Numbers And Games*. A K Peters Ltd., 2nd edition, 2001. First Edition: 1976.
- 6 Peter Dybjer. A general formulation of simultaneous inductive-recursive definitions in type theory. *J. Symb. Log.*, 65(2):525–549, 2000. doi:10.2307/2586554.
- 7 Philip Ehrlich. The absolute arithmetic continuum and the unification of all numbers great and small. *Bulletion Symbolic Logic*, 18(1):1–45, 2012. doi:10.2178/bs1/1327328438.
- 8 Philip Ehrlich. Number systems with simplicity hierarchies: A generalization of Conway’s theory of surreal numbers. *J. Symb. Log.*, 66(3):1231–1258, 2001. doi:10.2307/2695104.
- 9 Adam Grabowski and Artur Korniłowicz. Implementing more explicit definitional expansions in mizar (short paper). In Adam Naumowicz and René Thiemann, editors, *14th International Conference on Interactive Theorem Proving, ITP 2023, July 31 to August 4, 2023, Białystok, Poland*, volume 268 of *LIPICs*, pages 37:1–37:8. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPICs.ITP.2023.37.
- 10 Adam Grabowski, Artur Korniłowicz, and Adam Naumowicz. Four decades of Mizar. *Journal of Automated Reasoning*, 55(3):191–198, 2015. doi:10.1007/s10817-015-9345-1.
- 11 Gretchen Grimm. An introduction to surreal numbers, 2012.
- 12 Cezary Kaliszyk and Karol Pąk. Semantics of Mizar as an Isabelle object logic. *Journal of Automated Reasoning*, 63(3):557–595, 2019. doi:10.1007/s10817-018-9479-z.
- 13 Sebastian Koch. Natural addition of ordinals. *Formalized Mathematics*, 27(2):139–152, 2019. doi:10.2478/forma-2019-0015.
- 14 Lionel Elie Mamane. Surreal numbers in Coq. In Jean-Christophe Filliâtre, Christine Paulin-Mohring, and Benjamin Werner, editors, *Types for Proofs and Programs, TYPES 2004*, volume 3839 of *LNCS*, pages 170–185. Springer, 2004. doi:10.1007/11617990_11.
- 15 Norman D. Megill. *Metamath: A Computer Language for Pure Mathematics*. Lulu Press, Morrisville, North Carolina, 2007.
- 16 Robin Nittka. Conway’s games and some of their basic properties. *Formalized Mathematics*, 9(2):73–71, 2011.
- 17 Steven Obua. Partizan games in Isabelle/HOLZF. In Kamel Barkaoui, Ana Cavalcanti, and Antonio Cerone, editors, *Theoretical Aspects of Computing - ICTAC 2006*, volume 4281 of *LNCS*, pages 272–286. Springer, 2006.
- 18 Lawrence C. Paulson. Set theory for verification: I. From foundations to functions. *J. Autom. Reasoning*, 11(3):353–389, 1993. doi:10.1007/BF00881873.
- 19 Karol Pąk. Conway numbers – formal introduction. *Formalized Mathematics*, 31(1):193–203, 2023. doi:10.2478/forma-2023-0018.
- 20 Karol Pąk. Integration of game theoretic and tree theoretic approaches to Conway numbers. *Formalized Mathematics*, 31(1):205–213, 2023. doi:10.2478/forma-2023-0019.

29:18 Conway Normal Form for the Formalization of Surreal Numbers

- 21 Karol Pał. The ring of Conway numbers in Mizar. *Formalized Mathematics*, 31(1):215–228, 2023. doi:10.2478/forma-2023-0020.
- 22 Dierk Schleicher and Michael Stoll. An introduction to Conway’s games and numbers. *Moscow Mathematical Journal*, 6, 2004. doi:10.17323/1609-4514-2006-6-2-359-388.
- 23 Claus Tøndering. Surreal numbers—an introduction. *HOTP*, version 1.7, December 2019.

A Coq Formalization of Taylor Models and Power Series for Solving Ordinary Differential Equations

Sewon Park 

Graduate School of Informatics, Kyoto University, Japan

Holger Thies 

Graduate School of Human and Environmental Studies, Kyoto University, Japan

Abstract

In exact real computation real numbers are manipulated exactly without round-off errors, making it well-suited for high precision verified computation. In recent work we propose an axiomatic formalization of exact real computation in the Coq theorem prover. The formalization admits an extended extraction mechanism that lets us extract computational content from constructive parts of proofs to efficient programs built on top of AERN, a Haskell library for exact real computation.

Many processes in science and engineering are modeled by ordinary differential equations (ODEs), and often safety-critical applications depend on computing their solutions correctly. The primary goal of the current work is to extend our framework to spaces of functions and to support computation of solutions to ODEs and other essential operators.

In numerical mathematics, the most common way to represent continuous functions is to use polynomial approximations. This can be modeled by so-called Taylor models, that encode a function as a polynomial and a rigorous error-bound over some domain. We define types of classical functions that do not hold any computational content and formalize Taylor models to computationally approximate those classical functions. Classical functions are defined in a way to admit classical principles in their constructions and verification. We define various basic operations on Taylor models and verify their correctness based on the classical functions that they approximate. We then shift our interest to analytic functions as a generalization of Taylor models where polynomials are replaced by infinite power series. We use the formalization to develop a theory of non-linear polynomial ODEs. From the proofs we can extract certified exact real computation programs that compute solutions of ODEs on some time interval up to any precision.

2012 ACM Subject Classification Theory of computation → Logic and verification

Keywords and phrases Exact real computation, Taylor models, Analytic functions, Computable analysis, Program extraction

Digital Object Identifier 10.4230/LIPIcs.ITP.2024.30

Supplementary Material

Software (Source Code): <https://github.com/holgerthies/coq-aern> [20]

archived at [swh:1:dir:91c89245541a8dbcad3ab085bb8682112e684311](https://swh.1.dir:91c89245541a8dbcad3ab085bb8682112e684311)

Funding *Sewon Park*: has been supported by JSPS KAKENHI (Grant-in-Aid for JSPS Fellows) JP22F22071 as a JSPS International Research Fellow.

Holger Thies: Supported by JSPS KAKENHI Grant Numbers JP20K19744, JP23H03346 and JP24K20735.

1 Introduction

A typical problem in numerical analysis with vast applications across various fields such as physics, engineering, biology, and economics, is to approximate solutions to initial value problems (IVPs) of the form

$$\dot{y}(t) = f(y(t)) ; y(t_0) = y_0$$



© Sewon Park and Holger Thies;

licensed under Creative Commons License CC-BY 4.0

15th International Conference on Interactive Theorem Proving (ITP 2024).

Editors: Yves Bertot, Temur Kutsia, and Michael Norrish; Article No. 30; pp. 30:1–30:19

Leibniz International Proceedings in Informatics



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

where $\dot{y}(t)$ denotes the derivative of the function $y(t)$ with respect to the time variable t . Approximating the solution accurately can be challenging as small perturbations in the initial condition can lead to large changes in the solution. Traditional numerical methods therefore often lack the robustness required in safety-critical applications. Thus, approaches to rigorously solve ODEs have been studied extensively. For example, methods from interval arithmetic such as [31] allow to compute the solution reliably in the sense that the resulting intervals are promised to contain the exact solutions. However, accumulation of over-estimations in interval computations make it challenging to obtain solutions in arbitrarily high accuracy. Exact real computation, based on the theory of constructive [4] and computable analysis [36], solves this problem by using infinite representations of real numbers. Instead of approximating a real number by an interval, real numbers are expressed exactly e.g. by an infinite stream of nested shrinking intervals. In this way, real numbers can be represented and manipulated exactly, without round-off errors and other sources of numerical uncertainty inherent in finite precision arithmetic. Implementations of exact real computation such as, for example, [30, 3, 22], provide abstract data types for real numbers hiding representation-specific details from their users. Consequently, users can write and reason about their programs intuitively regarding exact real numbers as the familiar abstract entities from mathematics. See [6, 32] for more about this aspect of exact real computation.

Recently, we have worked on axiomatically formalizing exact real computation in a type-theoretical setting [19] and implementing the idea as the Coq library `cAERN` [20]. By making use of Coq’s code extraction features, we map axiomatically defined types and operations to abstract data types and primitive operations in `AERN`, a Haskell library for exact real computation [22]. As primitive real number operations are optimized in the implementation of `AERN`, the extracted certified exact real computation programs are usually more efficient than mapping everything to primitive data-types. For example, in [19] we show that the execution time of extracted programs is comparable to hand-written `AERN` programs.

The main goal of this work is to extend the `cAERN` formalization of exact real computation to solution operators for IVPs and other higher-order problems. When dealing with function spaces, representing functions accurately is essential for efficient computation. In numerics, the predominant approach to represent continuous real functions is through polynomial approximations. Taylor models [27] provide a systematic approach to approximating functions by polynomials. A Taylor model represents a function on some interval as a polynomial approximation (typically derived from the function’s Taylor series expansion) together with a rigorous error bound on the interval. Many mathematical operations can be directly implemented on Taylor models, accounting for dependencies between variables and thus ensuring more accurate enclosures of function values. Recently the use of Taylor models has also been shown to be beneficial in efficient exact real computation [7].

In this work, we implement a solver for initial value problems for polynomial first-order ODEs and verify its correctness. The solver is based on computing the power series expansion of the solution up to an arbitrarily high degree. To this end we formalize Taylor models and their generalization as infinite power series. We use these to locally represent analytic functions exactly. Thus, our solver computes a functional representation of the solution $y(t)$ on a small time interval $[0, t_0]$, which can be used to approximate the value of the function in the interval up to any desired precision. We can then extend the solution by computing the value $y(t_0)$ and use it as a new initial value to continue the procedure, similar to single step methods in numerics. Note that since we get an exact representation of the real number $y(t_0)$ we do not have to deal with approximations in this step (see Section 6 for details).

Although the solution method is generic, for simplicity we currently only consider one-dimensional ODEs in the Coq implementation. While this excludes most interesting applications, we think it demonstrates well how the method works, and it should not be difficult to extend to higher dimension.

Since we work on a constructive setting, saying \mathbb{R} to be the type of exact real numbers, having a function $f : \mathbb{R} \rightarrow \mathbb{R}$ means that we already have a way to compute, in a sense, the function f exactly. Thus, it does not make much sense to develop a theory of approximating them using Taylor models or power series. In other words, the standard function type $\mathbb{R} \rightarrow \mathbb{R}$ is stronger than the type of functions that our Taylor models and power series intend to approximate. We therefore formalize a type of classical (partial) functions that is weaker than the standard functions. Classical functions can be constructed based on classical reasoning, e.g. the sign function of reals, but do not yield any computational contents. We define our Taylor models and power series to approximate such classical functions instead and use them as classical specifications which we can reason about based on the classical analysis.

The paper is structured as follows. In the rest of this section, we review the setting of our Coq development and some related works. In Section 2, we formalize classical partial functions and in Section 3 we define the notions of their continuity and differentiability. In Section 4 we formalize a variant of simple univariate Taylor models with real-valued polynomials and a real-valued error bound which we use to approximate classical partial functions. In Section 5 we then further extend this to analytic functions, which we can represent exactly by infinite sequences of polynomials. Finally, in Section 6 we use this to define polynomial initial value problems and solution operators for them.

All results in this paper have been implemented in Coq as an extension of the cAERN library [20]. The new formalization presented consists of approximately 8000 lines of code, while the complete cAERN library consists of approximately 29000 lines of code. The files relevant for the new formalization are `ClassicalMonads`, `ClassicalPartiality`, and `ClassicalPartialReals` for Section 2; `ClassicalTopology`, `ClassicalContinuity`, and `ClassicalDifferentiability` for Section 3; `Poly` and `TaylorModel` for Section 4; `Powerseries` for Section 5; and `Ode` for Section 6.

1.1 Background

In this paper, we focus mostly on our Coq development, and thus present most results directly in the syntax of Coq. However, it is also straightforward for readers who are not familiar with Coq syntax to translate it using type-theoretic constructions. For example, we use `Prop` for an impredicative type universe (of propositions) without large eliminations and `Type` for a type universe of (small) types. We write $(A \vee B) : \text{Prop}$ for the disjunction in `Prop` whereas we write $A + B : \text{Type}$ for the usual sum type. Similarly, by $(\text{exists } x : A, B x) : \text{Prop}$, we refer to the existence in `Prop` whereas $\{x : A \mid B x\} : \text{Type}$ and $\{x : A \ \& \ B x\} : \text{Type}$ denote the usual Σ -type. The Π -types are written as $\text{forall } x : A, B x$.

As in our previous works [19, 21], we assume axioms that make `Prop` classical, in particular the law of excluded middle of the form $(\text{forall } P : \text{Prop}, P \vee \neg P) : \text{Prop}$. We further assume the dependent function extensionality, classical propositional extensionality saying that for any two types $P \ Q : \text{Prop}$, $(P \rightarrow Q) \wedge (Q \rightarrow P)$ implies $P = Q$, and proof-irrelevance of classical propositions saying that any type P in `Prop` is a *subsingleton* type $\text{forall } x \ y : P, x = y$. We say a type P is a classical proposition when $P : \text{Prop}$. We also often write $a : A$ such that $B x$ classically exists to mean that $\text{exists } a : A, B x$ holds and write P or Q holds classically to mean that $(P \vee Q) : \text{Prop}$ holds. Otherwise, we refer to the constructive variants.

As a direct consequence of making `Prop` classical, equality types become classical and proof-irrelevant. Therefore, we refer a map $f : A \rightarrow B$ being a type-theoretic equivalence by the (constructive) existence of its inverse $g : B \rightarrow A$ such that for all $x : A$, $g (f x) = x$ and for all $y : B$, $f (g y) = y$ hold.

In the previous works, based on this setting we axiomatically formalize types and operations used in exact real number computation. Most importantly, we assume that there is a type `R` for real numbers containing two distinct constants `0` and `1`, the standard arithmetical operators and a semi-decidable comparison operator `<`. We restrict the reciprocal function x^{-1} to require a classical proof that $x \neq 0$ and the limit operator to receive a classical proof of the rapid convergence of a given sequence. The axioms are chosen carefully in a way that allows to reason about properties of real numbers classically, without breaking computational content. Hence, one important feature of the `cAERN` library (and thus the name) is that we can extract Haskell programs on top of the `AERN` library which is proven to be correct under the assumption that the basic operations in `AERN` are implemented correctly. This approach has the advantage that extracted programs are more efficient than extracting everything to basic types like integers, that programs are more readable, and that we can easily integrate certified code with non-verified code in the `AERN` library. Interesting examples we have formalized and extracted include the maximum function, the absolute value function, a root-finding functional from a constructive intermediate value theorem [17], real and complex square roots [18], computing fractals from proofs related to open, closed, compact, overt, located subsets of Euclidean spaces, and so on [21].

An important note to make here is that, to simplify presentation, the notations we use in this paper are not completely identical to the notations we use in the implementation. For example, we use the notation `^Real` in the source code for exact real numbers not `R` as in this paper. However, the modifications are minimal and thus such correspondence should be clear for readers who also read the source code.

1.2 Related work

Approximating solution trajectories for ordinary differential equations is a key problem in numerical mathematics and methods for rigorous computation have been studied extensively (e.g. [2, 12]) and several rigorous tools have been developed (e.g. [23, 9]). In particular, formal verification of numerical methods for ODEs in proof assistants has been considered previously, e.g. in Isabelle [14, 13] and Coq [24]. The above mentioned works are based on interval arithmetic, thus verify that ODE solutions can be rigorously enclosed in some interval. Our work, on the other hand, is based on computable analysis and therefore, in a sense, uses a stronger notion of correctness, i.e. we need to show that we can make the enclosure at each point arbitrarily small. In this sense, our work is similar to constructive approaches to analysis. A larger formalization of constructive analysis in Coq can be found in the `CoRN` library [11] which also includes some previous results on constructive formalizations of ODE solutions in Coq [25]. In contrast to `CoRN` our goal is not to compute inside the proof assistant, but to extract exact real computation programs and to adhere to an abstract axiomatic formalization of real numbers similar to real number types in implementations of exact real computation (see [18] for details). Further, [25] uses the Picard Iteration algorithm, while our proposed method is based on high-order power series expansions. The method is motivated by results from real number complexity theory which suggests that it should be more efficient if the desired output precision is high [29, 8, 16]. We are not aware of any formal verification of this method.

Many verified solvers for ODEs use Taylor models. Taylor models are typically used to deal with the dependency problem of interval arithmetic and to get tighter enclosures of function values than what is achieved with simple intervals [27]. There already exist several formalizations of Taylor models in proof assistants, e.g. [35, 28]. In particular, [28] presents a formally verified implementation of univariate Taylor models in Coq. However, it should be mentioned that both our use case and implementation are quite different from the usual approaches in interval arithmetic. First, as our polynomials and error terms are exact reals, we do not need to deal with issues arising when implementing them using interval arithmetic. Second, while our Taylor models could also be used to compute (real-valued) interval enclosures of functions, our main motivation is to use them to represent certain types of functions exactly as an infinite sequence of polynomials and operate on them efficiently, more similar to the idea of a function space representation in computable analysis [33].

Lastly, although our main aim is to formalize computational results, to verify correctness a substantial amount of classical analysis is needed. While there already is a large number of Coq libraries that deal with classical analysis, e.g. the Coquelicot [5] and MathComp Analysis [1] libraries, making use of them directly is challenging, as we define our own type of (computational) real numbers. Formalizing classical facts about real numbers and functions is therefore also a significant part of our Coq development. On one hand, this allows us to formalize the statements exactly in the way we need them, and make them integrate well with our computational theorems. On the other hand, we do not aim to provide a complete library for classical analysis and do not think that our classical results exceed what has already been proven in the above mentioned libraries. Thus, better integration with these existing formalizations is a goal for future work.

2 Classical Functions

Under our setting, the standard function type of real numbers $\mathbb{R} \rightarrow \mathbb{R}$ denotes a structured set of continuously realizable or computable real functions, and thus excludes any discontinuous classical function. However, often discontinuous classical functions play an important role in specifying and developing a meta-theory for computable procedures. For example, though the sign function of real numbers is not computable or continuous, it is used everywhere for specifying a computation on the signs of its real number inputs.

2.1 Classicalizing Monad

Recall the classical singleton subset monad which also appears in [18]:

Definition $\nabla (A : \text{Type}) := \{S : A \rightarrow \text{Prop} \mid \text{exists! } x : A, S x\}$

and its monad operations that are defined naturally. In this paper, we write $[x]$ for the monad unit on x and \tilde{A} as an abbreviation for ∇A . Notice that this monad acts as an eraser that erases the computational contents such that $A \rightarrow \tilde{B}$ denotes the set of classical functions from A to B . Though for different x and y we can prove $[x]$ and $[y]$ are distinct, both of the terms will be removed in program extraction. Note also that any classical function from constructive domain $f : A \rightarrow \tilde{B}$ can be extended to be a function from the classical domain $f^\dagger : \tilde{A} \rightarrow \tilde{B}$ by the monadic bind. It is noteworthy that this mapping $f \mapsto f^\dagger$ is a type-theoretic equivalence between $A \rightarrow \tilde{B}$ and $\tilde{A} \rightarrow \tilde{B}$.

As a simple sanity-check of our construction of the so-called *classicalizing monad*, we prove that under our set of axioms, any classical proposition $P : \text{Prop}$ is classical also in the sense that the monad unit $[\cdot] : P \rightarrow \tilde{P}$ admits an inverse. To construct the inverse, for any

30:6 Taylor Models and Power Series for ODE Solving in Coq

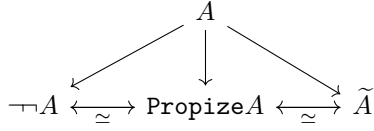
$t : \tilde{P}$, we destruct t to obtain its unique classical existential witness of $P : \text{Prop}$. Note that this construction is allowed because P is of type `Prop`. This construction defining the inverse follows straightforwardly from function extensionality and propositional extensionality.

Coq with our set of base axioms now admits three different ways to make a type $A : \text{Type}$ classical: (1) by reflecting the type as a classical proposition

Definition `Propize (A : Type) : Prop := exists _ : A, True`

(2) by applying the double negation $(\neg\neg A) : \text{Prop}$ (as a notation for $(A \rightarrow \text{False}) \rightarrow \text{False}$) and (3) by the classicalizing monad $\tilde{A} : \text{Type}$. An advanced sanity-check is to verify their equivalences on subsingletons.

► **Lemma 1.** *For any subsingleton type $A : \text{Type}$, \tilde{A} is again a subsingleton. Moreover, the three constructions of making A classical are equivalent:*



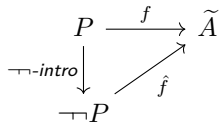
Proof. The type \tilde{A} being a subsingleton follows directly from the extensionality axioms. As the three classical types are all subsingleton, we only need to construct mappings between each types where mappings between $\neg\neg A$ and `Propize` A can be constructed trivially as both are of type `Prop`.

Given $t : \text{Propize } A$, define an element of type \tilde{A} with `fun x : A => True`. It being classically singleton follows from $t : \text{Propize } A$ and that A is a subsingleton. For its inverse, we lift the trivial mapping $A \rightarrow \text{Propize } A$ w.r.t. the classicalizing monad to get a mapping of type $\tilde{A} \rightarrow \widetilde{\text{Propize } A}$. Since `Propize` A is of type `Prop`, we apply the previous observation that there is an inverse `Propize` $A \rightarrow \text{Propize } A$ to the monad unit $[\cdot] : \text{Propize } X \rightarrow \widetilde{\text{Propize } X}$. Hence, post-composing this yields the desired inverse. ◀

Note that A being a subsingleton is necessary here because in the model when A is not a subsingleton, \tilde{A} is also not a subsingleton while $\neg\neg A$ and `Propize` A are.

The classicalizing monad being classical not only on subsingletons but for every types can be formalized in Coq as the following lemma:

► **Lemma 2.** *For any type $A : \text{Type}$ and a subsingleton type $P : \text{Type}$ there is a mapping that maps any $f : P \rightarrow \tilde{A}$ to $\hat{f} : \neg\neg P \rightarrow \tilde{A}$ such that the following diagram commutes:*



where `¬¬-intro` : $P \rightarrow \neg\neg P := \text{fun } x : P => \text{fun } f : P \rightarrow \text{False} => f x$ is the double negation introduction. Moreover, the mapping is a type-theoretic equivalence

$$(P \rightarrow \tilde{A}) \simeq (\neg\neg P \rightarrow \tilde{A})$$

where precomposing `¬¬-intro` is the inverse. I.e., for each $f : P \rightarrow \tilde{A}$ it holds that $f = \hat{f} \circ \neg\neg\text{-intro}$ and for each $g : \neg\neg P \rightarrow \tilde{A}$ it holds that $g = g \circ \neg\neg\text{-intro}$.

Proof. First, we construct the mapping \hat{f} of type $\neg\neg P \rightarrow \tilde{A}$ given $f : P \rightarrow \tilde{A}$ and that P is a subsingleton. We apply the monadic bind on f and get $f^\dagger : \tilde{P} \rightarrow \tilde{A}$ reducing the goal to constructing a term of type \tilde{P} from $\neg\neg P$. Let us write $j : \neg\neg P \rightarrow \tilde{P}$ for the mapping from Lemma 1. We conclude the construction with $\hat{f} := f^\dagger \circ j$.

Suppose any $f : P \rightarrow \tilde{A}$ and $x : P$. We prove the equality between $\hat{f}(\neg\neg\text{-intro } x) \equiv f^\dagger(j(\neg\neg\text{-intro } x))$ and $f x$. As P is a subsingleton type, due to Lemma 1, \tilde{P} is also a subsingleton type. Hence, we can prove that $j(\neg\neg\text{-intro } x) = [x]$ in \tilde{P} . That means we have $f^\dagger(j(\neg\neg\text{-intro } x)) = f^\dagger [x] = f x$ due to the monad axiom of ∇ .

Suppose any $g : \neg\neg P \rightarrow \tilde{A}$ and $x : \neg\neg P$. We need to prove $(g \circ \neg\neg\text{-intro})^\dagger(j x) = g x$. Since we are proving an equality as of type **Prop**, we can locate $y : P$ such that $x = \neg\neg\text{-intro } y$. Then, similarly to the previous case, $j x = [y]$. Therefore, $(g \circ \neg\neg\text{-intro})^\dagger(j x) = (g \circ \neg\neg\text{-intro})^\dagger([y]) = g(\neg\neg\text{-intro } y) = g x$ concludes the proof. ◀

A side note worth mentioning here is that thus the monad ∇ behaves like the double-negation sheafification in the setting of topos theory. As long as \tilde{A} is concerned, for a subsingleton P , the double negated classical version of it $\neg\neg P$ is equivalent to P itself.

For any subsingleton type P , the double negated excluded middle $\neg\neg(P + \neg P)$ is constructively provable and also is a subsingleton. Therefore, for any such P and a type $A : \mathbf{Type}$, one can construct a term of type \tilde{A} by a case distinction on P or $\neg P$. Let us write $\text{lem}(f) : \tilde{A}$ for this construction from $f : P + \neg P \rightarrow \tilde{A}$.

► **Example 3.** For a real number $x : \mathbb{R}$, define $f x : (x < 0) + \neg(x < 0) \rightarrow \widetilde{\text{bool}}$ by

```
f p := match p with
| inl _ => [false]
| inr _ => [true]
end.
```

that returns `[true]` when its input is an evidence that $x \geq 0$ and returns `[false]` when its input is an evidence that $x < 0$. Then, $(\text{fun } x : \mathbb{R} \Rightarrow \text{lem}(f x)) : \mathbb{R} \rightarrow \widetilde{\text{bool}}$ denotes the classical sign function.

Lemma 2 is effective enough to construct the reduction:

► **Example 4.** Suppose we have constructed $\text{lem}(f : P + \neg P \rightarrow \tilde{A}) : \tilde{A}$ but know either P holds or not by having a term $t : P + \neg P$. In this case, we can prove

$$\text{lem}(f) = f t$$

as a direct consequence of Lemma 2. Another important reduction case is when the value is irrelevant to the case distinction and is already known to be $y : \tilde{A}$. When we have that $f(\text{inl } t) = y$ for all $t : P$ and $f(\text{inr } t) = y$ for all $t : \neg P$, then we can obtain $\text{lem}(f) = y$. When we apply this to the classical sign function in Example 3, given $t : x < 0$ for example, we can obtain that the value of the sign function equals to `[false]`.

2.2 Classical Partial Functions

We define a classical partial function as

$$f : A \rightarrow \widetilde{B}_\perp$$

where B_\perp denotes option B in Coq, a maybe monad meaning that B_\perp is an inductive type with two constructors `Some` : $B \rightarrow B_\perp$ and `None` : B_\perp . We define some obvious operations on partial functions such as $f x \downarrow y$ when $y : B$ to denote $f x = [\text{Some } y]$ saying that $f x$ is defined to be y . We write $f x \downarrow$ for $\exists(y : Y). f x \downarrow y$.

An important way of constructing a classical partial function is from its classical graph or specification:

► **Lemma 5.** *Given a classical binary relation $G : A \rightarrow B \rightarrow \mathbf{Prop}$, such that for each $x : A$, $\{y : B \mid G \ x \ y\}$ is a subsingleton type, we can define the corresponding classical partial function $\hat{G} : A \rightarrow \widetilde{B}_\perp$ satisfying **forall** $(x : A) (y : B), f \ x \downarrow y \leftrightarrow \hat{G} \ x \ y$.*

Proof. Let us construct a mapping $\hat{G} : A \rightarrow \widetilde{B}_\perp$ from G by assuming any $x : A$ and constructing $y : \widetilde{B}_\perp$ such that there (classically) exists $y' : Y$ where $y \downarrow y'$. We apply Lemma 2 to the procedure of performing a case distinction on the constructive existence $\{y : B \mid G \ x \ y\}$, and returns the first projection $[\mathbf{Some} \ y]$ if there exists and returns $[\mathbf{None}]$ if not. This application is feasible as $\{y : B \mid G \ x \ y\}$ despite not being a classical proposition, is a subsingleton type. Proving that this resulting mapping satisfying the desired property is done by applying Example 4. ◀

2.3 Classical Partial Reals

Before concluding this section of introducing the formalization of classical partial functions, we present a special case when the classicalizing monad is applied to partial real numbers.

We define constants and arithmetical operations on classical partial reals $\widetilde{\mathbf{R}}_\perp$ naturally using the monad operations. Being natural here means that we can reason about the operations to be the exact values if and only if all the operands are defined.

An interesting operation that becomes possible using classical partial reals is the reciprocal operation where now we can define $0^{-1} = [\mathbf{None}]$ similarly to Example 3. Another interesting partial operation is the limit operation. Recall that the limit operation we have as primitive for computational \mathbf{R} has to be provided a proof that the given sequence is rapidly converging. However, since we can prove for any sequence that there classically exists at most one limit point, we can make the classical partial operation of type $(\mathbf{N} \rightarrow \widetilde{\mathbf{R}}_\perp) \rightarrow \widetilde{\mathbf{R}}_\perp$ to obtain limit classically for any converging sequences. We further extend the distance function $\text{dist} : \widetilde{\mathbf{R}}_\perp \rightarrow \widetilde{\mathbf{R}}_\perp \rightarrow \widetilde{\mathbf{R}}_\perp$, the absolute value function $\text{abs} : \widetilde{\mathbf{R}}_\perp \rightarrow \widetilde{\mathbf{R}}_\perp$, and so on.

Classical relations on classical partial real numbers are defined in a way that they fail when the partial real numbers are not defined. For example, we define $x < y$ on $\widetilde{\mathbf{R}}_\perp$ such that **exists** $x' \ y' : \mathbf{R}, x \downarrow x' \wedge y \downarrow y' \wedge x' < y'$. Of course, in the precise Coq formalization, we have to deal with the scopes but to simplify the presentation of this paper, we use the same function names and notations for both classical partial and exact real numbers as long as it is clear from the context or the formulation.

3 Classical Analysis: Continuity and Differentiability

We need a few more definitions from classical analysis to define the problems we are interested in. In particular, we need to define derivatives. We define classical (pointwise) continuity and differentiability closely to the standard definitions from analysis. However, in this work we only need those notions on compact intervals $[-r, r]$ around the origin. Classically, continuity and differentiability on compact domains correspond to their uniform versions. As the uniform versions are much easier to work with formally, we mostly formulate our results with respect to them. In our formal development we also show some results regarding pointwise continuity and differentiability, but we decided to omit them from this paper.

For any $r > 0$, we define the subset type $I \ r := \{x : \mathbf{R} \mid |x| \leq r\}$ with a coercion to \mathbf{R} by the first projection. We use the following definitions for (uniform) continuity and differentiability on $I \ r$ of a classical partial function $f : \mathbf{R} \rightarrow \widetilde{\mathbf{R}}_\perp$. Note that both properties automatically imply that f is defined on the interval.

```

Definition uniformly_continuous f r := forall  $\epsilon$ , ( $\epsilon > 0$ )  $\rightarrow$  exists  $\delta$ ,
 $\delta > 0 \wedge$  forall (x y : I r), dist x y  $\leq$   $\delta \rightarrow$  dist (f x) (f y)  $\leq$   $\epsilon$ .

```

```

Definition uniform_derivative f g r := forall  $\epsilon$ , ( $\epsilon > 0$ )  $\rightarrow$  exists  $\delta$ ,
 $\delta > 0 \wedge$  forall (x y : I r), dist x y  $\leq$   $\delta$ 
 $\rightarrow$  abs (f y - f x - g x * (y - x))  $\leq$   $\epsilon *$  abs (y - x).

```

We show some classical results, such as that every continuous function is bounded on $I r$ and that `uniform_derivative f f' r` implies that both f and f' are uniformly continuous (and thus bounded). We also show basic properties of derivatives, such as the sum, product and chain rules. Although formal proofs of those statements get quite lengthy, they are very similar to classical textbook proofs and we thus omit them from the paper.

We further define higher derivatives inductively:

```

Fixpoint nth_derivative f g r n :=
  match n with
  | 0  $\Rightarrow$  forall (x : I r), f x = g x
  | S n'  $\Rightarrow$  exists f', uniform_derivative f f' r  $\wedge$  nth_derivative f' g r n'
  end.

```

► Remark 6. By replacing classical existence by the constructive one, we can get constructive versions of the above definitions which can sometimes be useful for functions $f : \mathbb{R} \rightarrow \mathbb{R}$. We also show a few constructive statements in our Coq development, but as they are not needed for the results in this paper, we also decided to omit them here.

4 Polynomials and Taylor models

By the Stone-Weierstrass theorem, any continuous function on a closed interval can be uniformly approximated with arbitrarily small error (with respect to the supremum norm) by polynomials. In interval computation, a polynomial approximation together with a rigorous error bound is sometimes called a Taylor model [27]. Taylor models are usually used to approximate smooth functions using Taylor polynomials as the polynomial approximation (hence the name). Many mathematical operations (e.g. arithmetic, integrals, etc.) can be defined on Taylor models. When those operations are applied, dependencies between the variables are retained, giving tighter approximations for function values than simple intervals [26]. Our purpose for using Taylor models is slightly different from how they are used in interval computation. First, as we have exact real numbers in our formalization, we do not need to deal with intervals and can have a real-valued error bound. Secondly, we are mostly interested in sequences of Taylor models, that we use to approximate functions up to any absolute error, instead of mere approximations with fixed error bounds. Consequently, the operations we consider differ from what is usually considered in interval computation.

Before we use polynomials to approximate real valued functions, we need to define the polynomial functions themselves and some operations on them. We identify a polynomial with the list of its coefficients and define evaluation of the polynomial using Horner's scheme.

```

Definition poly := list R.
Fixpoint eval_poly (p : poly) (x : R) := match p with
  | []  $\Rightarrow$  0
  | h :: t  $\Rightarrow$  h + x * (eval_poly t x)
  end.

```

30:10 Taylor Models and Power Series for ODE Solving in Coq

That is, we identify the list $[a_0, a_1, \dots, a_d]$ with the polynomial function $\sum_{i=0}^d a_i x^i$, and the empty list with the zero polynomial. We allow the leading coefficient a_d to be zero, thus the length of the list can be larger than the actual degree of the polynomial. Nonetheless we sometimes use the notion $\text{deg } p$ for the length of the list. As checking if a real number is equal to zero is undecidable, requiring that the leading coefficient is nonzero would lead to rather complicated and unnatural domains for most operations. We further define the norm $\|p\| := \max |p_i|$ and prove some of its properties, which will be useful later.

For most inductive proofs, it is simpler to consider straight-forward evaluation, i.e., compute $\sum_{i=0}^d a_i x^i$ by directly summing up the terms of the sum. We therefore also define this evaluation method, and show that they result in the same number.

For polynomials p_1, p_2 and a real number $\lambda : \mathbb{R}$, we define arithmetic operations $p_1 + p_2$, $p_1 \cdot p_2$ and λp_1 . We currently do not use a sophisticated algorithm for multiplication. However, the statements are formulated in an abstract way, without explicit mention of the method:

```
Lemma mult_poly p1 p2 : {p3 | forall x, eval_poly p3 x = eval_poly p1 x * eval_poly p2 x}.
```

Thus, if we replace the algorithm encoded in the proof at some point, other parts of the formalization are not affected.

In this paper we are mostly concerned with results over compact intervals. For simplicity, we show most statements only with respect to intervals of the form $[-r, r]$ centered at 0. However, this easily generalizes to arbitrary intervals by shifting the polynomial accordingly:

```
Lemma shift_poly p1 c : {p2 | forall x, eval_poly p2 x = eval_poly p1 (x-c)}.
```

We often need to bound the values of a polynomial on intervals:

```
Lemma bound_polynomial p r : {B | forall x, abs x <= r -> abs (eval_poly p x) <= B}.
```

Again, the explicit bound is only encoded in the proof and not given in the statement, so that it can be replaced easily. For now, we use the simple bound $B := \sum_{i=0}^d |a_i| r^i$.

For any polynomial $\sum_{i=0}^d a_i x^i$, we can compute its derivative $\sum_{i=0}^{d-1} (i+1) a_{i+1} x^i$, which is the uniform derivative of the polynomial on any interval $[-r, r]$.

```
Lemma derive_poly p : {p' | forall r, uniform_derivative (eval_poly p) (eval_poly p') r}
```

As a corollary, we also get that polynomials are uniformly continuous.

A central tool to approximate functions by polynomials is Taylor's theorem. We define a polynomial p to be the Taylor polynomial for a function $f : \mathbb{R} \rightarrow \widetilde{\mathbb{R}}_{\perp}$ at $x_0 = 0$ as follows.

```
Definition is_taylor_polynomial p f r := forall n, (n < length p) ->
  (exists g, nth_derivative f g r n ^ nth n p 0 = inv_factorial n * (g 0)).
```

The following variant of Taylor's theorem will be useful later.

```
is_taylor_polynomial p f r M ->
  (exists g, nth_derivative f g r (length p) ^ (forall (x : I r), abs (g x) <= M))
  -> forall (x : I r), dist (eval_poly p x) (f x)
    <= inv_factorial (length p) * M * r ^ length p.
```

After having defined polynomials, we can now proceed to define Taylor models. A Taylor model for a function f consists of a polynomial p , a radius r and an error bound ϵ .

► **Definition 7.** We define a Taylor model $t : \text{tm } f$ for a classical partial function $f : \mathbb{R} \rightarrow \widetilde{\mathbb{R}}_{\perp}$ as a record consisting of a polynomial p_t , two real numbers $r_t, \epsilon_t : \mathbb{R}$ and a proof of the specification that $|f x - p_t x| \leq \epsilon_t$ for all x with $|x| \leq r_t$. The name of the field for r_t is `tm_radius` the name of the field for ϵ_t is `tm_error`.

Arithmetic operations on functions can be extended to their Taylor model representation. For example, we show

```
Definition sum_tm f1 f2 (t1 : tm f1) (t2 : tm f2) : tm (fun x => f1 x + f2 x).
Definition mult_tm f1 f2 (t1 : tm f1) (t2 : tm f2) : tm (fun x => f1 x * f2 x).
```

Proving those statements consists of two parts. The first is to define a polynomial that approximates the resulting function. Here, we can just use the corresponding operations on polynomials. The second is to define the radius and error bound. For both operations we can use the minimum of the two radii as the new radius. For addition we can simply add the error bounds. For multiplication we choose the error bound $\epsilon := B_1\epsilon_{t_1} + B_2\epsilon_{t_2} + \epsilon_1\epsilon_2$, where B_1 and B_2 are bounds for the polynomials p_{t_1} and p_{t_2} , respectively.

Note that we do not prune the degree of the resulting polynomial in our definition of multiplication. This means, however, that operating on Taylor models can increase the degree quickly, leading to performance issues. Thus, it might be necessary to reduce the degree at some point, for which we introduce the following operation.

```
Definition swipe_tm f (t : tm f) (m : N) : {t' : tm f | deg t' ≤ m}.
```

Here, $\deg t$ is defined as the length of the polynomial. The operation is performed by splitting the polynomial into two polynomials p_1 and p_2 such that $p_t x = (p_1 x) + x^m(p_2 x)$, compute a bound B for p_2 on the interval and “swiping” the bound into the error, i.e. defining the new Taylor model t' by the polynomial p_1 and error bound $\epsilon_{t'} = \epsilon_t + B$.

5 Exact Computation with Power Series

While Taylor models are interesting for computations with rigorous error bounds, we are mostly interested in representing certain functions exactly in the sense of computable analysis. That is, our main interest is to use sequences of polynomials that allow us to evaluate functions with any desired absolute precision. Formally, we will use a sequence of Taylor models that converges rapidly towards the function. That is, we say a sequence of Taylor models for a function f represents the function on $I r$ if all sequence elements have radius at least r and the n -th element has error at most 2^{-n} :

```
Definition represents f (t : nat → (tm f)) r :=
  forall n, (tm_error f (t n)) ≤ 2-n ∧ (tm_radius f (t n)) ≥ r.
```

Let us prove a few facts about functions represented in that way.

- **Lemma 8** (`represents_cont`). 1. *Any function that can be represented on $I r$ in the above sense is uniformly continuous on $I r$.*
 2. *Whenever we have representations for f and g , we can compute representations for $f + g$ and fg , respectively.*

Proof. To show the first claim, let $t : \mathbb{N} \rightarrow \mathbf{tm} f$ be a sequence of Taylor models that represents f and let $\epsilon > 0$ be arbitrary. By the Archimedean axiom there is some $n : \mathbb{N}$, such that $2^{-n} < \epsilon$. Choose the polynomial $p := t(n + 2)$ and recall that any polynomial is uniformly continuous. Thus, there is some $\delta > 0$, such that for all $x, y \in I r$, $|x - y| \leq \delta \rightarrow |(p x) - (p y)| \leq 2^{-(n+1)}$. But then $|(f x) - (f y)| = |(f x) - (p x) + (p y) - (f y) + (p x) - (p y)| < \epsilon$ holds.

To prove the second claim, we can simply perform the operations on the corresponding Taylor models and increase the index accordingly. For multiplication, this works as (by 1.) the function is bounded and thus the approximating polynomials can not grow much larger than that bound, showing that the error bound for the product of the approximating polynomials defined above gets arbitrarily small when increasing the index. ◀

30:12 Taylor Models and Power Series for ODE Solving in Coq

Now assume that we have a sequence of Taylor models t representing a function f and a sequence t' of derivatives of t . The notion behaves well with respect to differentiability, in the sense that whenever the sequence t' converges to some function f' , then f' is the derivative of f .

```

Lemma differentiable_limit f t f' t' r :
  (represents f t r) →
  (represents f' t' r) →
  (forall n, uniform_derivative_fun (eval_tm (t n)) (eval_tm (t' n))) r →
  uniform_derivative f f' r.

```

Here, `eval_tm` is the evaluation of the polynomial recorded in the Taylor model.

However, note that in general the sequence of derivatives of a representation does not need to be convergent. Thus, let us now focus on a large class of functions where we can use the theorem to compute the derivatives, namely functions analytic at 0.

Analytic functions allow a canonical approximation by polynomials in the following sense.

A function $f : \mathbb{R} \rightarrow \mathbb{R}$ is called analytic at 0 if f is locally defined by a power series, i.e., $f(x) = \sum a_k x^k$ on the interval $(-R, R)$ for some $R > 0$ called radius of convergence. Any analytic function is smooth and the power series coincides with the Taylor series.

We want to represent analytic functions locally by their power series and define a calculus of function operations on that representation. Before we do that, let us first consider infinite sequences $a : \mathbb{N} \rightarrow \mathbb{R}$ and series $\sum_{i=0}^{\infty} a_i$ in a general sense. Note that for brevity, we use a_k instead of $(a\ k)$ to denote the k -th element of a sequence $a : \mathbb{N} \rightarrow \mathbb{R}$.

For an infinite sequence $a : \mathbb{N} \rightarrow \mathbb{R}$ we define a partial sum operation in the obvious way:

```

Fixpoint partial_sum a n := match n with
  | 0 => (a 0)
  | (S n') => (a n) + partial_sum a n'
end.

```

We can then define that the infinite sum $\sum_{i=0}^{\infty} a_k$ classically exists.

```

Definition is_sum a x := forall  $\epsilon$ ,  $\epsilon > 0 \rightarrow$ 
  exists N, forall n, (n  $\geq$  N)  $\rightarrow$  dist (partial_sum a n) x  $\leq$   $\epsilon$ .

```

We show a few classical results about sums, such as that the sum of a sequence is unique. Further, when the sequence decreases rapidly, we can compute the sum.

Let us proceed to power series, that is infinite series of the form $\sum_{n=0}^{\infty} a_n x^n$. For any sequence $a : \mathbb{N} \rightarrow \mathbb{R}$ and $x : \mathbb{R}$, we define $\text{ps } a\ x := (\text{fun } n \Rightarrow a_n x^n)$. For any power series a , we define a canonical partial function f_a by $f_a\ x = y \leftrightarrow \text{is_sum } (\text{ps } a\ x)\ y$. On the other hand, we can also define a series to be the power series for some classical function g by

$$\text{is_ps_for } g\ a := \Leftrightarrow \exists (r > 0). \forall (x : I\ r). f_a\ x = g\ x.$$

To compute the value $\sum_{n=0}^{\infty} a_n x^n$, we need to approximate how close the partial sums are to the limit, which in general can not be computed from the series alone [36, Sect. 6.5]. However, if the power series corresponds to some function that is analytic at 0, there is some $R > 0$ such that the series converges for all x with $|x| < R$. This gives us the following fact.

► **Fact 1.** *Assume there is some $R > 0$ such that $\sum_{n=0}^{\infty} a_n x^n$ converges whenever $|x| < R$. Then there exist constants $M, r \in \mathbb{R}$ such that $|a_n| \leq M r^{-n}$ for all $n \in \mathbb{N}$.*

Proof. Choose any $0 < r < R$. Then the series converges absolutely on $[-r, r]$. In particular, there is some $M \in \mathbb{R}$ such that $\sum_{i=0}^{\infty} |a_n| r^n = M$. As all summands are positive, $|a_n| r^n \leq M$ must hold for all n and thus the claim holds. ◀

On the other hand, assume we are given constants M, r as in Fact 1. Then for any $x : \mathbb{R}$ with $|x| < r$ and any $N : \mathbb{N}$, we can get the tail estimate

$$\left| \sum_{n=N+1}^{\infty} a_n x^n \right| \leq M \sum_{n=N+1}^{\infty} \left(\frac{x}{r}\right)^n = M \frac{\left(\frac{x}{r}\right)^{N+1}}{1 - \frac{x}{r}}. \quad (1)$$

In particular, if we choose $|x| \leq \frac{r}{2}$, we get $\left| \sum_{n=N+1}^{\infty} a_n x^n \right| \leq M 2^{-N}$, or put differently the sequence `fun n => partial_sum (ps a x) (n + ⌈log M⌉)` defines a fast Cauchy sequence.

We call the constants M, r from Fact 1 a *series bound* for $(a_k)_{k \in \mathbb{N}}$ and encode power series together with the bound:

```
Record bounded_ps : Type := mk_bounded_ps
{
  series : N → R;
  bounded_ps_M : N;
  bounded_ps_r : R;
  bounded_ps_rgt0 : bounded_ps_r > 0;
  bounded_ps_bounded : forall n, abs (a n) ≤ bounded_ps_M * bounded_ps_r^{-n}.
}
```

Note that we chose r to be a real and M to be an integer. The main reason for that is that M is used to find the (integer valued) degree necessary to get a good approximation. If we chose M as real-valued, we could only compute an integer upper bound for $\log M$ non-deterministically, which would have made the statements slightly more complicated.

Using the tail bound from Equation (1), we can evaluate the power series on any $r' < r$. However, for simplicity we chose the fixed evaluation interval $[-\frac{r}{2}, \frac{r}{2}]$:

```
Lemma eval_val (a : bounded_ps) x :
  abs x ≤ (bounded_ps_r a) / 2 → {y | is_sum (ps series a x) y}.
```

To show that this is true, we show that computing the partial sum up to $n + \lceil \log M \rceil$ coefficients defines a fast Cauchy sequence, for which we compute the limit. Similarly, for any $n : \mathbb{N}$, we can canonically transform a bounded power series to a Taylor model for f_a with radius $\frac{r}{2}$ and error bound 2^{-n} . We denote this operation as `to_taylor_model`.

► **Remark 9.** The choice $\frac{r}{2}$ is somewhat arbitrary, but leads to simple bounds and evaluation is efficient in the region. While it would be possible to replace it by any $r' < r$, in most cases shifting the power series to a new center is more effective.

We next aim to define operations uniformly on power series, in a similar way as we did for Taylor models. In many cases, we can directly generalize results on Taylor models to results on power series by using the following lemma.

```
Lemma approx_ps f a :
  (forall n x, dist (eval_tm (to_taylor_model a n) x) (f x) ≤ 2^{-n})
  → is_ps_for f a.
```

Thus, whenever we define an operation on power series, we only need to show that the operation does what it is supposed to do on the Taylor model approximations. Let us demonstrate this by showing that we can add power series. Other, more complicated operations like multiplication can be done similarly.

```
Lemma sum_ps (a b : bounded_ps) : {c : bounded_ps | is_ps_for (f_a + f_b) c}.
```

Proof. We define the bounded power series c by adding the coefficients of a and b and defining $r_c := \min r_a r_b$ and $M_c := M_a + M_b$. It is easy to see that this choice is a valid bound for the series. To show that the series converges to the sum, we use `approx_ps`, thus

30:14 Taylor Models and Power Series for ODE Solving in Coq

we have to show that the distance of the 2^{-n} Taylor model approximation to $f_a + f_b$ is at most 2^{-n} . However, the Taylor model is identical to summing Taylor models for $2^{-(n+1)}$ approximations of f_a and f_b , thus the claim holds. ◀

The representation for power series further allows us to compute derivatives.

Lemma `derive_ps (a : bounded_ps) : {b : bounded_ps | uniform_derivative f_a f_b r_b}`.

Proof. Let us first show how to bound the power series $b := \sum_{n=0}^{\infty} (n+1)a_{n+1}x^n$ of the derivative. We choose $r_b := \frac{r_a}{2}$. Then

$$|b_n| \leq (n+1)M_a r_a^{-(n+1)} = (n+1)2^{-n}(M_a r_a)r_b^{-n} \leq (M_a r_a)r_b^{-n}.$$

Thus taking any $M_b > M_a r_a$ works. To show that the power series converges to the derivative of f_a , it suffices to show that the sequence of Taylor models of b is a sequence of derivatives of the Taylor models for a . ◀

6 Ordinary Differential Equations

Let us now consider initial value problems (IVPs) for autonomous ordinary differential equations of the form $\dot{y} = f(y(t)); y(0) = y_0$. We call f the right-hand side function, y_0 the initial value and y the solution of the IVP. It is well known that an initial value problem with analytic right-hand side function has an analytic solution. For simplicity we currently only consider polynomial right-hand side, which already includes many interesting applications. With a few minor adaptations, the same method works for general analytic functions. Note that as polynomials are analytic functions, the solution will again be analytic. However, in general the solution does not need to be polynomial.

In numerics, so-called single step methods such as Euler's method or Runge-Kutta methods are commonly used to solve initial value problems for ordinary differential equations. These methods approximate the solution by taking one step at a time from the current point to the next point, using information from the current point to approximate the solution at the next point. To approximate this solution, often polynomial approximations of a fixed degree are used, and to improve the accuracy the step size is reduced. However, such a method does usually not work well for high precisions as the number of steps grows exponentially with the precision. In exact real computation therefore a different method is more applicable [16, 8]. In this section we show how to implement a variant of this method in our formal development. The method is similar to a single step method, but instead of varying the step size for higher accuracy, we compute a local power series for the solution valid on a small interval around the current point. We can thus keep the step size fixed and get more accurate approximations by using more terms of the power series. This also integrates well with the tools in our library, as the evaluation of the power series can be defined by the limit operation as described in the previous section.

Let us first define a function to be the solution of a polynomial IVP as follows.

Definition `pivp_solution p y y0 r := (y 0) = y0 ∧ uniform_derivative y (fun t => (eval_poly p (y t))) r`.

That is, `pivp_solution p y y0 r` says that the function $y : \mathbb{R} \rightarrow \mathbb{R}$ is a solution to the polynomial IVP $\dot{y} = p(y(t)); y(0) = y_0$ on the interval $(-r, r)$.

It further suffices to only consider the case $y_0 = 0$, as we can always transform a general IVP to an IVP with initial value 0 by replacing $y(t)$ by $y(t) - y_0$ and $p(x)$ by $p(x + y_0)$. Also note that we can always assume that we start at time $t = 0$, as the ODE is autonomous.

Note that, although classically true, we have not formalized the proof of existence and uniqueness of the solution in our Coq development yet. As the focus of this work is to compute the solution, and thus the classical proof is less important, currently our formal statements include the additional assumption that the solution exists, which we plan to remove later. This is, however, irrelevant for the extracted programs as this non-computational assumption is removed in the extracted code anyway.

A well-known method to solve an IVP is the so-called Taylor series method (see e.g. [31]). The idea behind the method is to automatically generate the coefficients of the power series for the solution $y(t)$. To this end, let us define the following sequence of polynomials.

```
Fixpoint pn p n := match n with
  | 0 => p
  | S n' => n-1 * (p * (derive_poly (pn p n')))
end.
```

We show that for each n , the polynomial $(pn p n)$ applied to $y(t)$ gives the $(n+1)$ -st derivative of the solution function y divided by $(n+1)!$:

```
pivp_solution p y 0 r → nth_derivative y (fun t => (n+1)! * ((pn p n) (y t))) r (n+1)
```

The proof is by induction and is essentially an application of the chain rule for derivatives. It follows that evaluating the polynomial at $y_0 = 0$ gives the coefficients of the Taylor series of the solution:

```
Definition yn p n := match n with
  | 0 => 0
  | S n' => (eval_poly (pn p n') 0)
end.
```

```
Lemma pivp_ps_taylor_series p y r : pivp_solution p y 0 r →
  forall n, (is_taylor_polynomial (to_poly (yn p) n) y r).
```

However, the error bounds for the truncated Taylor series depend on the range of $y(t)$, which we do not know yet. In interval arithmetic, estimating the value of the solution $y(t)$ for an IVP thus usually consists of two steps. The first step is to find a coarse a priori enclosure for $y(t)$ on some interval. This bound can then be used in a second step to find a tighter bound for $y(t)$, e.g., by using it for the error bound in the power series method. Finding a good a priori enclosure is important, as it essentially determines the step size that can be used in the algorithm. An often used simple but effective method is the high order enclosure method [10]. However, its correctness depends on an application of Banach's fixed points theorem, and a formal proof seems challenging. We propose an alternative method, by bounding the coefficients of the Taylor series. Note, however, that a more sophisticated approach would give us better bounds, and thus should be considered in the future. We show

► **Lemma 10** (`yn_bound`). *Let $p = [a_1, \dots, a_d]$ be a polynomial. For all $n : \mathbb{N}$, we get the bound $|(yn p n)| \leq (d^2 \|p\|)^n$.*

Proof. Let $p = [a_1, \dots, a_d]$ and recall that $\|p\| = \max |a_k|$. For any polynomials p, q it holds $\|pq\| \leq \deg p \|p\| \|q\|$ and $\|\text{derive_poly } p\| \leq \deg p \|p\|$. We show for all n , $\|pn p n\| \leq (d^2 \|p\|)^{n+1}$ from which the claim follows. The proof is by induction on n . For $n = 0$, we get

30:16 Taylor Models and Power Series for ODE Solving in Coq

$\|p\| \leq d^2 \|p\|$ which holds trivially. Now assume $\|\text{pn } p \ n\| \leq (d^2 \|p\|)^{n+1}$. We have to show

$$\left\| \frac{1}{n+1} p \cdot (\text{derive_poly } (\text{pn } p \ n)) \right\| \leq (d^2 \|p\|)^{n+2}.$$

We can further show that $\text{deg } (\text{pn } p \ n) = (n+1)d - 2n$. Thus we get

$$\left\| \frac{1}{n+1} p \cdot (\text{derive_poly } (\text{pn } p \ n)) \right\| \leq \frac{1}{n+1} \|p\| (n+1) d^2 \|(\text{pn } p \ n)\|$$

from which the claim follows. ◀

Thus, putting the above results together, `yn` defines a bounded power series for the solution y with bounds $r = \frac{1}{(d^2 \|p\|)}$ and $M = 1$.

```
Definition pivp_ps_exists p :
  {a:bounded_ps | exists r, r > 0 ∧ forall y, pivp_solution p y y0 r → is_ps_for (y-y0) a}.
```

Note that this also shows that the resulting function is analytic, as we can compute its power series and a positive lower bound on its radius of convergence. We can then use this bounded power series to compute a local solution $y(t)$ for some small $t > 0$. An important property of this method is that we do not need to evaluate the function on additional steps to increase the precision. More precisely, the precision is controlled by applying the limit operator on the sequence of partial sums of the power series at the point t . Thus, we get an exact representation of the number $y(t)$ as a converging sequence of values that only depend on the power series at 0. In contrast, when using the Euler or Runge-Kutta methods, the function needs to be evaluated on additional points to increase the precision, and the methods are therefore harder to adapt to our system.

Next, we want to extend the solution to a larger interval. To this end, we first define an operation that gives us one specific pair $(t_1, y(t_1))$ with $y_1 > 0$ from the local solution.

```
Lemma local_solution p y0 :
  {ty1 : R * R | (fst ty) > 0 ∧
    exists r, r > 0 ∧ forall y, pivp_solution p y y0 r → (snd ty1) = (y (fst ty1))}.
```

The method works by simply applying the evaluation algorithm for the power series on the largest possible point in the evaluation interval and adding y_0 .

► **Example 11.** Consider the initial value problem

$$\dot{y}(t) = 1 + y(t)^2 ; y(0) = y_0.$$

It has the explicit solution $y(t) = \tan(t + \arctan y_0)$. From the term `(local_solution [1 0 1] y0)` we can extract a program that computes the pair (t_1, y_1) . The extracted program proceeds as follows. The first step is to transform the system to an equivalent one with initial value 0. Application of the shifting procedure yields the new system $\dot{y}(t) = p(y(t)) ; y(0) = 0$ with $p(x) = 1 + y_0^2 + 2y_0x + x^2$. The operator then computes the power series of the local solution. We get the evaluation interval $\frac{1}{8\|p\|}$ where $\|p\| = \max(1 + y_0^2) (2y_0)$ for the power series, which is a rather coarse under-approximation of the actual radius of convergence $\frac{\pi}{2} - \arctan y_0$ of the series. The solution operator will produce the value and time for this maximal point in the evaluation domain.

Next, we can turn this into a single step method to solve the IVP on a larger interval by repeatedly applying the local solution operator to get a new initial value.

That is, we prove the following statement.

```

Lemma solve_ivp p y0 n : {l : list (R * R) |
  length l = n+1 ∧
  forall m, m < n → (fst (nth m l (0,0)) < fst (nth (m+1) l (0,0)))
  forall y r, pivp_solution p y y0 r → forall ty, In ty l → (snd ty) = (y (fst ty))}.

```

The method produces a list $[(t_0, y_0), \dots, (t_n, y_n)]$ such that the t_0 are increasing and $y(t_i) = y_i$. The step size is adaptive as when the growth of the function is faster in a region, the evaluation radius we compute gets smaller. Note that each $y(t_i)$ is given as an exact representation of the number and thus the step size only depends on parameters of the function and the initial value and not on the desired precision of the solution. Thus, we do not need to consider error propagation due to working with approximations, allowing for a simple proof of correctness.

If we consider again Example 11, we can see that starting with $y_0 = 0$, in each step the interval will get smaller and smaller as we approach $\frac{\pi}{2}$.

7 Conclusion and Future Work

We presented an extension of our formalization of exact real computation to include rigorous approximations of classical partial functions by polynomials, analytic functions and solutions to initial value problems for non-linear polynomial ordinary differential equations. One of the main limitations of the current work is that we only consider univariate functions, while most interesting problems for differential equations occur at higher dimensions. Technically, all the algorithms presented in this paper generalize directly to the multivariate setting and even certain forms of partial differential equations [34]. Extending the formalization should therefore not be too challenging, although it might require formalizing additional results from multivariate analysis. Another possible extension for which one dimensional functions suffice, is to consider holonomic functions. A function is holonomic if it is the solution to a linear differential equation of the form $p_r(x)f^{(r)}(x) + p_{r-1}(x)f^{(r-1)}(x) + \dots + a_1(x)f'(x) + a_0(x)f(x) = 0$ where p_0, \dots, p_r are polynomials. Similar to the analytic case, it is possible to compute the power series of solutions to such equations [15] and thus it might be simpler to include in our formalization than extending to higher dimension.

Lastly, many of the algorithms currently encoded in the proofs are rather simple and not very efficient. Replacing them by more sophisticated algorithms would yield better extracted programs. In particular, the bound we currently use for the radius of convergence of an ODE solution is a very coarse approximation, and improving this bound would increase the step size and thus the efficiency of the algorithm drastically. Thus, we plan to verify better methods such as the higher-order enclosure method [10] in future work. Due to the abstract formulation of most results, exchanging algorithms in one part should have little effect on other parts of the formalization, allowing to replace those methods easily in the future.

References

- 1 Reynald Affeldt, Yves Bertot, Cyril Cohen, Marie Kerjean, Assia Mahboubi, Damien Rouhling, Pierre Roux, Kazuhiko Sakaguchi, Zachary Stone, Pierre-Yves Strub, et al. *Mathcomp-analysis: Mathematical components compliant analysis library*, 2022.
- 2 Rajeev Alur, Costas Courcoubetis, Thomas A. Henzinger, and Pei Hsin Ho. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In Robert L. Grossman, Anil Nerode, Anders P. Ravn, and Hans Rischel, editors, *Hybrid Systems*, pages 209–229, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg.

- 3 A. Balluchi, A. Casagrande, P. Collins, A. Ferrari, T. Villa, and A.L. Sangiovanni-Vincentelli. Ariadne: a Framework for Reachability Analysis of Hybrid Automata. In *Proc. 17th Int. Symp. on Mathematical Theory of Networks and Systems*, Kyoto, 2006.
- 4 Errett Bishop and Douglas Bridges. *Constructive analysis*, volume 279. Springer Science & Business Media, 2012.
- 5 Sylvie Boldo, Catherine Lelay, and Guillaume Melquiond. Coquelicot: A user-friendly library of real analysis for Coq. *Mathematics in Computer Science*, 9(1):41–62, 2015. doi:10.1007/s11786-014-0181-1.
- 6 Franz Brauße, Pieter Collins, and Martin Ziegler. Computer science for continuous data. In François Boulter, Matthew England, Timur M. Sadykov, and Evgenii V. Vorozhtsov, editors, *Computer Algebra in Scientific Computing*, pages 62–82, Cham, 2022. Springer International Publishing.
- 7 Franz Brauße, Margarita Korovina, and Norbert Müller. Using taylor models in exact real arithmetic. In *Revised Selected Papers of the 6th International Conference on Mathematical Aspects of Computer and Information Sciences - Volume 9582*, MACIS 2015, pages 474–488, Berlin, Heidelberg, 2015. Springer-Verlag. doi:10.1007/978-3-319-32859-1_41.
- 8 Franz Brauße, Margarita Korovina, and Norbert Th Müller. Towards using exact real arithmetic for initial value problems. In *Perspectives of System Informatics: 10th International Andrei Ershov Informatics Conference, PSI 2015, in Memory of Helmut Veith, Kazan and Innopolis, Russia, August 24-27, 2015, Revised Selected Papers 10*, pages 61–74. Springer, 2016. doi:10.1007/978-3-319-41579-6_6.
- 9 Xin Chen, Erika Ábrahám, and Sriram Sankaranarayanan. Flow*: An analyzer for non-linear hybrid systems. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification*, pages 258–263, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.
- 10 George F Corliss and Robert Rihm. Validating an a priori enclosure using high-order taylor series. *Mathematical Research*, 90:228–238, 1996.
- 11 Luís Cruz-Filipe, Herman Geuvers, and Freek Wiedijk. C-CoRN, the constructive Coq repository at Nijmegen. In *International Conference on Mathematical Knowledge Management*, pages 88–103. Springer, 2004. doi:10.1007/978-3-540-27818-4_7.
- 12 Laurent Doyen, Goran Frehse, George J. Pappas, and André Platzer. *Verification of Hybrid Systems*, pages 1047–1110. Springer International Publishing, Cham, 2018. doi:10.1007/978-3-319-10575-8_30.
- 13 Fabian Immler. *A Verified ODE Solver and Smale’s 14th Problem (Ein Verifizierter GDGL-Löser und Smales 14. Problem)*. PhD thesis, Technical University of Munich, Germany, 2018. URL: <https://mediatum.ub.tum.de/1422071>.
- 14 Fabian Immler. A verified ODE solver and the lorenz attractor. *J. Autom. Reason.*, 61(1-4):73–111, 2018. doi:10.1007/S10817-017-9448-Y.
- 15 Manuel Kauers. *D-finite Functions*, volume 30. Springer Nature, 2023.
- 16 Akitoshi Kawamura, Florian Steinberg, and Holger Thies. Parameterized complexity for uniform operators on multidimensional analytic functions and ODE solving. In *International Workshop on Logic, Language, Information, and Computation*, pages 223–236. Springer, 2018. doi:10.1007/978-3-662-57669-4_13.
- 17 Michal Konečný, Sewon Park, and Holger Thies. Axiomatic reals and certified efficient exact real computation. In *International Workshop on Logic, Language, Information, and Computation*, pages 252–268. Springer, 2021. doi:10.1007/978-3-030-88853-4_16.
- 18 Michal Konečný, Sewon Park, and Holger Thies. Certified computation of nondeterministic limits. In *NASA Formal Methods Symposium*, pages 771–789. Springer, 2022. doi:10.1007/978-3-031-06773-0_41.
- 19 Michal Konečný, Sewon Park, and Holger Thies. Extracting efficient exact real number computation from proofs in constructive type theory. *arXiv preprint arXiv:2202.00891*, 2022.
- 20 Michal Konečný, Sewon Park, and Holger Thies. cAERN: Axiomatic Reals and Certified Efficient Exact Real Computation. <https://github.com/holgerthies/coq-aern>, 2024.

- 21 Michal Konečný, Sewon Park, and Holger Thies. Formalizing Hyperspaces for Extracting Efficient Exact Real Computation. In Jérôme Leroux, Sylvain Lombardy, and David Peleg, editors, *48th International Symposium on Mathematical Foundations of Computer Science (MFCS 2023)*, volume 272 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 59:1–59:16, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.MFCS.2023.59.
- 22 Michal Konečný. aern2-real: A Haskell library for exact real number computation. <https://hackage.haskell.org/package/aern2-real>, 2021.
- 23 Soonho Kong, Sicun Gao, Wei Chen, and Edmund Clarke. dreach: δ -reachability analysis for hybrid systems. In Christel Baier and Cesare Tinelli, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 200–205, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.
- 24 Assia Mahboubi, Guillaume Melquiond, and Thomas Sibut-Pinote. Formally verified approximations of definite integrals. *J. Autom. Reason.*, 62(2):281–300, 2019. doi:10.1007/S10817-018-9463-7.
- 25 Evgeny Makarov and Bas Spitters. The picard algorithm for ordinary differential equations in coq. In Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie, editors, *Interactive Theorem Proving*, pages 463–468, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. doi:10.1007/978-3-642-39634-2_34.
- 26 Kyoko Makino and Martin Berz. Efficient control of the dependency problem based on taylor model methods. *Reliable Computing*, 5(1):3–12, 1999. doi:10.1023/A:1026485406803.
- 27 Kyoko Makino and Martin Berz. Taylor models and other validated functional inclusion methods. *International Journal of Pure and Applied Mathematics*, 6:239–316, 2003.
- 28 Érik Martin-Dorel, Laurence Rideau, Laurent Théry, Micaela Mayero, and Ioana Pasca. Certified, efficient and sharp univariate taylor models in coq. In *Proceedings of the 2013 15th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC '13*, pages 193–200, USA, 2013. IEEE Computer Society. doi:10.1109/SYNASC.2013.33.
- 29 Norbert Th Müller. Constructive aspects of analytic functions. In *Proc. Workshop on Computability and Complexity in Analysis*, volume 190, pages 105–114, 1995.
- 30 Norbert Th Müller. The iRRAM: Exact arithmetic in C++. In *International Workshop on Computability and Complexity in Analysis*, pages 222–252. Springer, 2000. doi:10.1007/3-540-45335-0_14.
- 31 Nedialko S Nedialkov. Interval tools for odes and daes. In *12th GAMM-IMACS International Symposium on Scientific Computing, Computer Arithmetic and Validated Numerics (SCAN 2006)*, pages 4–4. IEEE, 2006.
- 32 Sewon Park, Franz Brauße, Pieter Collins, SunYoung Kim, Michal Konečný, Gyesik Lee, Norbert Müller, Eike Neumann, Norbert Preining, and Martin Ziegler. Semantics, specification logic, and hoare logic of exact real computation, 2024. arXiv:1608.05787.
- 33 Arno Pauly and Florian Steinberg. Comparing representations for function spaces in computable analysis. *Theory Comput. Syst.*, 62(3):557–582, 2018. doi:10.1007/S00224-016-9745-6.
- 34 Svetlana Selivanova, Florian Steinberg, Holger Thies, and Martin Ziegler. Exact real computation of solution operators for linear analytic systems of partial differential equations. In *Computer Algebra in Scientific Computing: 23rd International Workshop, CASC 2021, Sochi, Russia, September 13–17, 2021, Proceedings 23*, pages 370–390. Springer, 2021. doi:10.1007/978-3-030-85165-1_21.
- 35 Christoph Traut and Fabian Immler. Taylor models. *Arch. Formal Proofs*, 2018, 2018. URL: https://www.isa-afp.org/entries/Taylor_Models.html.
- 36 K. Weihrauch. Computable analysis. Springer, Berlin, 2000. doi:10.1007/978-3-642-56999-9.

A Verified Earley Parser

Martin Rau ✉ 

Department of Computer Science, Technical University of Munich, Germany

Tobias Nipkow 

Department of Computer Science, Technical University of Munich, Germany

Abstract

An Earley parser is a top-down parsing technique that is capable of parsing arbitrary context-free grammars. We present a functional implementation of an Earley parser verified using the interactive theorem prover Isabelle/HOL. Our formalization builds upon Cliff Jones' extensive, refinement-based paper proof. We implement and prove soundness and completeness of a functional recognizer modeling Jay Earley's original imperative implementation and extend it with the necessary data structures to enable the construction of parse trees following the work of Elizabeth Scott. Building upon this foundation, we develop a functional parser and prove its soundness. We round off the paper by providing an informal argument and empirical data regarding the running time and space complexity of our implementation.

2012 ACM Subject Classification Software and its engineering → Formal software verification; Software and its engineering → Parsers

Keywords and phrases Verification, Parsers, Earley, Isabelle

Digital Object Identifier 10.4230/LIPIcs.ITP.2024.31

1 Introduction

Parsing is fundamental. Nearly every application interacts with its environment, and usually by means of parsing textual input into a structured data format. In the age of big data, applications handle enormous amounts of data and any parser bugs or vulnerabilities entail severe security risks. Although the semantics of a parser are relatively easy to specify, correctly implementing a parser is a difficult task. Attackers regularly exploit parsing bugs to obtain sensitive user data [1, 3, 2]. Hence, parsing algorithms are well-suited for formal verification which allows us to precisely specify the semantics of a parser and obtain strong correctness guarantees.

A zoo of parsing algorithms exists, and one of the core trade-offs one has to make when deciding on a parser is between performance and usability. Earley [12] parsing, originally conceived by Jay Earley in 1968, is an algorithm that allows the full range of context-free grammars while still being very performant for a large subset. In this paper, we present the, to our knowledge, first formalization of an Earley parser. Our formalization builds upon Cliff Jones' [20] extensive, refinement-based paper proof.

Section 2 shortly introduces Isabelle/HOL [29, 28]. Section 3 contains the formalization of context-free grammars and derivations. Section 4 defines and proves correct an inductive definition of an Earley recognizer. Section 5 refines this definition to an executable algorithm. Section 6 extends the recognizer to a parser. Section 7 contains an analysis of the running time. Sections 8 and 9 discuss related work and conclude.

The whole formalization, including all proofs, can be found online in the Archive of Formal Proofs [33]. The size of the formalization (more than 6000 lines) prohibits a detailed exposition in this paper, especially of the proofs. The interested reader is referred to the online material.



© Martin Rau and Tobias Nipkow;

licensed under Creative Commons License CC-BY 4.0

15th International Conference on Interactive Theorem Proving (ITP 2024).

Editors: Yves Bertot, Temur Kutsia, and Michael Norrish; Article No. 31; pp. 31:1–31:18

Leibniz International Proceedings in Informatics



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

2 Isabelle/HOL

Isabelle [29, 28] is an interactive theorem prover based on a fragment of higher-order logic. It supports core concepts commonly found in functional programming languages.

The notation $t :: \tau$ indicates that term t has type τ . Basic types include *bool* and *nat*, while type variables are written *'a*, *'b* etc. Pairs are expressed as (a, b) , and triples as (a, b, c) , and so forth. Functions *fst* and *snd* return the first and second components of a pair, while the operator (\times) is used for pairs at the type level. Most type constructors are written postfix, such as *'a set* and *'a list*, and the function space arrow is \Rightarrow . Function *set* converts a list to a set.

Algebraic data types are defined using the **datatype** keyword. Non-recursive definitions are introduced with the **definition** keyword. Lists are constructed from the empty list $[]$ using the infix cons-operator $(\#)$. The operator $(@)$ appends two lists, $|xs|$ denotes the length of xs , $xs ! n$ returns the n -th item of the list xs (starting with $n = 0$), and $xs[i := x]$ returns an updated list by setting the n -th item to the value x .

3 Context-free Grammars and Derivations

A symbol, either non-terminal or terminal, is represented as an arbitrary type *'a*. We use lowercase letters a, b, c to denote terminals and capital letters A, B, C to denote non-terminals. Additionally, we use the letters s, t to represent arbitrary symbols. A *sentence* is defined as a list of symbols and can be represented by either Greek letters α, β, γ or lowercase letters u, v, w .

The data type *cfg* represents context-free grammars. An instance \mathcal{G} comprises a list of production rules $\mathfrak{R} \mathcal{G}$, where each rule is a pair consisting of a left-hand side, *lhs_rule*, a single symbol, and a right-hand side, *rhs_rule*, a list of symbols. Additionally, the instance \mathcal{G} contains the start symbol $\mathfrak{S} \mathcal{G}$.

We formalize the set of non-terminals as the union of all left-hand sides of a grammar's production rules and its start symbol. A *word* is a sentence that consists only of terminal symbols, meaning *is_word* $\mathcal{G} \omega = (\text{nonterminals } \mathcal{G} \cap \text{set } \omega = \emptyset)$. The empty word is denoted by $[]$.

Given a grammar \mathcal{G} , the sentence β can be derived from the sentence α in a single step, denoted by $\mathcal{G} \vdash \alpha \Rightarrow \beta$, if \mathcal{G} contains a production rule (A, γ) such that α is of the form $u @ [A] @ v$ and $\beta = u @ \gamma @ v$. Defining *derivations* $\mathcal{G} = \{(\alpha, \beta) \mid \mathcal{G} \vdash \alpha \Rightarrow \beta\}^*$ we abbreviate $(\alpha, \beta) \in \text{derivations } \mathcal{G}$ by $\mathcal{G} \vdash \alpha \Rightarrow^* \beta$.

Some of the core proofs of this work make use of an analogous formalization of derivations. The term *Derivation* $\mathcal{G} \alpha D \beta$ signifies that the grammar \mathcal{G} allows the sentence β to be derived from the sentence α via the *derivation* D . In this context, D is a list containing pairs of production rules and indices, which constitute the specific rewriting steps. When applied in sequence to α , these steps lead to β . Both definitions of derivations are indeed equivalent, meaning $\mathcal{G} \vdash \alpha \Rightarrow^* \beta$, if and only if, there exists a derivation D such that the predicate *Derivation* $\mathcal{G} \alpha D \beta$ holds. We omit the proof.

4 Defining the Set of Earley Items

An Earley recognizer determines whether the input ω is in the language defined by the grammar \mathcal{G} by following a two-step process: first, it generates a set of items, then it checks if there exists a *finished* item. In the following, we consider a fixed grammar \mathcal{G} and input ω .

An item takes the form $Item\ r\ d\ i\ j$, which consists of four components: a production rule r from the grammar \mathcal{G} , referred to as the *rule_item*, a natural number d , or *dot_item*, marking how far the algorithm has processed the right-hand side of r , and two natural numbers i, j , *start_item* and *end_item*, representing the start index and the end index (exclusive) of the sublist of the input ω recognized by the item. Alternatively, an item with a production rule $A \rightarrow \alpha\beta$, which recognizes the subsequence of the input from index i up to but excluding j by processing α , is written $A \rightarrow \alpha \bullet \beta, i, j$.

The functions *lhs_item* and *rhs_item* project the *lhs_rule* and *rhs_rule* of an item. Functions *alpha_item* and *beta_item* split the production rule body at the position of the dot. An item is complete, *is_complete*, when the dot is at the end of the production rule body, as in $A \rightarrow \alpha \bullet, i, j$. The *next_symbol* of an item can either be *None*, if it is complete, or *Some s*, where s is the symbol in the production rule body following the dot.

An item x is well-formed, *wf_item*, if the item's rule belongs to the grammar \mathcal{G} , the item dot must be within the length of the item's right-hand side, the item start does not exceed the item end, and finally, the item end must be at most the length of the input ω .

An item is finished, *is_finished*, if it is of the form $\mathfrak{S}\ \mathcal{G} \rightarrow \alpha \bullet, 0, |\omega|$, meaning the left-hand side of the item is the start symbol of the grammar \mathcal{G} , the item is complete, and the entire input ω has been recognized; or the item start is zero, and the item end is the length of ω .

The set of *Earley items*, $Earley\ \mathcal{G}\ \omega$, is an inductive definition of Earley's recognizer, i.e. an inductively defined set. The four defining rules are: the initial set of items, and one rule for each of the core operations that expand the set of items: scanning, prediction, and completion.

$$\frac{(\mathfrak{S}\ \mathcal{G}, \alpha) \in set(\mathfrak{R}\ \mathcal{G})}{\mathfrak{S}\ \mathcal{G} \rightarrow \bullet \alpha, 0, 0 \in Earley\ \mathcal{G}\ \omega} \text{INIT}$$

$$\frac{A \rightarrow \alpha \bullet a\beta, i, j \in Earley\ \mathcal{G}\ \omega \quad \omega ! j = a \quad j < |\omega|}{A \rightarrow \alpha a \bullet \beta, i, j + 1 \in Earley\ \mathcal{G}\ \omega} \text{SCAN}$$

$$\frac{A \rightarrow \alpha \bullet B\beta, i, j \in Earley\ \mathcal{G}\ \omega \quad (B, \gamma) \in set(\mathfrak{R}\ \mathcal{G})}{B \rightarrow \bullet \gamma, j, j \in Earley\ \mathcal{G}\ \omega} \text{PREDICT}$$

$$\frac{A \rightarrow \alpha \bullet B\beta, i, j \in Earley\ \mathcal{G}\ \omega \quad B \rightarrow \gamma \bullet, j, k \in Earley\ \mathcal{G}\ \omega}{A \rightarrow \alpha B \bullet \beta, i, k \in Earley\ \mathcal{G}\ \omega} \text{COMPLETE}$$

The *Init* rule specifies all initial items $\mathfrak{S}\ \mathcal{G} \rightarrow \bullet \alpha, 0, 0$. There is one item for each grammar rule that begins with the grammar's start symbol. For these items, the dot, start, and end indices are all initialized to 0. This signifies that we haven't processed the right-hand side of the rule at all, started the recognition process at the beginning of the word, and still are at this initial position.

The *Scan* rule applies if there is a terminal symbol to the right of the dot: $A \rightarrow \alpha \bullet a\beta, i, j$. In this case, if the j -th symbol of ω is the *next_symbol* of the item, we add a new item $A \rightarrow \alpha a \bullet \beta, i, j + 1$, moving the dot over the recognized terminal symbol.

The *Predict* rule is applicable to an item when there is a non-terminal symbol to the right of the dot: $A \rightarrow \alpha \bullet B\beta, i, j$. It adds a new item $B \rightarrow \bullet \gamma, j, j$ for each production rule (B, γ) of the grammar. Similar to the initial items, the dot is set to 0, but the start and end indices are set to j to indicate that we are beginning recognition at position j in the input ω .

The *Complete* rule is applied to all complete items $B \rightarrow \gamma \bullet, j, k$. These items indicate successful recognition of a subsequence of ω starting at index j and ending at index k . Now, we consider any items where we already predicted the non-terminal symbol B . Specifically,

we look for items $A \rightarrow \alpha \bullet B\beta, i, j$ with a matching end index j and a dot in front of the non-terminal B . Since we have successfully recognized the predicted non-terminal, we are allowed to move the dot, resulting in the addition of a new item $A \rightarrow \alpha B \bullet \beta, i, k$.

We will prove soundness and completeness of *Earley*:

1. Soundness: If $x \in \text{Earley } \mathcal{G} \ \omega$ and $\text{is_finished } \mathcal{G} \ \omega \ x$, then $\mathcal{G} \vdash [\mathfrak{S} \ \mathcal{G}] \Rightarrow^* \omega$.
2. Completeness: If $\mathcal{G} \vdash [\mathfrak{S} \ \mathcal{G}] \Rightarrow^* \omega$, then there exists an item $x \in \text{Earley } \mathcal{G} \ \omega$ such that $\text{is_finished } \mathcal{G} \ \omega \ x$.

Two further important properties that we will need:

1. Well-formedness: For all $x \in \text{Earley } \mathcal{G} \ \omega$, $\text{wf_item } \mathcal{G} \ \omega \ x$ holds.
2. Finiteness: The set $\text{Earley } \mathcal{G} \ \omega$ is finite.

4.1 Proving Well-formedness and Finiteness

The proof of the well-formedness of the set of Earley items is straightforward by induction on the definition of *Earley*. We omit it.

Furthermore, there exist only a finite number of Earley items: Given that all Earley items are well-formed, it suffices to prove that there is only a finite number of well-formed Earley items. We define the set T as $\text{set } (\mathfrak{R} \ \mathcal{G}) \times \{0..m\} \times \{0..|\omega|\} \times \{0..|\omega|\}$, where m denotes the maximum length of all right-hand sides of production rules from the grammar \mathcal{G} . The set T is finite as there is only a finite number of production rules, and both the right-hand side of each production rule and the input ω are finite. Furthermore, T is an over-approximation of $\text{Earley } \mathcal{G} \ \omega$, as every well-formed Earley item is contained within T by definition of well-formedness.

4.2 Proving Soundness

An item $A \rightarrow \alpha \bullet \beta, i, j$ is considered sound, *sound_item*, if it satisfies $\mathcal{G} \vdash [A] \Rightarrow^* (\omega_{i/j} @ \beta)$, where $\omega_{i/j}$ is the subsequence of ω from index i to (but excluding) j . Let x denote an arbitrary item in $\text{Earley } \mathcal{G} \ \omega$. We prove $\text{sound_item } \mathcal{G} \ \omega \ x$ by induction on the definition of *Earley*.

For the *Init* rule, we have $x = \mathfrak{S} \ \mathcal{G} \rightarrow \bullet \alpha, 0, 0$. Furthermore, we know that $\omega_{0/0}$ equals the empty list. Our goal is to show that there exists a derivation from $\mathfrak{S} \ \mathcal{G}$ to α . This is immediately evident as $(\mathfrak{S} \ \mathcal{G}, \alpha)$ is a production rule from the grammar due to the well-formedness of the item.

For the *Scan* rule, we deal with an item $x = A \rightarrow \alpha \bullet a\beta, i, j$, and the induction hypothesis is that $\mathcal{G} \vdash [A] \Rightarrow^* \omega_{i/j} @ (a \# \beta)$. Since we have that $\omega ! j = a$, this is equivalent to $\mathcal{G} \vdash [A] \Rightarrow^* \omega_{i/j+1} @ \beta$, which, in turn, implies the soundness of the new item $x = A \rightarrow \alpha a \bullet \beta, i, j+1$.

For the *Prediction* rule, the new item is $x = B \rightarrow \bullet \alpha, j, j$. Since $\omega_{j/j}$ equals the empty list, our task is to show that $\mathcal{G} \vdash [B] \Rightarrow^* \alpha$. This follows directly as (B, α) is a production rule of the grammar.

For the *Complete* rule, we have two items: $x = A \rightarrow \alpha \bullet B\beta, i, j$ and $y = B \rightarrow \gamma \bullet, j, k$. The two induction hypotheses are $\mathcal{G} \vdash [A] \Rightarrow^* \omega_{i/j} @ (B \# \beta)$ and $\mathcal{G} \vdash [B] \Rightarrow^* \omega_{j/k}$. Combining these statements yields $\mathcal{G} \vdash [A] \Rightarrow^* \omega_{i/j} @ \omega_{j/k} @ \beta$, which is equivalent to $\mathcal{G} \vdash [A] \Rightarrow^* \omega_{i/k} @ \beta$, and thus, implies the soundness of the new item $A \rightarrow \alpha B \bullet \beta, i, k$, concluding the soundness proof.

theorem *soundness_Earley*:

assumes $\exists x \in \text{Earley } \mathcal{G} \ \omega. \ \text{is_finished } \mathcal{G} \ \omega \ x$

shows $\mathcal{G} \vdash [\mathfrak{S} \ \mathcal{G}] \Rightarrow^* \omega$

4.3 Proving Completeness

Completeness is the most intricate proof obligation, and we begin by providing some intuition about the fundamental proof idea. We call a set I of items *partially completed* if for every item $A \rightarrow \alpha \bullet s\beta, i, j$ in I and every derivation $\mathcal{G} \vdash [s] \Rightarrow^* \omega_{j/k}$, the set I also contains the item $A \rightarrow \alpha s \bullet \beta, i, k$.

Now, consider the item $A \rightarrow \bullet s_0 s_1 \dots s_n, i, i_0$. If this item is present in a partially completed set of items I , and there exists a derivation $\mathcal{G} \vdash [s_0] \Rightarrow^* \omega_{i_0/i_1}$, then the item $A \rightarrow s_0 \bullet s_1 \dots s_n, i, i_1$ is also included in the set. If there exists another derivation $\mathcal{G} \vdash [s_1] \Rightarrow^* \omega_{i_1/i_2}$, the statement holds again for the item $A \rightarrow s_0 s_1 \bullet \dots s_n, i, i_2$, and so on. This continues until, for a derivation $\mathcal{G} \vdash [s_n] \Rightarrow^* \omega_{i_n/j}$, the completed item $A \rightarrow s_0 s_1 \dots s_n \bullet, i, j$ is in I , provided that we have $i \leq i_0 \leq i_1 \leq \dots \leq i_n \leq j$.

The definitions of *partially completed* and the subsequent theorem that captures the outlined proof idea are more intricate in their details. They also encompass the necessary bounds for the indices, and make use of the analogous definition of derivations through the predicate *Derivation*, which contains an actual derivation D . They also incorporate an additional predicate on D . Its purpose is to limit the length of the derivation D . This is crucial because the proof of the partial completeness of the set of Earley items is by induction on the length of the derivation.

$$\begin{aligned} \text{partially_completed } l \mathcal{G} \omega I P &= (\forall r d i j k x s D. \\ &j \leq k \wedge k \leq l \wedge l \leq |\omega| \wedge \\ &x = \text{Item } r d i j \wedge x \in I \wedge \text{next_symbol } x = \text{Some } s \wedge \\ &\text{Derivation } \mathcal{G} [s] D (\omega_{j/k}) \wedge P D \longrightarrow \text{Item } r (d+1) i k \in I) \end{aligned}$$

theorem *partially_completed_upto*:

$$\begin{aligned} \text{assumes } &j \leq k \text{ and } k \leq |\omega| \\ \text{assumes } &x = \text{Item } (A, \alpha) d i j \text{ and } x \in I \text{ and } \forall x \in I. \text{wf_item } \mathcal{G} \omega x \\ \text{assumes } &\text{Derivation } \mathcal{G} (\beta\text{-item } x) D (\omega_{j/k}) \\ \text{assumes } &\text{partially_completed } k \mathcal{G} \omega I (\lambda D'. |D'| \leq |D|) \\ \text{shows } &\text{Item } (A, \alpha) |\alpha| i k \in I \end{aligned}$$

theorem *partially_completed_Earley*:

$$\text{shows } \text{partially_completed } |\omega| \mathcal{G} \omega (\text{Earley } \mathcal{G} \omega) (\lambda _ . \text{True})$$

To establish the completeness of the inductive definition for the set of Earley items, we apply both of the preceding theorems. By assumption, there exists a derivation of the input ω from the grammar's start symbol. We can decompose this derivation into a single initial production rule $(\mathfrak{S} \mathcal{G}, \alpha)$ and a subsequent derivation $\text{Derivation } \mathcal{G} \alpha D \omega$. Additionally, we know, by definition of the *Init* rule, that the item $\mathfrak{S} \mathcal{G} \rightarrow \bullet \alpha, 0, 0$ is in *Earley* $\mathcal{G} \omega$. Moreover, considering that each Earley item is well-formed and the set of Earley items is partially completed, as proved by the theorem *partially_completed_Earley*, we can consequently discharge the assumptions of the theorem *partially_completed_upto*. As a result, we know that the finished item $\mathfrak{S} \mathcal{G} \rightarrow \alpha \bullet, 0, |\omega|$ is indeed present in *Earley* $\mathcal{G} \omega$.

theorem *completeness_Earley*:

$$\begin{aligned} \text{assumes } &\mathcal{G} \vdash [\mathfrak{S} \mathcal{G}] \Rightarrow^* \omega \text{ and } \text{is_word } \mathcal{G} \omega \\ \text{shows } &\exists x \in \text{Earley } \mathcal{G} \omega. \text{is_finished } \mathcal{G} \omega x \end{aligned}$$

theorem *correctness_Earley*:

$$\begin{aligned} \text{assumes } &\text{is_word } \mathcal{G} \omega \\ \text{shows } &(\exists x \in \text{Earley } \mathcal{G} \omega. \text{is_finished } \mathcal{G} \omega x) \longleftrightarrow \mathcal{G} \vdash [\mathfrak{S} \mathcal{G}] \Rightarrow^* \omega \end{aligned}$$

5 An Executable Earley Recognizer

We refine the inductive Earley definition of the previous section to an executable algorithm, a *recognizer* that tells us if the input ω is in the language specified by the grammar \mathcal{G} . Our Earley recognizer is a functional algorithm modeled after Earley’s original imperative implementation. We start with an informal explanation. The algorithm processes the input $\omega = a_0, \dots, a_{n-1}$ while maintaining a list of $n+1$ *bins*. An initial bin B_0 and one bin B_{i+1} for each symbol a_i in the input. Each bin is a variable length list of Earley *entries*. Each entry is a pair consisting of an Earley item and “pointers”, i.e. indices, indicating the originating entry needed for the construction of parse trees. These pointers are elements of a data type with three alternatives:

A pointer can either be a *null* pointer, denoted by \perp , a predecessor pointer representing a single index i , or a nonempty list of reduction pointers containing triples of indices, $(a, b, c), (d, e, f), \dots$. We define the exact semantics in the following paragraphs. To improve readability we omit showing any constructors of the entry and pointer data types and only use the shorthand notation. For example, an entry comprising the item $A \rightarrow \alpha \bullet \beta, i, j$ and the reduction pointer (a, b, c) is written $A \rightarrow \alpha \bullet \beta, i, j; (a, b, c)$.

The algorithm generates the bins in ascending order, starting at bin B_0 . Each bin serves a dual purpose: as a worklist of entries to be processed, and as a set of items that are already present, ensuring that no two entries with identical items are present within the same bin. An entry with the item $A \rightarrow \alpha \bullet \beta, i, j$ is always in bin B_j , in other words, the end index of the item equals the index of the bin.

Initially, the algorithm populates bin B_0 with the items corresponding to the *Init* rule of the inductive Earley definition. Each initial item is accompanied by a null pointer. Table 1 illustrates the executable algorithm by example for the toy grammar $\mathcal{G} : S \rightarrow x \mid S + S$ and input: $\omega = x + x + x$, showcasing the complete bins after a run of the algorithm. In the example, the bins contain the two initial entries $S \rightarrow \bullet x, 0, 0; \perp$ and $S \rightarrow \bullet S + S, 0, 0; \perp$ in bin B_0 . The algorithm proceeds to process the worklist, from top to bottom, until the bin stabilizes. Then, it moves on to the next bin.

For each item x of the current entry at index l in the k -th bin, the algorithm applies operations corresponding to the three rules *Scan*, *Predict*, *Complete*.

- Case $x = A \rightarrow \alpha \bullet a\beta, j, k$: if the symbol at position k in ω is the terminal symbol a , the entry $A \rightarrow \alpha a \bullet \beta, j, k + 1; l$ is inserted into the next bin B_{k+1} . The index l indicates the predecessor index, signifying that the originating entry of this new entry resides in the previous bin at index l . Table 1 contains the entry $S \rightarrow x \bullet, 0, 1; 0$ at index 0 in bin B_1 , and its predecessor is the entry $S \rightarrow \bullet x, 0, 0; \perp$ at index 0 in bin B_0 .
- Case $x = A \rightarrow \alpha \bullet B\beta, j, k$: for each production rule (B, γ) of the grammar \mathcal{G} , an entry $B \rightarrow \bullet \gamma, k, k; \perp$ is inserted into the current bin B_k . A null pointer is added to the entry, as no origin information is required for constructing parse trees. Table 1 contains the entries $S \rightarrow \bullet x, 2, 2; \perp$ and $S \rightarrow \bullet S + S, 2, 2; \perp$ in bin B_2 , both predicted by the entry $S \rightarrow S + \bullet S, 0, 2; 1$ in the same bin.
- Case $x = B \rightarrow \gamma \bullet, j, k$: if an item is complete, the algorithm searches the origin bin B_j for any entries with items of the form $A \rightarrow \alpha \bullet B\beta, i, j$. If it finds such an entry at index l' , it inserts one new entry $A \rightarrow \alpha B \bullet \beta, i, k; (j, l', l)$ into the current bin. The origin information (j, l', l) is a reduction pointer. The first two indices, j and l' , indicate that the predecessor entry resides in bin B_j at index l' . The last index, l , describes the position of the reduction entry at index l in the current bin B_k . An entry may contain more than one reduction pointer in cases where the grammar is ambiguous and there are

multiple ways to derive the input corresponding to the item. Table 1 contains the entry $S \rightarrow S + S\bullet, 0, 5; (4, 1, 0), (2, 0, 1)$, capturing the two possible derivations of ω : $(x + x) + x$ and $x + (x + x)$. The entry, with a single reduction pointer $(4, 1, 0)$, was initially created due to the reduction entry $S \rightarrow x\bullet, 4, 5; 2$ at index 0 in bin B_5 and the predecessor entry $S \rightarrow S + \bullet S, 0, 4; 3$ at index 1 in bin B_4 . However, the second reduction pointer $(2, 0, 1)$ was later added due to the reduction entry $S \rightarrow S + S\bullet, 2, 5; (4, 0, 0)$ at index 1 in bin B_5 and the predecessor entry $S \rightarrow S + \bullet S, 0, 2; 1$ at index 0 in bin B_2 .

The algorithm inserts an entry into a bin as follows: Iterate through the bin, and, for each entry, check if its item matches the item of the entry to be inserted. If a match is found, and the pointer of the entry to be inserted is a reduction pointer, merge the items by adding the reduction pointer to the already present entry. Otherwise, if there is no match or the pointer is not a reduction pointer, do not make any additions. In both cases, terminate the insertion process. If there are no entries with matching items, append the entry to the end of the bin.

■ **Table 1** Earley bins for $\mathcal{G}: S \rightarrow x \mid S + S$, and $\omega = x + x + x$.

	B_0	B_1	B_2
0	$S \rightarrow \bullet x, 0, 0; \perp$	$S \rightarrow x\bullet, 0, 1; 0$	$S \rightarrow S + \bullet S, 0, 2; 1$
1	$S \rightarrow \bullet S + S, 0, 0; \perp$	$S \rightarrow S \bullet + S, 0, 1; (0, 1, 0)$	$S \rightarrow \bullet x, 2, 2; \perp$
2			$S \rightarrow \bullet S + S, 2, 2; \perp$
	B_3	B_4	B_5
0	$S \rightarrow x\bullet, 2, 3; 1$	$S \rightarrow S + \bullet S, 2, 4; 2$	$S \rightarrow x\bullet, 4, 5; 2$
1	$S \rightarrow S + S\bullet, 0, 3; (2, 0, 0)$	$S \rightarrow S + \bullet S, 0, 4; 3$	$S \rightarrow S + S\bullet, 2, 5; (4, 0, 0)$
2	$S \rightarrow S \bullet + S, 2, 3; (2, 2, 0)$	$S \rightarrow \bullet x, 4, 4; \perp$	$S \rightarrow S + S\bullet, 0, 5; (4, 1, 0), (2, 0, 1)$
3	$S \rightarrow S \bullet + S, 0, 3; (0, 1, 1)$	$S \rightarrow \bullet S + S, 4, 4; \perp$	$S \rightarrow S \bullet + S, 4, 5; (4, 3, 0)$
4			$S \rightarrow S \bullet + S, 2, 5; (2, 2, 1)$
5			$S \rightarrow S \bullet + S, 0, 5; (0, 1, 2)$

5.1 Recognizer Implementation

We now examine the formal definition of the recognizer. There are four functions $Init_L$, $Scan_L$, $Predict_L$, and $Complete_L$ implementing *list*-based versions of the four corresponding rules of the inductive Earley definition. Due to space restrictions we only show the function $Init_L$, constructing the initial bins, and the function $Predict_L$ that returns a list of new entries to be inserted into the bins. Functions $Scan_L$ and $Complete_L$ have the same return type.

```

Init_L  $\mathcal{G} \omega = ($ 
  let  $rs = filter (\lambda r. lhs\_rule r = \mathfrak{S} \mathcal{G}) (remdups (\mathfrak{R} \mathcal{G}))$  in
  let  $b0 = map (\lambda r. (Item r 0 0 0, Null)) rs$  in
  let  $bs = replicate (|\omega| + 1) ([])$  in  $bs[0 := b0]$ 

```

```

Predict_L  $k \mathcal{G} A = ($ 
  let  $rs = filter (\lambda r. lhs\_rule r = A) (\mathfrak{R} \mathcal{G})$  in
  map  $(\lambda r. (Item r 0 k k, Null)) rs$ 

```

The central piece of the implementation is the function $Earley_L-bin'$. The function computes the entries of the k -th bin starting at the entry at index i . It examines the symbol following the dot of the item of the entry and, depending on the type of the symbol or

whether such a symbol exists at all, applies one of the three executable operations, obtaining a list of potentially new entries. These entries are subsequently inserted into the bins using the function *upd_bins* (definition omitted). Function *Earley_L-bin* starts this process at the beginning of the bin at index 0.

```

EarleyL-bin' k G ω bs i = (
  if i ≥ |(items (bs!k))| then bs
  else
    let x = items (bs!k) ! i in
    let bs' =
      case next_symbol x of
        Some s ⇒ (
          if s ∉ nonterminals G then
            if k < |ω| then upd_bins bs (k+1) (ScanL k ω s x i) else bs
          else upd_bins bs k (PredictL k G s)
        | None ⇒ upd_bins bs k (CompleteL k x bs i)
    in EarleyL-bin' k G ω bs' (i+1))
EarleyL-bin k G ω bs = EarleyL-bin' k G ω bs 0

```

The function *Earley_L-bin'* is defined as a partial function as it might not terminate if it keeps inserting newly generated entries forever into the bin it currently operates on. However, we know that the newly generated entries do not contain arbitrary but only well-formed bin items. In other words, each bin B_k contains only entries with items that are well-formed and additionally have the end index k . We have already proved that the number of well-formed Earley items is finite, and the implementation ensures that a new entry is added to the bin only if its item is not already present in one of the bin's entries. Therefore, the function will eventually run out of new entries to insert into the bin it currently operates on and terminate.

Although HOL is a logic of total functions, Isabelle supports the definition of potentially non-terminating functions provided they are tail-recursive (like *Earley_L-bin'*) or their result is an optional value (like function *build_tree'* below). The underlying domain-theoretic definitional constructions are due to Krauss [24]. However, we cannot prove anything about such a function because Isabelle does not know for which inputs it terminates, or if it terminates at all. As a result, Isabelle does not generate an appropriate induction schema for it. Such a schema must be proved by hand by specifying a suitable type and measure for which the function terminates. For the function *Earley_L-bin'* we define the measure *earley_measure* (k, G, ω, bs) $i = |\{x \mid wf_item\ G\ \omega\ x \wedge end_item\ x = k\}| - i$ and prove that it is strictly decreasing for every tail-recursive function call.

The function *Earley_L-bins* computes the bins upto a specific index starting at bin zero. And finally, function *Earley_L* computes the complete bins.

```

EarleyL-bins 0 G ω = EarleyL-bin 0 G ω (InitL G ω)
EarleyL-bins (Suc n) G ω = EarleyL-bin (Suc n) G ω (EarleyL-bins n G ω)
EarleyL G ω = EarleyL-bins |ω| G ω

```

5.2 Recognizer Correctness Proof

We follow Jones' [20] refinement approach, proving that the set of items formed by the implementation's bins is exactly the inductive set of Earley items, thereby establishing soundness and completeness. The main complications arise since the deterministic implementation necessarily generates the set of Earley items in a particular order. It starts with the initial items in bin zero and constructs the subsequent bins in a horizontal ascending order. But each

bin is computed top to bottom, introducing a second vertical order. Our refinement approach reflects these two orders. We first refine the inductive definition to an intermediate fixpoint algorithm, and then refine this algorithm further to the actual list-based implementation.

Let $\text{bin } I k$ denote the subset of the set of items I that end with index k . Furthermore, let $\text{base } \omega I k$ denote the subset of I that forms the k -th *base* of a bin, meaning the subset of I containing only items of the form $A \rightarrow \alpha a \bullet \beta, i, j$, where a is a terminal symbol preceding the dot. If k is zero, $\text{base } \omega I 0$ consists of all initial items $\mathfrak{S} \mathcal{G} \rightarrow \bullet \alpha, 0, 0$.

For the intermediate fixpoint algorithm we define the set of initial items Init_F and three functions Predict_F , Scan_F , and Complete_F mirroring the rules of the inductive definition. Using $\text{Earley}_F\text{-bin-step } k \mathcal{G} \omega I = I \cup \text{Scan}_F k \omega I \cup \text{Complete}_F k I \cup \text{Predict}_F k \mathcal{G} I$ we define the computation of a single bin as a fixpoint computation. The remaining functions $\text{Earley}_F\text{-bins}$ and Earley_F are defined analogously to the list-based implementation. The following lemma states the completeness argument for the first refinement step.

lemma *Earley-bin-base-sub-Earley-bin*:

assumes $\text{Init}_F \mathcal{G} \subseteq I$ **and** $\forall k' < k. \text{bin } (\text{Earley } \mathcal{G} \omega) k' \subseteq I$
assumes $\text{base } \omega (\text{Earley } \mathcal{G} \omega) k \subseteq I$ **and** *is-word* $\mathcal{G} \omega$
shows $\text{bin } (\text{Earley } \mathcal{G} \omega) k \subseteq \text{bin } (\text{Earley}_F\text{-bin } k \mathcal{G} \omega I) k \wedge$
 $\text{base } \omega (\text{Earley } \mathcal{G} \omega) (k+1) \subseteq \text{bin } (\text{Earley}_F\text{-bin } k \mathcal{G} \omega I) (k+1)$

The fixpoint computation of the k -th bin yields a superset of the k -th bin and base $k+1$ of the inductive definition. We omit the proof, and the analogous but much simpler soundness lemma. As both the inductive and fixpoint definition commence with the same items, $(\text{base } \omega (\text{Earley } \mathcal{G} \omega) 0 = \text{Init}_F \mathcal{G})$, we apply this argument $|\omega|$ *times* (i.e. by induction), yielding the correctness proposition $\text{Earley } \mathcal{G} \omega = \text{Earley}_F \mathcal{G} \omega$.

Refining the algorithm further to the list-based implementation uncovers a well-known problem concerning the computation of a single bin. Consider an item $A \rightarrow \bullet, j, k$ for an epsilon rule $(A, [])$. Since the item is by definition complete the algorithm applies the Complete_L operation. It identifies the origin bin j of the item. Due to the epsilon rule this is the k -th bin, meaning the bin that the algorithm is currently computing. It then searches this bin for any items $B \rightarrow \alpha A \bullet \beta, i, j$. However, the bin might not be fully constructed at this point, and some of these items could be missing. Consequently, the algorithm may not generate all items $B \rightarrow \alpha A \bullet \beta, i, j$, when applying the completion operation for the item $A \rightarrow \bullet, j, k$. Moreover, there could be transitively dependent items that the algorithm fails to compute. Various solutions have been proposed:

- Earley [12] suggests that the implementation keeps track of items with epsilon rules and considers this information in the subsequent execution of the algorithm.
- Grune and Jacobs [18] and Aho and Ullman [4] propose to interleave the prediction and completion operations until the algorithm stabilizes.
- Kegler [22] addresses the problem by internally rewriting the grammar into epsilon-free form.
- Aycock and Horspool [6] precompute nullable non-terminals and modify the prediction operation.
- Polat et al. [32] roughly follow the work of Aycock and Horspool.

We follow Jones [20], define $\varepsilon\text{-free } \mathcal{G} = (\forall r \in \text{set } (\mathfrak{R} \mathcal{G}). \text{rhs-rule } r \neq [])$, and consequently restrict the grammar to be epsilon free. If we disallow any production rules of the form $(A, [])$, then the function $\text{Earley}_L\text{-bin}$ is idempotent and in particular the result of the completion operation is invariant of state of the current bin.

On paper this argument is straightforward, but the formalization is surprisingly tricky in the details. The function $\text{Earley}_L\text{-bin } k \mathcal{G} \omega bs = \text{Earley}_L\text{-bin}' k \mathcal{G} \omega bs 0$ is defined in terms of $\text{Earley}_L\text{-bin}'$ which may start its computation at an arbitrary index i instead of 0 .

31:10 A Verified Earley Parser

We need the following two generalized lemmas for the completeness proof.

lemma *Earley_F-bin-step-sub-Earley_L-bin'*:

assumes $(k, \mathcal{G}, \omega, bs) \in wf_earley_input$ **and** $is_word \mathcal{G} \omega$

assumes $\forall x \in bins \ bs. \ sound_item \mathcal{G} \omega x$ **and** $\epsilon_free \mathcal{G}$

assumes $Earley_F_bin_step \ k \ \mathcal{G} \ \omega \ (bins_upto \ bs \ k \ i) \subseteq bins \ bs$

shows $Earley_F_bin_step \ k \ \mathcal{G} \ \omega \ (bins \ bs) \subseteq bins \ (Earley_L_bin' \ k \ \mathcal{G} \ \omega \ bs \ i)$

If applying a single step of the fixpoint computation, *Earley_F-bin-step*, to the bins including the items of the first k bins but only up (but not including) the i -th item of the k -th bin doesn't change the content of the bins, or, in other words, those items are already correctly processed, then the list-based implementation computes at least the same items as applying one step of the fixpoint computation.

lemma *Earley_L-bin'-idem*:

assumes $(k, \mathcal{G}, \omega, bs) \in wf_earley_input$

assumes $i \leq j \ \forall x \in bins \ bs. \ sound_item \mathcal{G} \omega x$ **and** $\epsilon_free \mathcal{G}$

shows $bins \ (Earley_L_bin' \ k \ \mathcal{G} \ \omega \ (Earley_L_bin' \ k \ \mathcal{G} \ \omega \ bs \ i) \ j) = bins \ (Earley_L_bin' \ k \ \mathcal{G} \ \omega \ bs \ i)$

Using those two lemmas we can prove completeness of the list-based algorithm for a single bin. Since the list-based algorithm follows the same horizontal order as the fixpoint algorithm the completeness proof for all bins is then straightforward. The soundness proof is again similar in structure, but once more much simpler.

We then define *recognizer* $\mathcal{G} \ \omega = (\exists x \in set \ (items \ (Earley_L \ \mathcal{G} \ \omega \ ! \ |\omega|)). \ is_finished \ \mathcal{G} \ \omega \ x)$, prove the equivalence of *Earley* and *Earley_L* and obtain a corollary stating the correctness of the recognizer under the assumption of an epsilon-free grammar.

theorem *Earley-eq-Earley_L*:

assumes $is_word \mathcal{G} \ \omega$ **and** $\epsilon_free \mathcal{G}$

shows $Earley \ \mathcal{G} \ \omega = bins \ (Earley_L \ \mathcal{G} \ \omega)$

corollary *correctness-recognizer*:

assumes $is_word \mathcal{G} \ \omega$ **and** $\epsilon_free \mathcal{G}$

shows $recognizer \ \mathcal{G} \ \omega \longleftrightarrow \mathcal{G} \vdash [\mathcal{G}] \Rightarrow^* \omega$

6 An Earley Parser

We now upgrade our recognizer to a parser. Extending an Earley recognizer to a parser is no simple task. Tomita [36] even pointed out a bug in Earley's original implementation that may lead to erroneous derivations.

A major complication is that Earley's parser allows for ambiguous grammars, which may lead to exponentially many or even infinitely many parse trees. For the ambiguous grammar $S \rightarrow SS \mid a$, the number of possible parse trees corresponds to the Catalan number $C_n = \frac{1}{n+1} \binom{2n}{n}$ for an input of length $n - 1$. For the cyclic grammar $A \rightarrow B \mid a, B \rightarrow A$ the input a has infinitely many parse trees because of the by cycle of non-terminals A and B .

An Earley recognizer can be made to run in at most quadratic space and cubic time. Any extension of the recognizer to a parser, especially for ambiguous or cyclic grammars, must choose a suitable data representation and be implemented carefully, in order not to degrade those time and space bounds too much.

Probably the most well-known data representation is the *shared packed parse forest* (SPPF), as described and used by Tomita [37]. However, Johnson [19] showed that these forests are of unbounded polynomial size in the worst case. On the other hand, Scott [35] introduced a slightly different version of SPPFs, and proved that her Earley parser implementation runs in cubic time and space.

If the pointer of e is a null pointer, the algorithm begins building a new branch. Specifically, it constructs a branch *Branch* $A []$, where A is the left-hand side symbol of the production rule of the item x . If the algorithm encounters a predecessor pointer *Pre* pre , it recursively calls itself for the previous bin, $k - 1$, at index pre . This recursive call results in a partially completed parse branch. Following the semantics of the predecessor pointer, the algorithm appends a new Leaf containing the terminal symbol at index $k - 1$ of the input ω to the list of subtrees of this branch. In the case of a reduction pointer, the algorithm considers only the first triple (k', pre, red) . It calls itself recursively for the predecessor entry in bin k' at index pre and the completed entry in the same bin at index red . These recursive calls yield respectively a partially completed parse branch *Branch* $A ts$ and a complete parse tree t . Following the semantics of the reduction pointer, the complete branch t is appended to the list of subtrees ts .

The final algorithm *build_tree* (definition omitted) first computes the complete bins using the function *Earley_L*. It then searches the last bin for any finished item x , and calls the function *build_tree'* at the index of x in the final bin, returning the resulting parse tree as an optional value, if such a tree exists, or *None* in case of non-termination.

6.2 Proving Termination

The function *build_tree'* is a partial function. It calls itself recursively, following the information provided by the pointers. Intuitively, it terminates because predecessor pointers lead to earlier bins, and reduction pointers point upwards within a bin. Consequently, we define a measure *build_tree'_measure* $(bs, \omega, k, i) = \text{foldl } (+) \ 0 \ (\text{map length } (\text{take } k \ bs)) + i$, counting the number of entries in the first k bins up to the i -th entry in bin $k+1$. But in the case of malformed input, the pointers might result in a cycle of recursive calls and thus the measure is not strictly decreasing. And even for well-formed input, complications arise.

Consider an entry at index i in the k -th bin. If the entry contains a reduction triple (k', pre, red) , the algorithm calls itself recursively for the reduction entry at index red in bin k . Now consider the cyclic grammar $A \rightarrow B \mid a, B \rightarrow A$ and the input $\omega = a$. In this case, the last bin contains a cycle of reductions: there is an entry $B \rightarrow A\bullet, 0, 1; (0, 2, 0), (0, 2, 2)$ at index 1, and its *second* reduction triple $(0, 2, 2)$ leads to index 2 of the same bin. There, we find the entry $A \rightarrow B\bullet, 0, 1; (0, 0, 1)$ with a reduction triple to index 1, completing the cycle, and leading to potential non-termination of the algorithm.

We constrain the input of the function *build_tree'* to *wf_tree_input* $= \{(bs, \omega, k, i) \mid \text{sound_ptrs } \omega \ bs \wedge \text{mono_red_ptr } bs \wedge k < |bs| \wedge k \leq |\omega| \wedge i < |bs| \ k\}$ where the definition of *sound_ptrs* and *mono_red_ptr* is the following:

$$\begin{aligned} \text{sound_ptrs } \omega \ bs &= (\forall k < |bs|. \forall e \in \text{set } (bs!k). \\ &(\text{snd } e = \text{Null} \longrightarrow \text{predicts } (\text{fst } e)) \wedge \\ &(\forall pre. \text{snd } e = \text{Pre } pre \longrightarrow \\ &k > 0 \wedge pre < |bs!(k-1)| \wedge \text{scans } \omega \ k \ (\text{fst } (bs!(k-1)!pre)) \ (\text{fst } e)) \wedge \\ &(\forall p \ ps \ k' \ pre \ red. \text{snd } e = \text{PreRed } p \ ps \wedge (k', pre, red) \in \text{set } (p\#\ps) \longrightarrow \\ &k' < k \wedge pre < |bs!k'| \wedge red < |bs!k| \wedge \text{completes } k \ (\text{fst } (bs!k'!pre)) \ (\text{fst } e) \ (\text{fst } (bs!k!red)))) \\ \text{mono_red_ptr } bs &= (\forall k < |bs|. \forall i < |bs!k|. \\ &\forall k' \ pre \ red \ ps. \text{snd } (bs!k!i) = \text{PreRed } (k', pre, red) \ ps \longrightarrow red < i) \end{aligned}$$

The predicate *sound_ptrs* defines the semantics for the pointer datatype, ensuring that the pointers do not exceed the bounds of the bins and that related items follow the semantics of the respective operation. The function *build_tree'* always follows the first reduction triple $(k', pre, red) \in \text{set } (p \# \ps)$ for a reduction pointer *PreRed* $p \ ps$, and the predicate *mono_red_ptr*

guarantees that the reduction pointer *red* of this reduction triple always points strictly upwards within the bin, even for cyclic grammars as in the example above, enabling us to prove termination of the algorithm.

lemma *build_tree'_termination*:

assumes $(bs, \omega, k, i) \in wf_tree_input$

shows $\exists A \text{ ts. } build_tree' \text{ } bs \ \omega \ k \ i = \text{Some } (Branch \ A \ ts)$

6.3 Proving Correctness

To prove the correctness of the parse tree algorithm, we first show that the resulting tree corresponds in derivation and yield to the Earley item where the construction originated.

theorem *wf_item_yield_build_tree'*:

assumes $(bs, \omega, k, i) \in wf_tree_input$ **and** $wf_bins \ \mathcal{G} \ \omega \ bs$

assumes $build_tree' \text{ } bs \ \omega \ k \ i = \text{Some } t$ **and** $x = fst \ (bs!k!i)$

shows $wf_item_tree \ \mathcal{G} \ x \ t \wedge wf_yield \ \omega \ x \ t$

The predicate *wf_bins* states that each bin only contains entries with distinct items, all items are well-formed and their end index equals the index of the bin they reside in. The proof is by induction on *build_tree'_measure* (bs, ω, k, i) .

As a corollary, we obtain the first correctness statement for epsilon-free grammars: any constructed parse tree adheres to the rules of the grammar, is rooted at its start symbol and yields the complete input:

corollary *wf_rule_root_yield_build_tree_EarleyL*:

assumes $\varepsilon_free \ \mathcal{G}$ **and** $build_tree \ \mathcal{G} \ \omega \ (Earley_L \ \mathcal{G} \ \omega) = \text{Some } t$

shows $wf_rule_tree \ \mathcal{G} \ t \wedge root \ t = \mathfrak{S} \ \mathcal{G} \wedge yield \ t = \omega$

The *build_tree* function scans the last bin for any finished item x , and calls the function *build_tree'*. Given that the function *Earley_L* guarantees well-formed tree inputs, the resulting tree conforms both in derivation and yield to the item x , as proved in the preceding theorem. Since x is a finished item, the root of the tree corresponds to the start symbol of the grammar, the tree's yield encompasses the complete input, and the definition of *wf_item_tree* aligns with the definition of *wf_rule_tree*.

The second and final correctness theorem follows: the algorithm returns a parse tree, if and only if, there exists a derivation of ω from the grammar's start symbol.

theorem *correctness_build_tree_EarleyL*:

assumes $is_word \ \mathcal{G} \ \omega$ **and** $\varepsilon_free \ \mathcal{G}$

shows $(\exists t. build_tree \ \mathcal{G} \ \omega \ (Earley_L \ \mathcal{G} \ \omega) = \text{Some } t) \longleftrightarrow \mathcal{G} \vdash [\mathfrak{S} \ \mathcal{G}] \Rightarrow^* \omega$

The function *build_tree* finds a finished item in the last bin and returns a parse tree, if and only if, the bins generated by the function *Earley_L* contain a finished item. These items align precisely with those items specified by the inductive definition. Lastly, the inductive set of Earley items contains a finished item, if and only if, there exists a derivation of the input from the grammar's start symbol.

7 Evaluation

We present an informal argument for the algorithm's running time, and empirically access its efficiency. Given that we construct solely a single parse tree, the running time is dominated by the function *Earley_L*.

Let n denote the length of the input ω . Each bin B_j ($0 \leq j \leq n$) exclusively contains well-formed items $Item\ r\ d\ i\ j$. The number of possible production rules r , and possible dot positions d , are both constants independent of n , although they may be rather large. The index i is bounded by $0 \leq i \leq j$, and thus depends on j , which is bounded by n . The end index j always equals the index of the bin. Thus, the number of items in each bin B_j is $\mathcal{O}(n)$. The initial bin construction takes linear time. Scanning is also a linear operation, producing at most one new entry. Prediction runs in time proportional to the grammar, or constant time, adding a constant amount of new entries. Lastly, completion takes linear time as it scans the entire origin bin for applicable items. However, as the size of the origin bin is linear, this operation adds $\mathcal{O}(n)$ new entries to the current bin. The algorithm implements a bin as a list, so inserting e new entries into any bin takes time $e \times \mathcal{O}(n)$. The time for one execution of the body of the function $Earley_L_bin'$ is dominated by the time it takes to update the bins with e new items. In the worst case, this is completion, resulting in a running time of $e \times \mathcal{O}(n)$, for $e \in \mathcal{O}(n)$. Moreover, the function calls itself at most n times due to the size of the bin it operates on, resulting in cubic time complexity. Finally, the algorithm iterates over all n bins, leading to a final upper bound on the running time of $\mathcal{O}(n^4)$.

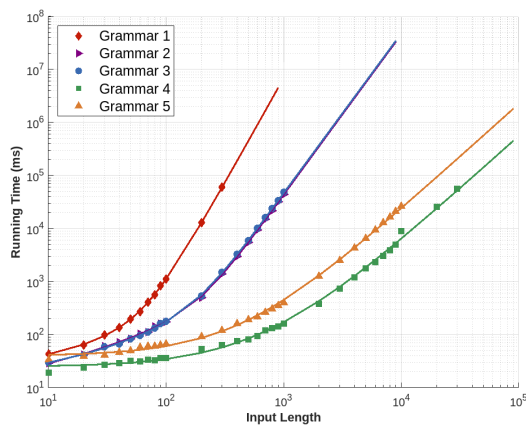
The space complexity for an Earley recognizer is quadratic: n bins of linear size. However, as each entry may have n reduction pointers in the worst case, the space required to represent all Earley entries with pointers becomes cubic in n .

It is worth noting that the running time is not optimal. Earley's [12] original implementation achieves an optimal running time of $\mathcal{O}(n^3)$ by implementing a bin as an imperative singly-linked list and additionally maintaining a cache of already inserted items. This cache reduces the insertion time of a new entry into a bin from linear to constant time such that computing a single bin takes in the worst case quadratic time. Further refinement of our functional implementation to an imperative algorithm with cache is future work.

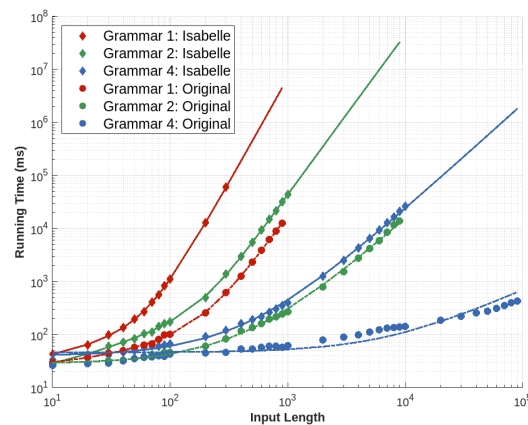
Additionally, we evaluate the running time of the exported recognizer Isabelle code in comparison to a hand-written imperative implementation of Earley's recognizer. Our target language is Scala (verified as well as handwritten) and the verified code can be integrated easily into existing code bases. Alternatively, Isabelle also supports Haskell, ML, and OCaml. We then conducted tests on five different grammars, averaging the execution of ten runs for each data point. The grammars and running times for Earley's [12] original implementation are:

1. An ambiguous grammar $S \rightarrow SS \mid a$, cubic running time.
2. A right-recursive grammar $S \rightarrow aS \mid a$, quadratic running time.
3. A palindrome grammar $S \rightarrow aSa \mid a$, quadratic running time.
4. A left recursive grammar $S \rightarrow Sa \mid a$, linear running time.
5. A grammar with bounded-ambiguity $S \rightarrow SX \mid a, X \rightarrow Y \mid Z, Y \rightarrow a, Z \rightarrow a$, linear running time.

Figure 1 depicts the running times in milliseconds for the exported Isabelle code on all five grammars. Figure 2 compares the verified code against the handwritten recognizer implementation for grammars one, two, and four. In both cases, the input is $\omega = a^n$. For sufficiently large inputs the hand-written implementation exhibits optimal running time, while the verified code exhibits a linear slowdown in the size of the input. This is also confirmed by a regression analysis. It is possible to fit polynomial models of the respective order, capturing the expected running time, to the data set, depicted as solid lines in the figures.



■ **Figure 1** Isabelle: all grammars.



■ **Figure 2** Isabelle vs Handwritten.

8 Related Work

We highlight related work on formalization of parsing algorithms, starting with LL-based parsing: Lasser et al. [25] verify an LL(1) parser generator using the Coq proof assistant. Edelmann et al. [13] formalize a derivative-based LL(1) parsing algorithm, proving correctness using the Coq proof assistant.

There also exist verified LR-based algorithms: Barthwal et al. [7] formalize background theory about context-free languages and grammars, and subsequently verify an SLR automaton and parser produced by a parser generator with the HOL4 proof assistant. Jourdan et al. [21] present a validator which checks if a context-free grammar and an LR(1) parser agree, producing correctness guarantees required by verified compilers.

Furthermore, there is relevant work on the verification of PEGs [17, 16], an alternative representation to CFGs. Blaudeau et al. [8] formalize the meta theory of PEGs. They build a verified parser interpreter based on higher-order parsing combinators for expression grammars using the PVS specification language and verification system. Koprowski et al. [23] present TRX: a PEG interpreter formally developed in Coq which also parses expression grammars.

Lastly, there exist a variety of verified parsers for general context-free grammars. Ridge [34] constructs a generic parser generator based-on combinator parsing. His approach has a worst case complexity of $\mathcal{O}(n^5)$ and is verified using the HOL4 theorem prover. Obua formalizes Local Lexing [31, 30] in Isabelle, a parsing concept that interleaves lexing and parsing allowing the lexing phase to be dependent on the parsing process. Firsov and Ustalu [14, 15] rewrite a context-free grammar into an equivalent one in Chomsky normal form and implement the CYK parsing algorithm. They verify their work in Agda. The CYK algorithm had already been verified by Bortin [9]. Danielsson [11] develops and verifies a monadic parser combinator library in Agda.

9 Conclusions

We formalized and verified a functional implementation of an Earley recognizer and parser based on Earley's [12] original imperative implementation, the refinement-based paper proof of Jones [20], and the work of Scott [35]. Initially, we defined an Earley recognizer inductively and proved soundness and completeness. We refined the inductive definition to a functional recognizer implementation (proving equivalence between the two levels). We also enhanced

the implementation with “pointers”, following the work of Scott [35]. Following Aho and Ullman [4], we implemented a functional algorithm that constructs a single parse tree, and proved its termination and correctness. Finally, we argued informally about the running time of our functional implementation, comparing it to an asymptotically optimal, hand-written, imperative implementation and providing empirical evidence supporting our claims.

Future work is mainly centered around improving the algorithm’s efficiency. A first step is a refinement to an imperative implementation that incorporates a cache to achieve optimal cubic time and space bounds. Further performance optimizations include improving the representation of the grammar for faster prediction [32] and grouping the items of a bin based on their next symbol [12]. This avoids searching the complete origin bin during the completion operation. Leo [26] describes an extension applicable to an Earley recognizer and parser that improves the complexity for grammars containing right recursion from quadratic to linear time and space. Earley [12] suggested using lookahead for the completion operation to improve the performance of his algorithm. However, Bouckaert et al. [10] argued that lookahead is better suited for the prediction operation. McLean and Horspool [27] claimed that lookahead actually slowed down an Earley parser, and Aycock and Horspool [5] concluded that the necessity of lookahead is at least controversial. Lastly, we would like to incorporate the work of Aycock and Horspool [6] and Polat et al. [32] to lift the minor restriction to epsilon-free grammars.

References

- 1 Failure to patch two-month-old bug led to massive equifax breach. <https://arstechnica.com/information-technology/2017/09/massive-equifax-breach-caused-by-failure-to-patch-two-month-old-bug>. Accessed: 2024-03-16.
- 2 Stagefright: Scary code in the heart of android. <https://www.blackhat.com/docs/us-15/materials/us-15-Drake-Stagefright-Scary-Code-In-The-Heart-Of-Android.pdf>. Accessed: 2024-03-16.
- 3 Windows media parsing remote code execution vulnerability. <https://nvd.nist.gov/vuln/detail/CVE-2016-0101>. Accessed: 2024-03-16.
- 4 Alfred V. Aho and Jeffrey D. Ullman. *The Theory of Parsing, Translation, and Compiling*. Prentice-Hall, Inc., USA, 1972.
- 5 John Aycock and R. Nigel Horspool. Directly-executable Earley parsing. In Reinhard Wilhelm, editor, *Compiler Construction, CC 2001*, volume 2027 of *LNCS*, pages 229–243. Springer, 2001. doi:10.1007/3-540-45306-7_16.
- 6 John Aycock and R. Nigel Horspool. Practical Earley parsing. *Comput. J.*, 45(6):620–630, 2002. doi:10.1093/comjnl/45.6.620.
- 7 Aditi Barthwal and Michael Norrish. Verified, executable parsing. In Giuseppe Castagna, editor, *Programming Languages and Systems*, pages 160–174, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.
- 8 Clement Blaudeau and Natarajan Shankar. A verified packrat parser interpreter for parsing expression grammars. In *Certified Programs and Proofs, CPP 2020*, pages 3–17. ACM, 2020. doi:10.1145/3372885.3373836.
- 9 Maksym Bortin. A formalisation of the Cocke-Younger-Kasami algorithm. *Archive of Formal Proofs*, April 2016. , Formal proof development. URL: <https://isa-afp.org/entries/CYK.html>.
- 10 M. Bouckaert, A. Pirotte, and M. Snelling. Efficient parsing algorithms for general context-free parsers. *Information Sciences*, 8(1):1–26, 1975. doi:10.1016/0020-0255(75)90002-X.

- 11 Nils Anders Danielsson. Total parser combinators. In Paul Hudak and Stephanie Weirich, editors, *International Conference on Functional Programming, ICFP 2010*, pages 285–296. ACM, 2010. doi:10.1145/1863543.1863585.
- 12 Jay Earley. An efficient context-free parsing algorithm. *Commun. ACM*, 13(2):94–102, 1970. doi:10.1145/362007.362035.
- 13 Romain Edelmann, Jad Hamza, and Viktor Kunčák. Zippy LL(1) parsing with derivatives. In *Programming Language Design and Implementation, PLDI 2020*, pages 1036–1051. ACM, 2020. doi:10.1145/3385412.3385992.
- 14 Denis Firsov and Tarmo Uustalu. Certified CYK parsing of context-free languages. *Journal of Logical and Algebraic Methods in Programming*, 83(5):459–468, 2014. Nordic Workshop on Programming Theory (NWPT 2012). doi:10.1016/j.jlamp.2014.09.002.
- 15 Denis Firsov and Tarmo Uustalu. Certified normalization of context-free grammars. In *Certified Programs and Proofs, CPP '15*, pages 167–174. ACM, 2015. doi:10.1145/2676724.2693177.
- 16 Bryan Ford. Packrat parsing: Simple, powerful, lazy, linear time, functional pearl. In *International Conference on Functional Programming, ICFP '02*, pages 36–47. ACM, 2002. doi:10.1145/581478.581483.
- 17 Bryan Ford. Parsing expression grammars: A recognition-based syntactic foundation. In *Principles of Programming Languages, POPL '04*, pages 111–122. ACM, 2004. doi:10.1145/964001.964011.
- 18 Dick Grune and Ceriel J. H. Jacobs. *Parsing Techniques: a Practical Guide*. Ellis Horwood, 1990.
- 19 Mark Johnson. *The Computational Complexity of GLR Parsing*, pages 35–42. Springer US, Boston, MA, 1991. doi:10.1007/978-1-4615-4034-2_3.
- 20 C B Jones. Formal development of correct algorithms: An example based on Earley’s recogniser. In *Proceedings of ACM Conference on Proving Assertions about Programs*, pages 150–169. ACM, 1972. doi:10.1145/800235.807083.
- 21 Jacques-Henri Jourdan, François Pottier, and Xavier Leroy. Validating LR(1) parsers. In Helmut Seidl, editor, *European Symposium on Programming, ESOP 2012*, volume 7211 of *LNCS*, pages 397–416. Springer, 2012. doi:10.1007/978-3-642-28869-2_20.
- 22 Jeffrey Kegler. Marpa, a practical general parser: the recognizer, 2023. arXiv:1910.08129.
- 23 Adam Koprowski and Henri Binsztok. TRX: A formally verified parser interpreter. *Log. Methods Comput. Sci.*, 7(2), 2011. doi:10.2168/LMCS-7(2:18)2011.
- 24 Alexander Krauss. Recursive definitions of monadic functions. In Ekaterina Komendantskaya, Ana Bove, and Milad Niqui, editors, *Partiality and Recursion in Interactive Theorem Provers, PAR@ITP 2010*, volume 5 of *EPiC Series*, pages 1–13. EasyChair, 2010. doi:10.29007/1mdt.
- 25 Sam Lasser, Chris Casinghino, Kathleen Fisher, and Cody Roux. A Verified LL(1) Parser Generator. In John Harrison, John O’Leary, and Andrew Tolmach, editors, *Interactive Theorem Proving (ITP 2019)*, volume 141 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 24:1–24:18, Dagstuhl, Germany, 2019. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ITP.2019.24.
- 26 Joop M.I.M. Leo. A general context-free parsing algorithm running in linear time on every LR(k) grammar without using lookahead. *Theoretical Computer Science*, 82(1):165–176, 1991. doi:10.1016/0304-3975(91)90180-A.
- 27 Philippe McLean and R. Nigel Horspool. A faster Earley parser. In Tibor Gyimóthy, editor, *Compiler Construction, CC’96*, volume 1060 of *LNCS*, pages 281–293. Springer, 1996. doi:10.1007/3-540-61053-7_68.
- 28 Tobias Nipkow and Gerwin Klein. *Concrete Semantics with Isabelle/HOL*. Springer, 2014. URL: <http://concrete-semantics.org>.
- 29 Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
- 30 Steven Obua. Local lexing. *Archive of Formal Proofs*, 2017. , Formal proof development. URL: <https://isa-afp.org/entries/LocalLexing.html>.

- 31 Steven Obua, Phil Scott, and Jacques Fleuriot. Local lexing, 2017. [arXiv:1702.03277](https://arxiv.org/abs/1702.03277).
- 32 Sinan Polat, Merve Selcuk-Simsek, and Ilyas Cicekli. A modified Earley parser for huge natural language grammars. *Res. Comput. Sci.*, 117:23–35, 2016. URL: https://rcs.cic.ipn.mx/2016_117/A%20Modified%20Earley%20Parser%20for%20Huge%20Natural%20Language%20Grammars.pdf.
- 33 Martin Rau. Earley parser. *Archive of Formal Proofs*, July 2023. , Formal proof development. URL: https://devel.isa-afp.org/entries/Earley_Parser.html.
- 34 Tom Ridge. Simple, functional, sound and complete parsing for all context-free grammars. In Jean-Pierre Jouannaud and Zhong Shao, editors, *Certified Programs and Proofs, CPP 2011*, volume 7086 of *LNCS*, pages 103–118. Springer, 2011. doi:10.1007/978-3-642-25379-9_10.
- 35 Elizabeth Scott. SPPF-style parsing from Earley recognisers. *Electronic Notes in Theoretical Computer Science*, 203(2):53–67, 2008. Workshop on Language Descriptions, Tools, and Applications (LDTA 2007). doi:10.1016/j.entcs.2008.03.044.
- 36 Masaru Tomita. *Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems*. Kluwer Academic Publishers, USA, 1985.
- 37 Masaru Tomita. An efficient augmented-context-free parsing algorithm. *Comput. Linguist.*, 13(1-2):31–46, January 1987.

Abstractions for Multi-Sorted Substitutions

Hannes Saffrich  

University of Freiburg, Germany

Abstract

Formalizing a typed programming language in a proof assistant requires to choose representations for variables and typing. Variables are often represented as de Bruijn indices, where substitution is usually defined in terms of renamings to allow for proofs by structural induction. Typing can be represented extrinsically by defining untyped terms and a typing relation, or intrinsically by combining syntax and typing into a single definition of well-typed terms. For extrinsic typing, there is again a choice between extrinsic scoping, where terms and the notion of free variables are defined separately, and intrinsic scoping, where terms are indexed by their free variables.

This paper describes an Agda framework for formalizing programming languages with extrinsic typing, intrinsic scoping, and de Bruijn Indices for variables. The framework supports object languages with arbitrary many variable sorts and dependencies, making it suitable for polymorphic languages and dependent types. Given an Agda definition of syntax and typing, the framework derives substitution operations and lemmas for untyped terms, and provides an abstraction to prove type preservation of these operations with just a single lemma. The key insights behind the framework are the use of multi-sorted syntax definitions, which enable parallel substitutions that replace all variables of all sorts simultaneously, and abstractions that unify the definitions, compositions, typings, and type preservation lemmas of multi-sorted renamings and substitutions. Case studies have been conducted to prove subject reduction for System F with subtyping, dependently typed lambda calculus, and lambda calculus with pattern matching.

2012 ACM Subject Classification Theory of computation \rightarrow Type theory; Software and its engineering \rightarrow Syntax; Theory of computation \rightarrow Logic and verification

Keywords and phrases Agda, Metatheory, Framework

Digital Object Identifier 10.4230/LIPIcs.ITP.2024.32

Supplementary Material

Software (Source Code): <https://github.com/morphism/kitty-supplement>

archived at `swh:1:dir:8ee2a0aeb901498fca00f6d379099386fab74c60`

1 Introduction

Formalizing programming languages in proof assistants quickly gets repetitive. Almost every programming language supports variables with static binding, and hence requires numerous definitions and lemmas related to variable substitution.

Additionally, repetition can also occur within a single formalization. This can be seen with polymorphic languages, where multiple sorts of variables are present. Consider for example System F, which supports both expression- and type-variables. With a naive approach, the whole substitution machinery needs to be duplicated three times! We need to substitute expression-variables in expressions, type-variables in types, but also type-variables in expressions. Even worse, having two substitutions acting on expressions requires to also prove lemmas about their interactions. If we would additionally introduce kind-variables, we would end up with a total of six duplications of the substitution machinery and corresponding interaction lemmas!



© Hannes Saffrich;

licensed under Creative Commons License CC-BY 4.0

15th International Conference on Interactive Theorem Proving (ITP 2024).

Editors: Yves Bertot, Temur Kutsia, and Michael Norrish; Article No. 32; pp. 32:1--32:19

Leibniz International Proceedings in Informatics



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

Further repetition can occur due to the choice of variable representation. For example, for de Bruijn indices, substitution is usually defined in terms of renamings¹ to allow for structural induction. With a naive approach, this would again double the amount of substitution machinery, as all definitions and lemmas need to be written first for renamings and then again for substitutions.

Similarly, for a typed language formalized via extrinsic typing we need type preservation lemmas for each substitution and renaming operation, which again doubles the number of definitions.

If one is not careful, a formalization of a language with type and kind polymorphism can easily end up with 24 slightly changed copies of the whole substitution machinery! Clearly, this situation is in need of automation.

Our framework approaches this problem by using a combination of abstractions and reflection in the context of extrinsically typed, intrinsically scoped syntax with de Bruijn indices. The user can write a language specification using regular Agda definitions (no encodings via generic programming[2]) and our framework derives definitions and lemmas for untyped substitution, and provides an abstraction to prove type preservation for all substitution operations with only a single lemma. For System F, our framework allows to prove subject reduction with only a single handwritten lemma for type preservation.

Compared to standard practices, we do not derive substitutions for each of the variable sorts and syntactic categories, but instead use a novel approach for defining syntax, which directly supports substitutions that replace all variable sorts in parallel and can be applied to all syntactic categories. By further unifying renamings and substitutions, we gain the ability to talk abstractly about any kind of renaming or substitution that can occur in the formalization. This generality is key to then define abstractions for typing and type preservation on the same level of generality, allowing to prove type preservation for renamings and substitutions of all variable sorts and syntactic categories in a single lemma for many typing relations, including those of our case studies.

1.1 Structure

The rest of this paper introduces our framework using System F as a running example for a substitution-preserves-typing proof.

Code of the framework is displayed in gray boxes. Code of examples is displayed in yellow boxes. Code of the System F formalization is displayed without boxes. The latter is the only code a user of our framework has to write.

In this paper, we present a simplified version of the full framework, focusing on the core concepts. We present all necessary definitions and lemmas, but omit some proofs for the sake of space. The omitted proofs can be found in our supplementary material, which includes the simplified framework (365 lines of code) and the System F formalization (212 lines of code, where 79 can be derived by using the full framework). The full framework including the case studies is available on Github: <https://github.com/m0rphism/kitty>

The rest of this paper is structured as follows: Section 2 introduces the multi-sorted syntax and compares it to the more common unsorted syntax. Section 3 introduces multi-sorted substitutions and renamings, and an abstraction to unify them. Section 4 introduces composition of multi-sorted substitutions and renamings, and an abstraction to unify all four compositions. Section 5 shows how to define types, type contexts, and typing relations, and

¹ *Renamings* are substitutions that are only allowed to replace variables with variables

presents an abstraction for unifying type preservation lemmas for renamings and substitutions. Section 6 describes the class of object languages covered by our reflection algorithm. Section 7 discusses our case studies. Section 8 discusses related work. Section 9 concludes.

1.2 Contributions

1. a novel approach for formalizing intrinsically-scoped syntax with multiple variable sorts as a special kind of intrinsically-typed syntax, we call *multi-sorted syntax*;
2. a novel abstraction for composition and its metatheory, unifying the four compositions between renamings and substitutions;
3. a novel abstraction for typing, unifying type preservation of renaming and substitution;
4. a formalized specification of a large class of object languages for which untyped substitution and lemmas can be derived generically.
5. an implementation as an Agda framework featuring a reflection algorithm, representation independence for substitutions and type contexts, heterogeneous equality between renamings and substitutions, and absence of postulated axioms.
6. three case studies in using our approach to prove subject reduction for System F with subtyping, a dependently-typed lambda calculus, and pattern matching.

2 Syntax

2.1 Unsorted Syntax

The following shows a typical intrinsically-scoped syntax of System F:

```
data Kind : Set where
  * : Kind -- Type Kind
data Type (n : ℕ) : Set where
  ' _ : Fin n → Type n -- Type variable
  ∀[α:_]_ : Kind → Type (suc n) → Type n -- Universal quantification
  _⇒_ : Type n → Type n → Type n -- Function type
data Expr (n m : ℕ) : Set where
  ' _ : Fin m → Expr n m -- Expression variable
  λx_ : Expr n (suc m) → Expr n m -- Expression abstraction
  Λα_ : Expr (suc n) m → Expr n m -- Type abstraction
  _ · _ : Expr n m → Expr n m → Expr n m -- Expression application
  _ • _ : Expr n m → Type n → Expr n m -- Type application
```

Types are indexed by the number of free type variables n . Expressions are additionally indexed by the number of free expression variables m . Variables $'_$ are represented as de Bruijn indices, where $\text{Fin } n$ is the type of n elements.

We identify two drawbacks with this style of syntax:

1. the syntactic categories (Kind , Type , and Expr) have different types, which makes it difficult to treat them uniformly; and
2. the different sorts of variables are modeled separately, which requires to define not just type-in-type and expression-in-expression substitution, but also type-in-expression substitution. Consequently, interaction lemmas between the substitutions are required.

To avoid these drawbacks, we instead use a multi-sorted syntax.

2.2 Multi-Sorted Syntax

A multi-sorted syntax is defined by a single type of sort-indexed terms.

A sort describes to which syntactic category a term belongs and is itself indexed by a sort type, which describes whether the syntax permits variables of this sort:

```
data SortTy : Set where Var NoVar : SortTy
```

```
data Sort : SortTy → Set where -- Our syntax supports:
  _ : Sort Var -- expressions and expression variables;
  ≈ : Sort Var -- types and type variables; and
  ⊤ : Sort NoVar -- kinds, but no kind variables.
```

The term type $S \vdash s$ is indexed by its sort s and the sorts of its free variables S . For example, $[\approx, \approx] \vdash$ is the type of expressions $()$ with two free type-variables (\approx) .

```
data _⊢_ : List (Sort Var) → Sort st → Set where
  ' _ : S ∋ s → S ⊢ s -- Expression and type variables
  λx_ : ( :: S) ⊢ → S ⊢ -- Expression abstraction
  Λα_ : (≈ :: S) ⊢ → S ⊢ -- Type abstraction
  ∀[α:_]_ : S ⊢ ⊤ → (≈ :: S) ⊢ ≈ → S ⊢ ≈ -- Universal quantification
  _·_ : S ⊢ → S ⊢ → S ⊢ -- Expression application
  _•_ : S ⊢ → S ⊢ ≈ → S ⊢ -- Type application
  _⇒_ : S ⊢ ≈ → S ⊢ ≈ → S ⊢ ≈ -- Function type
  * : S ⊢ ⊤ -- Type kind
```

The notation $_⊢_$ is often used for terms in intrinsically-typed languages. This is no accident: in effect, we defined an intrinsically-typed language with the twist that the typing relation assures exactly that the syntactic categories are followed. Sorts s correspond to types, and lists of sorts S correspond to type environments.

As it is typical in intrinsic typing, variables are represented as typed (in our case sorted) de Bruijn indices $S \ni s$, i.e. values of the usual proof-relevant list-membership relation:

```
data _∋_ {ℓ} {A : Set ℓ} : List A → A → Set ℓ where
  zero : ∀ {xs x} → (x :: xs) ∋ x
  suc : ∀ {xs x y} → xs ∋ x → (y :: xs) ∋ x
```

Note that there is no straightforward way to construct a multi-sorted syntax with intrinsic typing: in a direct translation, the type of terms $_⊢_$ would be indexed by itself, which most proof assistants forbid to avoid breaking logical consistency.

2.3 A Structure for Multi-Sorted Syntax

The regularity of the multi-sorted syntax makes it easy to define a structure for arbitrary syntaxes, i.e. syntaxes with arbitrarily many syntactic categories and variable types:

```
record Syntax : Set1 where
  field Sort : SortTy → Set
  _⊢_ : ∀ {st} → List (Sort Var) → Sort st → Set
  ' _ : ∀ {S} {s : Sort Var} → S ∋ s → S ⊢ s
  '-injective : ∀ {S s} {x y : S ∋ s} → ' x ≡ ' y → x ≡ y
```

The first three fields record the definitions of sorts, terms, and variable introduction. The last field records that variable introduction $_⊢_$ is injective, which is trivially true for constructors. `Syntax` has type `Set1`, because the `Sort` and `_⊢_` fields have type `Set`.

The instantiation for our System F syntax is straightforward:

SystemF-Syntax : Syntax

SystemF-Syntax = record { Sort = Sort ; $_ \vdash _ = _ \vdash _ ; _ = _ ; \text{-injective} = \lambda \{ \text{refl} \rightarrow \text{refl} \} \}$

3 Renamings & Substitutions

3.1 Multi-Sorted Renamings & Substitutions

Working with a sort-indexed syntax allows us to define renamings and substitutions that replace all variables of all sorts simultaneously:

$_ \rightarrow_r _ \rightarrow_s _ : \text{List (Sort Var)} \rightarrow \text{List (Sort Var)} \rightarrow \text{Set}$

$S_1 \rightarrow_r S_2 = \forall s \rightarrow S_1 \ni s \rightarrow S_2 \ni s$

$S_1 \rightarrow_s S_2 = \forall s \rightarrow S_1 \ni s \rightarrow S_2 \vdash s$

A renaming $S_1 \rightarrow_r S_2$ maps variables from S_1 to variables from S_2 . A substitution $S_1 \rightarrow_s S_2$ maps variables from S_1 to terms with free variables from S_2 .

This representation has the benefit that there is no combinatory explosion of substitutions and renamings, e.g. no extra lemmas have to be proved between an expression-in-expression and a type-in-expression substitution, because both are simply substitutions.

3.2 Unifying Renamings & Substitutions

To avoid the duplication between renamings and substitutions, McBride[6, 15] introduced the *kit* abstraction. A kit is a structure that allows to abstract over whether something is a term or a variable. The intention is to instantiate this structure exactly twice (once for variables and once for terms), and then write definitions, which are parameterized over a kit and consequently can be used for both variables and terms.

```
record Kit ( $\_ \ni \_ / \vdash \_ : \text{List (Sort Var)} \rightarrow \text{Sort Var} \rightarrow \text{Set}$ ) : Set where
  field id/'      :  $S \ni s \rightarrow S \ni / \vdash s$ 
        '/id      :  $S \ni / \vdash s \rightarrow S \vdash s$ 
        wk        :  $\forall s' \rightarrow S \ni / \vdash s \rightarrow (s' :: S) \ni / \vdash s$ 
        '/-is-'   :  $\forall (x : S \ni s) \rightarrow \text{'/id (id/' x)} \equiv \text{' x}$ 
        id/'-injective :  $\text{id/' } x_1 \equiv \text{id/' } x_2 \rightarrow x_1 \equiv x_2$ 
        '/id-injective :  $\forall \{x/t_1 x/t_2 : S \ni / \vdash s\} \rightarrow \text{'/id } x/t_1 \equiv \text{'/id } x/t_2 \rightarrow x/t_1 \equiv x/t_2$ 
        wk-id/'   :  $\forall s' (x : S \ni s) \rightarrow \text{wk } s' (\text{id/' } x) \equiv \text{id/' (suc x)}$ 
```

As we intend to have exactly two *Kit* instances, we choose names of the form x/y , where x is the name we choose for the variable instance, and y is the name we choose for the term instance. For example the parameter type $_ \ni _ / \vdash _$ will be instantiated to $_ \ni _$ for the variable kit, and to $_ \vdash _$ for the term kit.

A kit consists of the following components:

- **id/'** converts a variable $S \ni s$ into a $S \ni / \vdash s$. For the variable kit, $_ \ni _ / \vdash _$ is instantiated to $_ \ni _$, so this operation is the identity. For the term kit, $_ \ni _ / \vdash _$ is instantiated to $_ \vdash _$, so this operation is the variable constructor **'_**.
- **'/id** converts a $S \ni / \vdash s$ into a term $S \vdash s$ and is analogous to the **id/'** operation.
- **wk** shifts the de Bruin indices in a variable or term. The new, unused variable **zero** can assume any sort s' . For variables, **wk** is the successor **suc**. For terms, **wk** means applying a shifting renaming to the term.

32:6 Abstractions for Multi-Sorted Substitutions

$\begin{aligned} _ \rightarrow_k _ &: \text{List (Sort Var)} \rightarrow \text{List (Sort Var)} \rightarrow \text{Set} \\ S_1 \rightarrow_k S_2 &= \forall s \rightarrow S_1 \ni s \rightarrow S_2 \ni / \vdash s \end{aligned}$	$\begin{aligned} \text{weaken} &: \forall s \rightarrow S \rightarrow_k (s :: S) \\ \text{weaken } s _ x &= \text{wk } s (\text{id}/' x) \end{aligned}$
$\begin{aligned} _ \& _ &: S_1 \ni s \rightarrow S_1 \rightarrow_k S_2 \rightarrow S_2 \ni / \vdash s \\ x \& \phi &= \phi _ x \end{aligned}$	$\begin{aligned} _ \sim _ &: (\phi_1 \phi_2 : S_1 \rightarrow_k S_2) \rightarrow \text{Set} \\ _ \sim _ \{S_1\} \phi_1 \phi_2 &= \forall s (x : S_1 \ni s) \rightarrow \\ &\quad \phi_1 s x \equiv \phi_2 s x \end{aligned}$
$\begin{aligned} _ \uparrow _ &: S_1 \rightarrow_k S_2 \rightarrow \forall s \rightarrow (s :: S_1) \rightarrow_k (s :: S_2) \\ (\phi \uparrow s) _ \text{zero} &= \text{id}/' \text{zero} \\ (\phi \uparrow s) _ (\text{suc } x) &= \text{wk } _ (\phi _ x) \end{aligned}$	$\begin{aligned} \text{postulate } \sim\text{-ext} &: \forall \{\phi_1 \phi_2 : S_1 \rightarrow_k S_2\} \\ &\rightarrow \phi_1 \sim \phi_2 \rightarrow \phi_1 \equiv \phi_2 \end{aligned}$
$\begin{aligned} (_ _) &: S \ni / \vdash s \rightarrow (s :: S) \rightarrow_k S \\ (_ x/t _) _ \text{zero} &= x/t \\ (_ x/t _) _ (\text{suc } x) &= \text{id}/' x \end{aligned}$	$\begin{aligned} \text{id} &: S \rightarrow_k S \\ \text{id } s x &= \text{id}/' x \end{aligned}$
	$\text{id}\uparrow\text{-id} : (\text{id } \{S\} \uparrow s) \sim \text{id } \{s :: S\}$

■ **Figure 1** Map Operations.

- **'/'-is-** states that converting a variable first to a “variable-or-term” and then further to a term is the same as converting it directly to a term using the variable constructor **'_'**.
- **'/id-injective** and **id/'-injective** state that **'/id** and **id/'** are injective.
- **wk-id/'** characterizes the behaviour of the **wk** function by how it acts on variables: injecting a variable and then shifting it, is the same as injecting a shifted variable.

Figure 1 shows the usual operations for renamings and substitutions. The definitions are included directly in the record module of **Kit**, so they are implicitly parameterized over a kit. The type $S_1 \rightarrow_k S_2$ unifies renamings $S_1 \rightarrow_r S_2$ and substitutions $S_1 \rightarrow_s S_2$. We call a value of type $S_1 \rightarrow_k S_2$ a *map* and use the meta-variable ϕ for it. The operation $\phi \& \times$ applies a map to a variable. The operation $\phi \uparrow s$ lifts a map under a binder of sort s . The operation $(_ x/t _)$ constructs a singleton map that replaces **zero** with x/t and decreases all other variables by one. The **weaken** map increases all variables by one. The $\phi_1 \sim \phi_2$ type expresses extensional equality of maps. For simplicity, we postulate functional extensionality **~ext**.² There is an identity map **id**. The lemma **id \uparrow -id** states that a lifted identity map is again an identity map.

To make it easier to talk about a specific kit, we introduce the following notations:

- we write $S_1 \text{ -}[K] \rightarrow S_2$ for the $S_1 \rightarrow_k S_2$ of some specific **Kit** K ; and
- we write $S \ni / \vdash [K] s$ for the $S \ni / \vdash s$ of some specific **Kit** K .

The operation of applying a map to a term depends on the concrete object language. It is captured by the following structure:

```
record Traversal : Set1 where
  field ..._ :  $\forall \{K : \text{Kit } \_ \ni / \vdash \_ \} \rightarrow S_1 \vdash s \rightarrow S_1 \text{ -}[ K ] \rightarrow S_2 \rightarrow S_2 \vdash s$ 
  ...-var :  $\forall \{K : \text{Kit } \_ \ni / \vdash \_ \} (x : S_1 \ni s) (\phi : S_1 \text{ -}[ K ] \rightarrow S_2) \rightarrow (' x) \dots \phi \equiv \text{'/id } (x \& \phi)$ 
  ...-id :  $\forall \{K : \text{Kit } \_ \ni / \vdash \_ \} (t : S \vdash s) \rightarrow t \dots \text{id } \{K\} \equiv t$ 
```

² The actual implementation does *not* use any postulates. Functional extensionality can be avoided by proving that $\phi_1 \sim \phi_2$ implies $(t \dots \phi_1) \equiv (t \dots \phi_2)$, i.e. that if two maps are extensionally equal, then their applications (\dots) to the same term are intensionally equal. The downside of this approach is boilerplate, because for each operation on maps, it needs to be proved that the operation preserves extensional equality, e.g. that $\phi_1 \sim \phi_2$ implies $(\phi_1 \uparrow s) \sim (\phi_2 \uparrow s)$. None of those lemmas are necessary with functional extensionality.

The fields of this structure have the following meaning:

- $t \dots \phi$ applies the map ϕ (a renaming or substitution) to the term t .
- $\dots\text{-var}$ states that applying a map ϕ to a variable term $'x$, is the same as applying ϕ to the variable x , and then converting the result to a term via id' .
- $\dots\text{-id}$ states that applying the identity map id to a term does not change the term.

Finally, we define the actual kit instances. The variable kit definition is straightforward:

```

Kr : Kit _∃_
Kr = record { id/'      = λ x → x      ; '/id      = '_
              ; wk       = λ s' x → suc x ; '/-is-'   = λ x → refl
              ; id/'-injective = λ eq → eq   ; '/id-injective = '-injective
              ; wk-id/'    = λ s' x → refl }

```

The term kit requires both the variable kit and the **Traversal** to be defined, because shifting a term with **wk** means applying the shifting renaming to the term. Hence, we define the term kit in the record module of **Traversal**:

```

Ks : Kit _/__
Ks = record { id/'      = '_          ; '/id      = λ t → t
              ; wk       = λ s' t → t ... weaken { Kr } s' ; '/-is-'   = λ x → refl
              ; id/'-injective = '-injective ; '/id-injective = λ eq → eq
              ; wk-id/'    = λ s' x → ...-var x (weaken { Kr } s') }

```

3.3 Instantiation for System F

In this subsection, we show how to instantiate the **Traversal** abstraction for System F. In practice, this is done by our reflection algorithm automatically (Section 6), but it can be instructive to see what happens under the hood.

First, we define the operation of applying a map to a term:

```

_..._ : ∀ { K : Kit _∃/_/_ } → S1 ⊢ s → S1 -[ K ]→ S2 → S2 ⊢ s
(' x)      ... φ = '/id (x & φ)
(λx t)     ... φ = λx (t ... (φ ↑))
(Λα t)     ... φ = Λα (t ... (φ ↑ ≈))
(∀[α: t1] t2) ... φ = ∀[α: t1 ... φ] (t2 ... (φ ↑ ≈))
(t1 · t2) ... φ = (t1 ... φ) · (t2 ... φ)
(t1 • t2) ... φ = (t1 ... φ) • (t2 ... φ)
(t1 ⇒ t2) ... φ = (t1 ... φ) ⇒ (t2 ... φ)
*          ... φ = *

```

The interesting cases are those with variables and binders:

- In the variable case $('x) \dots \phi$, we first apply the map ϕ to the variable x . If ϕ is a renaming, we get back a variable and need to apply the variable constructor $'_$. If ϕ is a substitution, we get back a term that we can use directly. This is exactly what 'id does.
- In cases where the operation needs to go under a binder, like $(\lambda x e) \dots \phi$, we lift the map using '↑ to account for the bound variable before we apply it to the subterm.

We then prove that applying an identity map does not change the term:

```

...id : ∀ { K : Kit _∃/_/_ } (t : S ⊢ s) → t ... id ≡ t

```

```

...id (' x) = '/-is-' x
...id (λx e) = λx (e ... (id ↑)) ≡⟨ cong (λ φ → λx (e ... φ)) (~~ext id↑~id) ⟩
    λx (e ... id) ≡⟨ cong (λ e → λx e) (...id e) ⟩
    λx e

```

We only display and discuss the interesting cases:

- in the variable case ' x, the `id/` from the identity map meets the `/id` from the traversal operation, so we need to use `'-is-`.
- in the lambda abstraction case `λx e`, the traversal lifts the identity under its binder. Here we need to use `id↑-id` to show that a lifted identity map is again an identity map.

Finally, we instantiate the `Traversal` structure:

```

SystemF-Traversal : Traversal
SystemF-Traversal = record { _..._ = _..._ ; ...id = ...id ; ...var = λ x φ → refl }

```

3.4 Extension Kits

As we defined the `Kit` structure before the `Traversal` structure, the fields of `Kit` could not use map application `_..._` in their types. This prevented us to include another useful axiom into the `Kit` structure. As this axiom also needs to be proved separately for variables and terms, we define a new structure for it which extends a `Kit`:

```

record WkKit (K : Kit _∃/!_) : Set₁ where
field wk-/id : ∀ s {S s'} (x/t : S ∃/! s') → '/id x/t ... weaken s ≡ '/id (wk s x/t)

```

The `wk-/id` axiom explains the `wk` function by how it acts on *terms*. It is the counterpiece to the `Kit` axiom `wk-id/`, which explains the `wk` function by how it acts on *variables*. This lemma is useful for proving extensional equalities between maps involving weakening, where `/id-injective` allows to add `/id` on both sides of the equation, such that `wk-/id` can be used to make further progress. The instantiations of the `WkKit` are straightforward:

```

Wr : WkKit Kr ; Ws : WkKit Ks
Wr = record { wk-/id = λ s x → ...var x (weaken s) }
Ws = record { wk-/id = λ s t → refl }

```

As the variable and term `Kits` are the only two `Kits`, and both have `WkKit` instances, it is always safe to assume that a `Kit` also supports the `WkKit` extension.

4 Map Composition

In this section, we extend our framework with an abstraction for the composition of arbitrary maps. The core property of composition is the `fusion` lemma, which states that applying two maps ϕ_1 and ϕ_2 in sequence to a term t , is the same as applying their composition $(\phi_1 \cdot_k \phi_2)$ to t , i.e. $(t \dots \phi_1) \dots \phi_2 \equiv t \dots (\phi_1 \cdot_k \phi_2)$. This property gives our framework the ability to reason about the application of multiple maps by reasoning about the application of a single map. As such it forms the basis for all lemmas involving multiple maps, e.g. that applying a weakening and then a singleton substitution cancel each other out.

As we defined substitution in terms of renamings, we need to consider all four compositions between renamings and substitutions. While the composition operations can be defined independently of each other, the `fusion` lemma for two substitutions, depends on the `fusion` lemmas for a renaming and a substitution, which in turn depend on the `fusion` lemma for two renamings.

Previous work on kits[6] addresses this issue by duplicating the definitions and using tactics to reduce boilerplate in proofs. In contrast, we define structures similar to **Kit** and **Traversal**, which allow us to abstract over all four compositions and use the same trick as before to eliminate the dependencies. With the help of a general map composition, we can prove lemmas about the interactions of general maps, which is crucial for proving a type preservation lemma for general map application instead of individual lemmas for renamings and substitutions.

4.1 An Examination of Composition

To motivate our abstraction, we first look at the four compositions individually:

$$\begin{array}{ll}
 _r \cdot _r _ : (S_1 \rightarrow_r S_2) \rightarrow (S_2 \rightarrow_r S_3) \rightarrow (S_1 \rightarrow_r S_3) & (\phi_1 \cdot_r \phi_2) _ x = (x \& \phi_1) \& \phi_2 \\
 _r \cdot _s _ : (S_1 \rightarrow_r S_2) \rightarrow (S_2 \rightarrow_s S_3) \rightarrow (S_1 \rightarrow_s S_3) & (\phi_1 \cdot_r \phi_2) _ x = (x \& \phi_1) \& \phi_2 \\
 _s \cdot _r _ : (S_1 \rightarrow_s S_2) \rightarrow (S_2 \rightarrow_r S_3) \rightarrow (S_1 \rightarrow_s S_3) & (\phi_1 \cdot_s \phi_2) _ x = (x \& \phi_1) \cdots \phi_2 \\
 _s \cdot _s _ : (S_1 \rightarrow_s S_2) \rightarrow (S_2 \rightarrow_s S_3) \rightarrow (S_1 \rightarrow_s S_3) & (\phi_1 \cdot_s \phi_2) _ x = (x \& \phi_1) \cdots \phi_2
 \end{array}$$

The definitions reveal two interesting properties:

1. If we compose two maps ϕ_1 and ϕ_2 , then the resulting map is a renaming, iff both ϕ_1 and ϕ_2 are renamings. In other words: if ϕ_1 is a K_1 -map and ϕ_2 is a K_2 -map, then the result is a $(K_1 \sqcup K_2)$ -map, where \sqcup refers to the lattice for $\{K_r, K_s\}$ generated by $K_r < K_s$.
2. All four compositions first apply ϕ_1 to x , and then apply ϕ_2 to the result. If ϕ_1 is a renaming, this result is another variable, but if ϕ_1 is a substitution, this result is a term.

With the **Kit** abstraction, we can easily abstract over ϕ_2 being a renaming or a substitution:

$$\begin{array}{ll}
 _r \cdot _ _ : (S_1 \rightarrow_r S_2) \rightarrow (S_2 \text{-[K]} \rightarrow S_3) \rightarrow (S_1 \text{-[K]} \rightarrow S_3) & (\phi_1 \cdot_r \phi_2) _ x = (x \& \phi_1) \& \phi_2 \\
 _s \cdot _ _ : (S_1 \rightarrow_s S_2) \rightarrow (S_2 \text{-[K]} \rightarrow S_3) \rightarrow (S_1 \rightarrow_s S_3) & (\phi_1 \cdot_s \phi_2) _ x = (x \& \phi_1) \cdots \phi_2
 \end{array}$$

But to abstract over ϕ_1 , the **Kit** abstraction is not sufficient: while it allows us to abstract over what we are applying, i.e. a renaming or a substitution, it does not allow us to abstract over what we are applying it to, i.e. a variable or a term. For the latter we have two distinct operations $_ \& _$ and $_ \cdots _$.

To fill this gap, we introduce a new abstraction that we call a *compose kit* (**CKit**), which provides an operation $_ \& / \cdots _$ that unifies $_ \& _$ and $_ \cdots _$. This allows us to define a general composition as follows:

$$\begin{array}{l}
 _ \cdot _k _ : S_1 \text{-[K}_1\text{]} \rightarrow S_2 \rightarrow S_2 \text{-[K}_2\text{]} \rightarrow S_3 \rightarrow S_1 \text{-[K}_1 \sqcup K_2\text{]} \rightarrow S_3 \\
 (\phi_1 \cdot_k \phi_2) _ x = (x \& \phi_1) \& / \cdots \phi_2
 \end{array}$$

4.2 An Abstraction for Composition

A compose kit **CKit** $K_1 K_2 K_1 \sqcup K_2$ describes the operations necessary for defining the composition of a K_1 -map with a K_2 -map that results in a $K_1 \sqcup K_2$ -map:

```

record CKit (K1 : Kit _∅/!_ ) (K2 : Kit _∅/!_ ) (K1⊔K2 : Kit _∅/!_ ) : Set where
field _&/..._ : S1 ∅/! [K1] s → S1 -[K2] → S2 → S2 ∅/! [K1⊔K2] s
    &/... : (x/t : S1 ∅/! [K1] s) (φ : S1 -[K2] → S2) →
        '/id (x/t &/... φ) ≡ '/id x/t ... φ
    &/...wk↑ : (x/t : S1 ∅/! [K1] s) (φ : S1 -[K2] → S2) →
        wk s' (x/t &/... φ) ≡ wk s' x/t &/... (φ ↑ s')

```

The third parameter $K_1 \sqcup K_2$ can be seen as a functional dependency[12] and is determined by the choice of K_1 and K_2 . The fields of a compose kit have the following meaning:

32:10 Abstractions for Multi-Sorted Substitutions

- The $_&/\dots_$ operation takes a variable or term x/t (according to K_1) and a renaming or substitution ϕ (according to K_2) and applies ϕ to x/t resulting in a variable or term (according to $K_1 \sqcup K_2$). From this operation we derive map composition $_ \cdot_k _$ as shown in the previous subsection.
- The $_&/\dots\dots$ lemma describes the behavior of $_&/\dots_$ in terms of $_ \dots _$, allowing subsequent lemmas to make use of the lemmas that we have already proved for $_ \dots _$.
- The $_&/\dots\text{-wk}\uparrow$ lemma states that applying a map and then weakening is the same as weakening first and then lifting the map over the variable introduced by the weakening. From this lemma, we can derive that lifting distributes over composition:

$$\text{dist}\uparrow\text{-}\cdot : \forall s (\phi_1 : S_1 \text{-}[K_1] \rightarrow S_2) (\phi_2 : S_2 \text{-}[K_2] \rightarrow S_3) \rightarrow \\ ((\phi_1 \cdot_k \phi_2) \uparrow s) \sim ((\phi_1 \uparrow s) \cdot_k (\phi_2 \uparrow s))$$

A **CTraversal** provides a **fusion** lemma that works for the composition of any **CKit**:

```
record CTraversal : Set1 where
  field fusion :
    ∀ { K1 : Kit _ $\exists$ /!-1 } { K2 : Kit _ $\exists$ /!-2 } { K : Kit _ $\exists$ /!- } { W1 : WkKit K1 }
      { C : CKit K1 K2 K } (t : S1  $\vdash$  s) ( $\phi_1$  : S1  $\text{-}[K_1] \rightarrow S_2$ ) ( $\phi_2$  : S2  $\text{-}[K_2] \rightarrow S_3$ )  $\rightarrow$ 
      (t  $\dots$   $\phi_1$ )  $\dots$   $\phi_2 \equiv t \dots (\phi_1 \cdot_k \phi_2)$ 
```

Given a **CTraversal**, we can prove the usual lemmas about interactions of multiple maps:

- A map ϕ followed by a weakening is equivalent to a weakening followed by ϕ that has been lifted over the weakened variable:

$$\text{---}\uparrow\text{-wk} : \forall \{ K : Kit _ \exists / ! - \} \{ W : WkKit K \} \{ C_1 : CKit K K_r K \} \{ C_2 : CKit K_r K K \} \\ (t : S_1 \vdash s) (\phi : S_1 \text{-}[K] \rightarrow S_2) s \rightarrow \\ t \dots \phi \dots \text{weaken } s \equiv t \dots \text{weaken } s \dots (\phi \uparrow s)$$

- A weakening followed by a singleton substitution act as an identity map:

$$\text{wk-cancels-}(\emptyset)\text{---} : \forall \{ K : Kit _ \exists / ! - \} (t : S \vdash s') (x/t : S \exists / ! [K] s) \rightarrow \\ t \dots \text{weaken } s \dots (x/t) \equiv t$$

- A singleton map can be swapped with any map ϕ :

$$\text{dist}\uparrow\text{-}(\emptyset)\text{---} : \forall \{ K_1 : Kit _ \exists / ! -_1 \} \{ K_2 : Kit _ \exists / ! -_2 \} \{ K : Kit _ \exists / ! - \} \\ \{ W_1 : WkKit K_1 \} \{ W_2 : WkKit K_2 \} \\ \{ C_1 : CKit K_1 K_2 K \} \{ C_2 : CKit K_2 K K \} \\ (t : (s :: S_1) \vdash s') (x/t : S_1 \exists / ! [K_1] s) (\phi : S_1 \text{-}[K_2] \rightarrow S_2) \rightarrow \\ t \dots (x/t) \dots \phi \equiv t \dots (\phi \uparrow s) \dots (x/t \&/\dots \phi)$$

Similarly, as it was the case for the **Kit** and **Traversal** structures, the idea is that we instantiate the **CTraversal** for our object language, and in return the framework defines the concrete **CKit** instances for us. Hence, we define the **CKit** instances in the record module of **CTraversal**:

```
Cr : { K : Kit _ $\exists$ /!- }  $\rightarrow$  CKit Kr K K
Cr = record { _&/\dots_ = _&_
  ; &/\dots\dots =  $\lambda$  x  $\phi \rightarrow$  sym (---var x  $\phi$ )
  ; &/\dots\text{-wk}\uparrow =  $\lambda$  x  $\phi \rightarrow$  refl }
Cs : { K : Kit _ $\exists$ /!- } { C : CKit K Kr K } { W : WkKit K }  $\rightarrow$  CKit Ks K Ks
Cs = record { _&/\dots_ = _ \dots _
  ; &/\dots\dots =  $\lambda$  t  $\phi \rightarrow$  refl
  ; &/\dots\text{-wk}\uparrow =  $\lambda$  t  $\phi \rightarrow$  --- $\uparrow$ -wk t  $\phi$  }
```

C_r is the compose kit between a renaming and another kit K . C_s is the compose kit between a substitution and another kit K , and requires that we already know how to compose a K -map with a renaming. The following verifies that C_r and C_s indeed get us all four compositions:

$$\begin{array}{llll} C_{rr} : \text{CKit } K_r K_r K_r & C_{rs} : \text{CKit } K_r K_s K_s & C_{sr} : \text{CKit } K_s K_r K_s & C_{ss} : \text{CKit } K_s K_s K_s \\ C_{rr} = C_r \{ \{ K = K_r \} \} & C_{rs} = C_r \{ \{ K = K_s \} \} & C_{sr} = C_s \{ \{ C = C_{rr} \} \} & C_{ss} = C_s \{ \{ C = C_{sr} \} \} \end{array}$$

4.3 Instantiation for System F

In this subsection, we show how to instantiate the $C\text{Traversal}$ abstraction for System F. In practice, this is done by our reflection algorithm automatically (Section 6), but it can be instructive to see, as it motivates the axioms of the $CKit$.

$$\begin{array}{l} \text{fusion} : \forall \{ K_1 : \text{Kit } _ \exists / _ \uparrow _ \} \{ K_2 : \text{Kit } _ \exists / _ \uparrow _ \} \{ K : \text{Kit } _ \exists / _ \uparrow _ \} \{ W_1 : \text{WkKit } K_1 \} \\ \quad \{ C : \text{CKit } K_1 K_2 K \} \{ t : S_1 \vdash s \} (\phi_1 : S_1 \dashv [K_1] \rightarrow S_2) (\phi_2 : S_2 \dashv [K_2] \rightarrow S_3) \rightarrow \\ \quad (t \cdots \phi_1) \cdots \phi_2 \equiv t \cdots (\phi_1 \cdot_k \phi_2) \\ \text{fusion } ('x) \phi_1 \phi_2 = \text{sym } (\&/\cdots\cdots (\phi_1 _ x) \phi_2) \\ \text{fusion } (\lambda x t) \phi_1 \phi_2 = \\ \quad \lambda x ((t \cdots (\phi_1 \uparrow)) \cdots (\phi_2 \uparrow)) \equiv (\text{cong } (\lambda t \rightarrow \lambda x t) (\text{fusion } t (\phi_1 \uparrow) (\phi_2 \uparrow))) \\ \quad \lambda x (t \cdots ((\phi_1 \uparrow) \cdot_k (\phi_2 \uparrow))) \equiv (\text{cong } (\lambda \phi \rightarrow \lambda x (t \cdots \phi)) (\text{sym } (\dashv\text{-ext } (\text{dist-}\uparrow \cdot \phi_1 \phi_2)))) \\ \quad \lambda x (t \cdots ((\phi_1 \cdot_k \phi_2) \uparrow)) \quad \blacksquare \end{array}$$

We only show the interesting cases, which are:

- variables, where we need to use the $\&/\cdots\cdots$ lemma provided by the $CKit$; and
- bindings, where the traversal operation $_ \cdots _$ needs to lift the map via $_ \uparrow _$, requiring us to distribute the lifting over the composition using $\text{dist-}\uparrow \cdot$.

5 Types & Typing

5.1 Types

In the context of multi-sorted syntax, the notion of a type can be described as a mapping between sorts. For System F, the expression sort maps to the type sort \approx , which in turn maps to the kind sort \top . The following structure is used to teach our framework about types:

```
record Types : Set1 where
  field ↑t : ∀ {st} → Sort st → ∃ [ st' ] Sort st'
```

For System F, the instantiation is

```
SystemF-Types : Types
SystemF-Types = record { ↑t = λ { → _, ≈ ; ≈ → _, ⊤ ; ⊤ → _, ⊤ } }
```

There are two things to discuss:

1. The \uparrow^t function maps a sort of arbitrary sort type, to a sort of a potentially different sort type, which is expressed by the use of an existential. For System F we require this generality, as the sort \approx can have variables, whereas its corresponding type sort \top cannot.
2. Some sorts, like \top , do not have corresponding type sorts, but we still need to provide one, as \uparrow^t is a total function. For such sorts, we can simply use arbitrary sort types, as the formalization will have no typing rules that use them.

To hide the existential, we define $S \vdash s$, which represents the type for a term $S \vdash s$.

```
⊢ : ⊢ _ : ∀ {t} → List (Sort Var) → Sort t → Set
S ⊢ s = S ⊢ proj2 (↑t s)
```

5.2 Type Contexts

Equipped with a notion of types, we are ready to define type contexts. As we want our framework to support dependent types, we allow a type in the context to use all variables bound previously in the context:

```
data Ctx : List (Sort Var) → Set where
  [] : Ctx []
  _::_ : S ⊢ s → Ctx S → Ctx (s :: S)
```

When looking up the type of a variable, we need to weaken it for each binding that comes after the variable³

```
lookup : Ctx S → S ∋ s → S ⊢ s
lookup (t :: Γ) zero = t ... weaken { Kr } _
lookup (t :: Γ) (suc x) = lookup Γ x ... weaken { Kr } _
```

Finally, a variable typing $\Gamma \ni x : t$ states that looking up x in Γ yields t :

```
_∋_ : Ctx S → S ∋ s → S ⊢ s → Set
Γ ∋ x : t = lookup Γ x ≡ t
```

5.3 Typing

Now that we have a notion of types and type contexts, we are ready to define the multi-sorted typing relation for System F, which describes both typing and kinding:

```
data _⊢_ : Ctx S → S ⊢ s → S ⊢ s → Set where
  ⊢' : ∀ {x : S ∋ s} {T : S ⊢ s} → Γ ∋ x : T → Γ ⊢' x : T
  ⊢λ : ∀ {e : (S ∋ s) ⊢} → (t1 :: Γ) ⊢ e : (wk t2) → Γ ⊢ λx e : t1 ⇒ t2
  ⊢Λ : (k :: Γ) ⊢ e : t2 → Γ ⊢ Λα e : ∀[α : k] t2
  ⊢· : Γ ⊢ e1 : t1 ⇒ t2 → Γ ⊢ e2 : t1 → Γ ⊢ e1 · e2 : t2
  ⊢• : ∀ {Γ : Ctx S} → (k2 :: Γ) ⊢ t1 : k1 → Γ ⊢ t2 : k2 → Γ ⊢ e1 : ∀[α : k2] t1 →
    Γ ⊢ e1 • t2 : t1 ... (t2)
  ⊢τ : Γ ⊢ t : *
```

The interesting cases are:

- the variable rule \vdash' , which covers both expression- and type-variables, analogously to the variable term constructor;
- the lambda rule $\vdash\lambda$, which weakens the codomain type t_2 . This is necessary, because multi-sorted syntax allows types to depend on expressions, so the typing derivation for e has to account for a variable, which is not used by the type; and
- the kinding rule $\vdash\tau$ states that all types have kind \star . This is sufficient for System F as types are automatically well-kinded due to intrinsic scoping.

To teach the framework about typing, we create a structure analogously to `Syntax`:

```
record Typing : Set1 where
  field _⊢_ : ∀ {s : Sort st} → Ctx S → S ⊢ s → S ⊢ s → Set
  ⊢' : ∀ {Γ : Ctx S} {x : S ∋ s} {t} → Γ ∋ x : t → Γ ⊢' x : t
```

The instantiation for System F is straightforward:

```
SystemF-Typing : Typing
SystemF-Typing = record { _⊢_ = _⊢_ ; ⊢' = ⊢' }
```

³ This includes the variable itself, which is not allowed to appear in its own type.

5.4 An Abstraction for Type Preservation

By now, the reader probably knows what comes next: we build an abstraction to unify type preservation for renamings and substitutions, eliminating the dependencies by yet another type of kits.

We start with **TKits**, which abstract over variable and term *typing*, and then define a **TTraversal**, which provides substitution-preserves-typing for all **TKits**.

```
record TKit (K : Kit _ $\exists$ /!_) : Set1 where
  field _ $\exists$ /!_ : Ctx S  $\rightarrow$  S  $\exists$ /! s  $\rightarrow$  S :! s  $\rightarrow$  Set
      id/!'   :  $\forall$  {t : S :! s} { $\Gamma$  : Ctx S}  $\rightarrow$   $\Gamma \ni x : t \rightarrow \Gamma \exists/! id/!' x : t$ 
      !'/id   :  $\forall$  {e : S  $\exists$ /! s} {t : S :! s} { $\Gamma$  : Ctx S}  $\rightarrow$   $\Gamma \exists/! e : t \rightarrow \Gamma !'/id e : t$ 
      !wk     :  $\forall$  ( $\Gamma$  : Ctx S) (t' : S :! s) (e : S  $\exists$ /! s') (t : S :! s')  $\rightarrow$ 
                 $\Gamma \exists/! e : t \rightarrow (t' :: \Gamma) \exists/! wk \_ e : (t \dots weaken \_)$ 
```

The first field abstracts over variable and term typing. The other fields express typings for the fields of a **Kit**. Building on the fields of a **TKit**, we define map typing and type preservation for map lifting and the singleton map in the record module of **TKit**:

- Using the variable/term typing, we can define a renaming/substitution typing:

```
_: $\Rightarrow$ _k_ : S1 -[ K ] $\rightarrow$  S2  $\rightarrow$  Ctx S1  $\rightarrow$  Ctx S2  $\rightarrow$  Set
_: $\Rightarrow$ _k_ {S1} {S2}  $\phi$   $\Gamma_1$   $\Gamma_2$  =  $\forall$  {s1} (x : S1  $\ni$  s1) (t : S1 :! s1)  $\rightarrow$ 
   $\Gamma_1 \ni x : t \rightarrow \Gamma_2 \exists/! (x \& \phi) : (t \dots \phi)$ 
```

$\phi : \Gamma_1 \Rightarrow_k \Gamma_2$ states that ϕ is a map that takes terms from Γ_1 to terms in Γ_2 .

- Lifting a map preserves its typing:

```
_! $\uparrow$ _ :  $\forall$  {W : WkKit K} {C1 : CKit K Kr K}
  { $\Gamma_1$  : Ctx S1} { $\Gamma_2$  : Ctx S2} { $\phi$  : S1 -[ K ] $\rightarrow$  S2}  $\rightarrow$ 
   $\phi : \Gamma_1 \Rightarrow_k \Gamma_2 \rightarrow (t : S1 :! s) \rightarrow (\phi \uparrow s) : (t :: \Gamma_1) \Rightarrow_k ((t \dots \phi) :: \Gamma_2)$ 
```

- If a variable/term has a typing, then so does its singleton renaming/substitution:

```
! $\uparrow$ (_) :  $\forall$  {s S} { $\Gamma$  : Ctx S} {x/t : S  $\ni$ /! s} {T : S :! s}  $\rightarrow$ 
   $\Gamma \exists/! x/t : T \rightarrow (! \uparrow x/t) : (T :: \Gamma) \Rightarrow_k \Gamma$ 
```

We then define a **TTraversal** analogously to **Traversal**, but instead of defining the application of maps, it defines that the application of a typed map to a typed term yields a typed term:

```
record TTraversal : Set1 where
  field _! $\uparrow$ ..._ :  $\forall$  {K : Kit _ $\exists$ /!_} {W : WkKit K} {TK : TKit K}
    {C1 : CKit K Kr K} {C2 : CKit K K K} {C3 : CKit K Ks Ks}
    {S1 S2 st} { $\Gamma_1$  : Ctx S1} { $\Gamma_2$  : Ctx S2} {s : Sort st}
    {e : S1 ! s} {t : S1 :! s} { $\phi$  : S1 -[ K ] $\rightarrow$  S2}  $\rightarrow$ 
       $\Gamma_1 \vdash e : t \rightarrow$ 
       $\phi : \Gamma_1 \Rightarrow_k \Gamma_2 \rightarrow$ 
       $\Gamma_2 \vdash (e \dots \phi) : (t \dots \phi)$ 
```

Given a term e with typing $\vdash e$ and a renaming/substitution ϕ with typing $\vdash \phi$, the term $\vdash e \dots \phi$ is a typing for $e \dots \phi$.

As before, we define the **TKit** instances in the record module of **TTraversal**:

```
TKr : TKit Kr ; TKs : TKit Ks
TKr = record { _ $\exists$ /!_ = _ $\exists$ !_ ; !'/id = !'
  ; id/!' =  $\lambda$  !x  $\rightarrow$  !x ; !wk =  $\lambda$  {  $\Gamma$  t' x t refl  $\rightarrow$  refl } }
TKs = record { _ $\exists$ /!_ = _! $\uparrow$ !_ ; !'/id =  $\lambda$  !x  $\rightarrow$  !x
  ; id/!' = !' ; !wk =  $\lambda$   $\Gamma$  t' e t !e  $\rightarrow$  !e !wk  $\Gamma$  t' }
```

32:14 Abstractions for Multi-Sorted Substitutions

The large amount of kit-parameters of $_ \vdash \dots _$ does not impose any restriction, as both our **Kits** support the **WkKit** extension and can be composed arbitrarily. Agda's instance resolution allows us to easily instantiate a concrete substitution-preserves-typing lemma:

```

$$\_ \vdash \dots \_ : \Gamma_1 \vdash e : t \rightarrow \sigma : \Gamma_1 \Rightarrow_s \Gamma_2 \rightarrow \Gamma_2 \vdash (e \dots \sigma) : (t \dots \sigma)$$


$$\_ \vdash \dots \_ = \_ \vdash \dots \_$$

```

5.5 Instantiation for System F

In this subsection, we show how to instantiate the **TTraversal** abstraction for System F. This is the only structure that is *not* instantiated automatically via reflection, as typing relations can be arbitrary complex in general.

```

$$\begin{array}{ll} \vdash' \vdash x & \vdash \dots \vdash \phi = \vdash' / \text{id} (\vdash \phi \_ \_ \vdash x) \\ \vdash \lambda \{t_2 = t_2\} \vdash e & \vdash \dots \vdash \phi = \vdash \lambda (\text{subst } (\lambda t \rightarrow \_ \vdash \_ : t) \\ & \quad (\text{sym } (\dots \uparrow \text{-wk } t_2 \_ \_)) \\ & \quad (\vdash e \vdash \dots (\vdash \phi \uparrow \_ \_))) \\ \vdash \Lambda \vdash e & \vdash \dots \vdash \phi = \vdash \Lambda (\vdash e \vdash \dots (\vdash \phi \uparrow \_ \_)) \\ \vdash \cdot \vdash e_1 \vdash e_2 & \vdash \dots \vdash \phi = \vdash \cdot (\vdash e_1 \vdash \dots \vdash \phi) (\vdash e_2 \vdash \dots \vdash \phi) \\ \vdash \bullet \{t_1 = t_1\} \{t_2 = t_2\} \vdash t_1 \vdash t_2 \vdash e_1 & \vdash \dots \vdash \phi = \text{subst } (\lambda t \rightarrow \_ \vdash \_ : t) \\ & \quad (\text{sym } (\text{dist-}\uparrow\text{-}\{\!\!\!\}\dots t_1 t_2 \_ \_)) \\ & \quad (\vdash \bullet (\vdash t_1 \vdash \dots (\vdash \phi \uparrow \_ \_)) \\ & \quad (\vdash t_2 \vdash \dots \vdash \phi) (\vdash e_1 \vdash \dots \vdash \phi)) \\ \vdash \tau & \vdash \dots \vdash \phi = \vdash \tau \end{array}$$

```

The type of $_ \vdash \dots _$ is the same as in the record definition and hence omitted. The interesting parts of the proof are:

- There is a strong similarity to the instantiation of map traversal $_ \dots _$: where $_ \dots _$ used $' / \text{id}$ or $_ \uparrow _$, our $_ \vdash \dots _$ uses their preservation lemmas \vdash' / id or $_ \uparrow _$.
- The lambda typing constructor $\vdash \lambda$ weakens the type t_2 to shield it from expression-substitution, requiring us to use $\dots \uparrow \text{-wk}$ to move the map under the weakening.
- The type application constructor $\vdash \bullet$ substitutes t_1 into t_2 , requiring us to use $\text{dist-}\uparrow\text{-}\{\!\!\!\}$ to move the map under the singleton substitution.

6 Reflection & Generics

We use Agda's reflection mechanism to derive instantiations related to all structures for untyped substitution, i.e. **Syntax**, **Traversal** and **CTraversal**. The user only needs to define syntax and typing, and can then move on to proving that substitution preserves typing, where all substitution lemmas are already available.

To gain insight into the class of object languages supported by our reflection algorithm, we have instantiated the structures for a generic syntax similar to the one in Allais et al.[2]. Our reflection algorithm derives proofs with the same structure as the generic proofs, giving high confidence that it covers the same class of languages.

Informally, all objects languages with multi-sorted syntax are supported that

1. have a variable constructor of type $\forall \{S s\} \rightarrow S \ni s \rightarrow S \vdash s$;
 2. use subterms only directly (e.g. not in lists); and
 3. only extend the scope-context of subterms, but never modify it otherwise.
- The formal definition of the generic syntax can be found in the supplementary material. Restriction 2 is purely technical and can be lifted with a more sophisticated reflection

algorithm. In the current system, this restriction can be worked around by inlining the data structure constructors into the syntax definition as terms of a new sort. For example, to allow lists of terms, we can add a polymorphic list sort

```
<list> : Sort st → Sort NoVar
```

and corresponding syntax constructors for lists

```
[] : S ⊢ <list> s
_::_ : S ⊢ s → S ⊢ <list> s → S ⊢ <list> s
```

This allows us to model, e.g., a multi-argument function call expression as

```
call : S ⊢ → S ⊢ <list> → S ⊢
```

Function types cannot be inlined, but require an extension of the reflection algorithm.

7 Case Studies

Using our full implementation of the framework, we proved subject reduction for the following object languages:

- For lambda calculus with dependent function types, the framework works out of the box for both deriving untyped substitution and instantiating the `TTraversal`. As types are terms, only a single sort is required. In the proof of confluence, the `CKit` abstraction allowed us to unify lemmas about the reduction of renamings and substitutions.
- For System F with subtyping, the main challenges are how to represent subtyping constraints and how to deal with the fact that substitution-preservation is not generally true, as type variables have subtyping bounds that need to be respected. While it would be possible to use our framework only to derive untyped substitution and define typing contexts and type preservation lemmas by hand, we instead used an encoding that allows us to use the `TKit` abstraction directly. Instead of binding a type variable with a subtyping constraint $\alpha <: t$, we first bind the type variable as $\alpha : *$, and then bind the constraint as $c : (\alpha <: t)$. This description is similar to first-class constraints, but restricted enough to be isomorphic to the original formalization, as constraint variables cannot be accessed by the user. With this encoding, substitution-preservation is generally true again: replacing a type variable $\alpha <: t$ with a type t' , which is not a subtype of t , results in a term that is still well-typed, but in a context with an unsatisfiable constraint $t' <: t$.
- Object languages with pattern matching can be modeled by adding the sorts of the variables bound by a pattern to the pattern sort `|` itself. A pattern matching clause $p \Rightarrow e$ can then be defined as

```
_=>_ : S ⊢ | S' → (S' ++ S) ⊢ → S ⊢
```

where S' describes the variables bound by the pattern and `|` is the sort of a clause.

8 Related Work

As the amount of related work is rather large, we focus on work that is closely related to ours and refer to other papers for the broader picture[21, 2].

8.1 Variable Binding

There is a plethora of different methods for representing variable binders: de Bruijn indices[10], co de Bruijn indices[16], locally nameless[8], locally named[17], higher order abstract syntax[18] and its parametric variant[9], nominal logic[23], shifted names[11], nameless painless[19], and scope graphs[24]. Many of these representations have been studied in solutions to the POPLMark challenges[4, 1].

8.2 Unifying Renamings & Substitution

The kit abstraction for unifying renamings and substitutions appeared first in an unpublished manuscript by McBride[15], and later in Benton et al.[6]. Wood and Atkey[25] propose an extension to kits that supports linear types via resource vectors. In all three cases kits are formulated for intrinsic typing and scenarios with polymorphism are not considered.

8.3 Extrinsically Typed Approaches

Autosubst[20] is a Coq framework, which derives parallel substitution definitions and lemmas for languages from annotated Coq syntax definitions using extrinsic typing, extrinsic scoping, and de Bruijn indices. The framework is implemented in Coq’s tactic language Ltac and comes with a decision procedure for all assumption-free, equational substitution-lemmas. The implementation of *Autosubst* deals with multiple variable sorts by generating multiple substitutions and corresponding interaction lemmas.

Autosubst 2[21] is a standalone code generator, which translates second-order HOAS specifications into mutual inductive term sorts. Compared to *Autosubst 1*, it features mutually recursive object languages, intrinsic scoping, and vectorized substitutions. Compared to our work, the syntax they generate takes the form of what we described as *unsorted syntax* in Section 2, i.e. different syntactic categories are described by different types with different amount of indices for variable counts. To eliminate the need for interaction lemmas, they define the notion of *vectorized substitution*, which combines the individual substitutions by putting them in a vector. We believe their great work of creating a decision procedure for vectorized substitutions should also translate to our setting with multi-sorted substitutions.

Needle and Knot[13] is a code generator for unscoped syntax with de Bruijn indices. They generate substitution and interaction lemmas for single-pointed substitution for languages with multiple variable sorts and binders that bind lists of variables.

All of the above work does not provide machinery to model typing and type preservation and does not unify renaming and substitution and their compositions. Hence, type preservation needs to be modeled manually and individually for renamings and substitutions.

8.4 Intrinsically Typed Approaches

Allais et al.[3] propose a powerful abstraction for denotational semantics and semantic fusion lemmas. In later work[2], they use generic programming to instantiate this abstraction for a class of object languages comparable to ours. They demonstrate how both renamings and substitutions can be described as semantics, how the four composition lemmas follow from their generic fusion lemma, and also provide an abstraction to unify renamings and substitutions. They show how to use their framework for both intrinsic and extrinsic typing, but are missing a story for polymorphism.

With only a slight modification to their framework, we can instantiate it for multi-sorted syntax, enabling the definition of polymorphic languages. However, as the intrinsic typing is then used to describe syntactic categories (and not the actual typing), the semantic

abstractions then refer to untyped terms, so typing and type preservation lemmas have to be modeled entirely manually. We believe it would be worthwhile to explore how their semantic abstractions can be lifted to typing relations similar to how our typing kits lift regular kits from terms to typing relations.

8.5 Pure Type Systems

Pure Type Systems[5, 7, 22] describe a class of typed lambda calculi parameterized over a set of sorts, dependencies between sorts, and rules for quantification. While pure type systems may seem very similar to multi-sorted syntax, they are actually quite different:

- In multi-sorted syntax, sorts describe syntactic categories of terms. Terms of different sorts are kept syntactically different, e.g. the set of expressions $S \vdash$ and types $S \vdash \approx$ in our System F example.
- In pure type systems, sorts are universe types, e.g. like `Set` in Agda or `Prop` in Coq. Terms which have different sorts as types, do still belong to the same syntactic category.

We can model pure type systems as a multi-sorted syntax with a single sort, where the sorts of the pure type system occur as terms representing universe types.

9 Conclusion

We have presented an Agda framework, which automatically derives definitions and lemmas for untyped substitution, and provides an abstraction for proving type preservation of renaming and substitution for all syntactic categories with a single lemma ($_ \vdash \dots _$).

Compared to many extrinsically typed approaches, our framework also models typing and type preservation. Compared to many intrinsically typed approaches, our framework gracefully extends to polymorphic scenarios.

The main limitation of our framework is the shape of typing relations, similarly as it is the case with approaches based on intrinsic typing: we can only model classical ternary typing relations. To adapt our framework to more complicated typing relations, the machinery for untyped substitution can be reused, but the abstractions related to typing need to be modified. We found that this works surprisingly well in practice, where we have already made extended typing abstractions that support linear typing ala Wood and Atkey[25] and general substructural typing ala Licata et al[14]. In both cases, the typing relation is extended with a fourth component that models usage restrictions on the type context.

While intrinsic typing allows to unify definitions with their type preservation proofs, extrinsic typing allows to unify substitutions across different syntactic categories, as we have demonstrated. We believe this makes our framework particularly suited for polymorphic languages, where the downside of extrinsic typing is automated away, and where we have variables across multiple syntactic categories, so the benefits of a unified substitution bear fruits.

References

- 1 Andreas Abel, Guillaume Allais, Aliya Hameer, Brigitte Pientka, Alberto Momigliano, Steven Schäfer, and Kathrin Stark. Poplmark reloaded: Mechanizing proofs by logical relations. *J. Funct. Program.*, 29:e19, 2019. doi:10.1017/S0956796819000170.
- 2 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. *Proc. ACM Program. Lang.*, 2(ICFP):90:1–90:30, 2018. doi:10.1145/3236785.

- 3 Guillaume Allais, James Chapman, Conor McBride, and James McKinna. Type-and-scope safe programs and their proofs. In Yves Bertot and Viktor Vafeiadis, editors, *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs, CPP 2017, Paris, France, January 16-17, 2017*, pages 195–207. ACM, 2017. doi:10.1145/3018610.3018613.
- 4 Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. Mechanized metatheory for the masses: The poplmark challenge. In Joe Hurd and Thomas F. Melham, editors, *Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005, Oxford, UK, August 22-25, 2005, Proceedings*, volume 3603 of *Lecture Notes in Computer Science*, pages 50–65. Springer, 2005. doi:10.1007/11541868_4.
- 5 Henk Barendregt and Kees Hemerik. Types in lambda calculi and programming languages. In Neil D. Jones, editor, *ESOP'90, 3rd European Symposium on Programming, Copenhagen, Denmark, May 15-18, 1990, Proceedings*, volume 432 of *Lecture Notes in Computer Science*, pages 1–35. Springer, 1990. doi:10.1007/3-540-52592-0_53.
- 6 Nick Benton, Chung-Kil Hur, Andrew Kennedy, and Conor McBride. Strongly typed term representations in coq. *J. Autom. Reason.*, 49(2):141–159, 2012. doi:10.1007/s10817-011-9219-0.
- 7 S Berardi. Towards a mathematical analysis of the coquand-huet calculus of constructions and the other systems in barendregt’s cube. dept. *Computer Science, Carnegie-Mellon University and Dipartimento Matematica, Universita di Torino, Italy*, 1988.
- 8 Arthur Charguéraud. The locally nameless representation. *J. Autom. Reason.*, 49(3):363–408, 2012. doi:10.1007/S10817-011-9225-2.
- 9 Adam Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In James Hook and Peter Thiemann, editors, *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*, pages 143–156. ACM, 2008. doi:10.1145/1411204.1411226.
- 10 Nicolaas Govert De Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. In *Indagationes Mathematicae (Proceedings)*, volume 75, pages 381–392. Elsevier, 1972.
- 11 Stephen Dolan and Leo White. Syntax with shifted names. In *TyDe Workshop*, 2019. URL: <http://tydeworkshop.org/2019-abstracts/paper16.pdf>.
- 12 Mark P. Jones. Type classes with functional dependencies. In Gert Smolka, editor, *Programming Languages and Systems, 9th European Symposium on Programming, ESOP 2000, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS 2000, Berlin, Germany, March 25 - April 2, 2000, Proceedings*, volume 1782 of *Lecture Notes in Computer Science*, pages 230–244. Springer, 2000. doi:10.1007/3-540-46425-5_15.
- 13 Steven Keuchel, Stephanie Weirich, and Tom Schrijvers. Needle & knot: Binder boilerplate tied up. 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 419–445. Springer, 2016. doi:10.1007/978-3-662-49498-1_17.
- 14 Daniel R. Licata, Michael Shulman, and Mitchell Riley. A fibrational framework for substructural and modal logics. In Dale Miller, editor, *2nd International Conference on Formal Structures for Computation and Deduction, FSCD 2017, September 3-9, 2017, Oxford, UK*, volume 84 of *LIPICs*, pages 25:1–25:22. Schloss Dagstuhl -- Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPICs.FSCD.2017.25.
- 15 Conor McBride. Type-preserving renaming and substitution. Unpublished manuscript, 2005. URL: <http://strictlypositive.org/ren-sub.pdf>.
- 16 Conor McBride. Everybody’s got to be somewhere. In Robert Atkey and Sam Lindley, editors, *Proceedings of the 7th Workshop on Mathematically Structured Functional Programming*,

- MSFP@FSCD 2018, Oxford, UK, 8th July 2018*, volume 275 of *EPTCS*, pages 53–69, 2018. doi:10.4204/EPTCS.275.6.
- 17 James McKinna and Robert Pollack. Pure type systems formalized. In Marc Bezem and Jan Friso Groote, editors, *Typed Lambda Calculi and Applications, International Conference on Typed Lambda Calculi and Applications, TLCA '93, Utrecht, The Netherlands, March 16-18, 1993, Proceedings*, volume 664 of *Lecture Notes in Computer Science*, pages 289–305. Springer, 1993. doi:10.1007/BFB0037113.
 - 18 Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In Richard L. Wexelblat, editor, *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, USA, June 22-24, 1988*, pages 199–208. ACM, 1988. doi:10.1145/53990.54010.
 - 19 Nicolas Pouillard. Nameless, painless. 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 320–332. ACM, 2011. doi:10.1145/2034773.2034817.
 - 20 Steven Schäfer, Tobias Tebbi, and Gert Smolka. Autosubst: Reasoning with de bruijn terms and parallel substitutions. In Christian Urban and Xingyuan Zhang, editors, *Interactive Theorem Proving - 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015, Proceedings*, volume 9236 of *Lecture Notes in Computer Science*, pages 359–374. Springer, 2015. doi:10.1007/978-3-319-22102-1_24.
 - 21 Kathrin Stark, Steven Schäfer, and Jonas Kaiser. Autosubst 2: reasoning with multi-sorted de bruijn terms and vector substitutions. In Assia Mahboubi and Magnus O. Myreen, editors, *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, Cascais, Portugal, January 14-15, 2019*, pages 166–180. ACM, 2019. doi:10.1145/3293880.3294101.
 - 22 Jan Terlouw. Een nadere bewijstheoretische analyse van gstt's. *Manuscript (in Dutch)*, 1989.
 - 23 Christian Urban, Andrew M. Pitts, and Murdoch Gabbay. Nominal unification. *Theor. Comput. Sci.*, 323(1-3):473–497, 2004. doi:10.1016/J.TCS.2004.06.016.
 - 24 Hendrik van Antwerpen, Casper Bach Poulsen, Arjen Rouvoet, and Eelco Visser. Scopes as types. *Proc. ACM Program. Lang.*, 2(OOPSLA):114:1–114:30, 2018. doi:10.1145/3276484.
 - 25 James Wood and Robert Atkey. A linear algebra approach to linear metatheory. In Ugo Dal Lago and Valeria de Paiva, editors, *Proceedings Second Joint International Workshop on Linearity & Trends in Linear Logic and Applications, Linearity&TLLA@IJCAR-FSCD 2020, Online, 29-30 June 2020*, volume 353 of *EPTCS*, pages 195–212, 2020. doi:10.4204/EPTCS.353.10.

Correctly Compiling Proofs About Programs Without Proving Compilers Correct

Audrey Seo*  

University of Washington, Seattle, WA, USA

Christopher Lam*  

University of Illinois Urbana-Champaign, IL, USA

Dan Grossman  

University of Washington, Seattle, WA, USA

Talia Ringer  

University of Illinois Urbana-Champaign, IL, USA

Abstract

Guaranteeing correct compilation is nearly synonymous with compiler verification. However, the correctness guarantees for certified compilers and translation validation can be stronger than we need. While many compilers do have incorrect behavior, even when a compiler bug occurs it may not change the program’s behavior meaningfully with respect to its specification. Many real-world specifications are necessarily partial in that they do not completely specify all of a program’s behavior. While compiler verification and formal methods have had great success for safety-critical systems, there are magnitudes more code, such as math libraries, compiled with incorrect compilers, that would benefit from a guarantee of its partial specification.

This paper explores a technique to get guarantees about compiled programs even in the presence of an unverified, or even incorrect, compiler. Our workflow compiles programs, specifications, and proof objects, from an embedded source language and logic to an embedded target language and logic. We implement two simple imperative languages, each with its own Hoare-style program logic, and a system for instantiating proof compilers out of compilers between these two languages that fulfill certain equational conditions in Coq. We instantiate our system on four compilers: one that is incomplete, two that are incorrect, and one that is correct but unverified. We use these instances to compile Hoare proofs for several programs, and we are able to leverage compiled proofs to assist in proofs of larger programs. Our proof compiler system is formally proven sound in Coq. We demonstrate how our approach enables strong target program guarantees even in the presence of incorrect compilation, opening up new options for which proof burdens one might shoulder instead of, or in addition to, compiler correctness.

2012 ACM Subject Classification Theory of computation → Logic and verification; Theory of computation → Hoare logic; Software and its engineering → Compilers

Keywords and phrases proof transformations, compiler validation, program logics, proof engineering

Digital Object Identifier 10.4230/LIPIcs.ITP.2024.33

Supplementary Material *Software (Source Code)*: <https://github.com/uwplse/potpie/tree/v0.5> [44], archived at [swh:1:dir:5b78a12be7ef95b92fe5db2acf903d436e951851](https://swh.io/1:dir:5b78a12be7ef95b92fe5db2acf903d436e951851)

Funding This research was developed with funding from the Defense Advanced Research Projects Agency. The views, opinions, and/or findings expressed are those of the author(s) and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

Acknowledgements We thank the Coq team for their proof engineering advice. We thank Guilherme

* Co-first authors



© Audrey Seo, Christopher Lam, Dan Grossman, and Talia Ringer; licensed under Creative Commons License CC-BY 4.0

15th International Conference on Interactive Theorem Proving (ITP 2024).

Editors: Yves Bertot, Temur Kutsia, and Michael Norrish; Article No. 33; pp. 33:1–33:20

Leibniz International Proceedings in Informatics



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

Espada, John Leo, Pedro Amorim, Sophia Roshal, and Zachary Tatlock for their paper feedback.

1 Introduction

Program logic systems help proof engineers do more advanced reasoning about program-specific properties. Iris [18, 24], VST [8], CHL [11], and SEPREF [27] are just a few examples of such program logics. Traditionally, strong guarantees for compiled programs required composing program logics with verified compilers [8]. However, because functional specifications are often *partial*, preserving them through compilation sometimes does not require a correct compiler pass, much less global compiler correctness.

To see an example of where correct compilation becomes too strict, consider a Hoare triple $\{0 \leq a \wedge 0 \leq \epsilon\} y := 42; x := \text{source_sqrt}(a) \{|a - x^2| \leq \epsilon\}$, which says that after setting y to 42 and calling `source_sqrt` on a , the variable x stores a square root approximation of a within ϵ . Suppose that `source_sqrt` is compiled to some program `target_sqrt` such that if $0 \leq a \wedge 0 \leq \epsilon$, then after `target_sqrt(a)` runs, we have $|a - x^2| \leq \frac{\epsilon}{2}$. In the end, we still have $|a - x^2| \leq \epsilon$ for `target_sqrt` since $\frac{\epsilon}{2} \leq \epsilon$, which meets the specification. Moreover, the 42 on the right-hand side of the assignment to y could be (mis)compiled to anything, and the specification would still be preserved. However, this compilation would be rejected by both certified compilation and translation validation, illustrating that *compiler correctness* is significantly more restrictive than *specification preservation*.

In order to achieve guaranteed specification-preserving compiler passes, we present the *proof compiler system* POTPIE. POTPIE takes an existing compiler and produces a proof compiler. A proof compiler takes a program, a specification, and a proof of the specification and compiles all three such that (1) the specification’s meaning is preserved, and (2) the compiled proof shows that the compiled program meets the compiled specification.

POTPIE is formally verified in Coq [45], and allows for partial specification-preserving compilation, even of *incorrectly compiled* programs. To get a sense of how POTPIE differs from similar techniques, imagine a proof engineer has already shown the Hoare triple $\{0 \leq a \wedge 0 \leq \epsilon\} x := \text{source_sqrt}(a) \{|x^2 - a| \leq \epsilon\}$ and wants to prove an analogous Hoare triple about the compiled square root approximation. Suppose also that the proof engineer has a compiler T on hand, which happens to have a small bug that switches $<$ to \leq in programs and specifications. The square root program uses a while loop to approximate square roots, and the while loop condition contains at least one $<$. At this point, POTPIE provides two options:

1. TREE workflow: use T to instantiate a *proof tree compiler* that produces a target proof tree. After compiling the square root Hoare tree, they invoke the TREE Coq plugin which will check the proof tree, and if possible, produce a certificate that is checkable in Coq. TREE has only one proof obligation to invoke the plugin, but may fail in certain cases.
2. CC workflow: use T to instantiate a *correct-by-construction proof compiler* by showing that it satisfies the equations in Figure 5 on Page 8. To call this proof compiler, the proof engineer must show that the square root program is well-formed. CC is complete in that if the translation preserves the specification, then it is possible to perform.

Both methods work, even though the compiler T has a bug that causes *miscompilation* in the square root program. Because of this miscompilation, we cannot use translation validation, the state of the art for ensuring correct compilation for an unverified compiler. But the miscompilation does not affect our specification, so with POTPIE, we can get strong guarantees about our compiled code regardless of miscompilation.

We make the following contributions:

$a ::= \mathbb{N} \mid x \mid \text{param } k \mid a + a \mid a - a \mid f(a, \dots, a)$ $b ::= T \mid F \mid \neg b \mid a \leq a \mid b \wedge b \mid b \vee b$ $i ::= \text{skip} \mid x := a \mid i; i$ $\quad \mid \text{if } b \text{ then } i \text{ else } i \mid \text{while } b \text{ do } i$ $\lambda ::= (f, k, i, \text{return } x)$ $p ::= (\{\lambda, \dots, \lambda\}, i)$	$a ::= \mathbb{N} \mid \#k \mid a + a \mid a - a \mid f(a, \dots, a)$ $b ::= T \mid F \mid \neg b \mid a \leq a \mid b \wedge b \mid b \vee b$ $i ::= \text{skip} \mid \text{push} \mid \text{pop} \mid \#k := a \mid i; i$ $\quad \mid \text{if } b \text{ then } i \text{ else } i \mid \text{while } b \text{ do } i$ $\lambda ::= (f, k, i, \text{return } a \ n)$ $p ::= (\{\lambda, \dots, \lambda\}, i)$
---	---

■ **Figure 1** IMP (left) and STACK (right) syntax, where a describes arithmetic expressions, b boolean expressions, i imperative statements, λ function definitions, and p whole programs, which consist of a set of functions and a “main” body. The evaluation of the main body yields the result of program. For IMP functions, $(f, k, i, \text{return } x)$ is a function named f with k parameters that returns the value of the variable x after executing the function body, i . For STACK functions $(f, k, i, \text{return } a \ n)$, we return the result of evaluating a after executing the body i , and then pop n indices from the stack.

1. We present the POTPIE system for specification-preserving proof compilation.
2. We describe two workflows for the POTPIE system: CC and TREE.
3. We demonstrate POTPIE on several case studies, using code compilers with varying degrees of incorrectness to correctly compile proofs. Our case studies include various mathematical functions, such as infinite series and square root approximation.
4. We prove the CC and TREE workflows sound in Coq.

Non-Goals and Limitations. Our work aims to complement, not replace, certified compilation. One potential motivation for alternative compiler correctness techniques is to ease the burden of compiler verification. However, easing the burden of compiler verification is not our goal, nor do we think that this is the case for our work at this time. Rather, our goal is demonstrate a complementary approach of specification-preserving compilation for program-specific specifications, even when the program itself is incorrectly compiled. Our work currently focuses on simple and closely related languages, and the compilers are likewise simple, though we do not believe that these choices are central to our approach. Currently, our work imposes significant limitations the kinds of control flow optimizations that can be performed. This simplifying decision made the problem initially tractable, but we do not believe it is inherent to our approach; we discuss a potential way of handling it in Section 7.

2 Programs, Specifications, and Proofs

In this section, we briefly present our six languages and how to compile programs and specifications, with Section 2.1 describing the programming languages and program compiler, Section 2.2 describing the specification languages and compiler, and Section 2.3 describing the proof languages (the proof compiler system is described in Section 3). Here and throughout the paper, we include links such as [\(42\)](#) to relevant locations in our code, which you can find on GitHub: <https://github.com/uwplse/potpie/tree/v0.4>.

2.1 Programs

Our languages IMP and STACK are both simple imperative languages that are similar in syntax (Figure 1) yet have differing memory models. IMP has an abstract environment with two components: a mapping of identifiers to their `nat` values, and function parameters (accessed via the `param k` construct), whereas STACK has a single function call stack, where new variables are pushed to the low indices and stack indices are accessed with the `#k`

$$\begin{array}{ll}
\text{comp}_a^\varphi(n) \triangleq n & \text{comp}_a^\varphi(x) \triangleq \#\varphi(x) & \text{comp}_b^\varphi(T) \triangleq T & \text{comp}_b^\varphi(F) \triangleq F \\
\text{comp}_a^\varphi(\text{param } k) \triangleq \#(|V| + k + 1) & & \text{comp}_b^\varphi(\neg b) \triangleq \neg \text{comp}_b^\varphi(b) & \\
\text{comp}_a^\varphi(a_1 \text{ op } a_2) \triangleq \text{comp}_a^\varphi(a_1) \text{ op } \text{comp}_a^\varphi(a_2) & & \text{comp}_b^\varphi(b_1 \text{ op } b_2) \triangleq \text{comp}_b^\varphi(b_1) \text{ op } \text{comp}_b^\varphi(b_2) & \\
\text{comp}_a^\varphi(f(a_1, \dots, a_n)) \triangleq f(\text{comp}_a^\varphi(a_1), \dots, \text{comp}_a^\varphi(a_n)) & & \text{comp}_b^\varphi(a_1 \leq a_2) \triangleq \text{comp}_a^\varphi(a_1) \leq \text{comp}_a^\varphi(a_2) &
\end{array}$$

■ **Figure 2** An arithmetic expression compiler comp_a (left) and a boolean expression compiler comp_b (right). op stands for the appropriate binary operators: $+$ and $-$, and \wedge and \vee , respectively.

$$\begin{array}{l}
M ::= T \mid F \mid p_n [e, \dots, e] \\
\quad \mid M \wedge M \mid M \vee M
\end{array}
\quad
\frac{}{\sigma \models T} \text{TRUE}
\quad
\frac{\text{map_eval}_\sigma [a_i]_1^n [v_i]_1^n \quad p_n \text{ vlist}}{\sigma \models p_n [a_1, \dots, a_n]} \text{N-ARY}$$

■ **Figure 3** Syntax (left) and semantics (right) for base assertions for both IMP and STACK. map_eval_σ is a relation from lists of expressions to lists of values. The semantic interpretation is parametric over the types of v , σ , and map_eval_σ . Interpretations for \wedge and \vee are standard.

construct. Function calls in IMP are always mutation-free since functions are limited to their (immutable) parameters and local scope. STACK’s functions can access the entire stack.

Bridging the Abstraction Gap. The difference in memory model must be taken into account when compiling from IMP to STACK. We define an equivalence between variable environments and stacks ④ so that “sound translation” is a well-defined concept.

► **Definition 1.** *Let V be a finite set of variable names, and let $\varphi : V \rightarrow \{1, \dots, |V|\}$ be bijective with inverse φ^{-1} . Then for all variable stores σ , parameter stores Δ , and stacks σ_s , we say that σ and Δ are φ -equivalent to σ_s , written $(\sigma, \Delta) \approx_\varphi \sigma_s$, if (1) for $1 \leq i \leq |V|$, we have $\sigma_s[i] = \sigma(\varphi^{-1}(i))$, and (2) for $|V| + 1 \leq i \leq |V| + |\Delta|$, we have $\sigma_s[i] = \Delta[i - |V|]$.*

This equivalence is entirely dependent on our choice of mapping between variables and stack slots. It has this form since parameters are always at the top of the stack at the beginning of a function call, and are then pushed down as space for local variables is allocated, so parameters appear “after” (i.e., appended to) the local variables. Note that this implies $|V| + |\Delta| \leq |\sigma_s|$ while saying nothing about stack indices beyond $|V| + |\Delta|$.

Compiling Programs. Although the POTPIE system allows for some choice of compiler between IMP and STACK, most of our compilers follow a common structure. We give a translation for IMP arithmetic and boolean expressions (which we will refer to in sum as *expressions* from now on) in Figure 2. This infrastructure is a straightforward extension of the variable mapping function φ from Definition 1. The program compilers we deal with in our case studies (Section 4) define variations on this common structure.

2.2 Specifications

The specification languages both embed IMP or STACK expressions inside of them, respectively. Base assertions are modeled as n-ary predicates over the arithmetic and boolean expressions of the given language. The semantics for assigning a truth value to a formula (Figure 3, right) parameterize predicates over the value types. For example, if we have the assertion $p_1 a$ where a is an IMP expression that evaluates to v , then $p_1 a$ is true if and only if calling

$$\begin{aligned} \text{comp}_{\text{spec}}^{\varphi,k}(T) &\triangleq (k, T) & \text{comp}_{\text{spec}}^{\varphi,k}(p_n(e_1, \dots, e_n)) &\triangleq (k, p_n(\text{comp}_{\text{expr}}^{\varphi}(e_1), \dots, \text{comp}_{\text{expr}}^{\varphi}(e_n))) \\ \text{comp}_{\text{spec}}^{\varphi,k}(F) &\triangleq (k, F) & \text{comp}_{\text{spec}}^{\varphi,k}(S_1 \text{ op } S_2) &\triangleq \text{comp}_{\text{spec}}^{\varphi,k}(S_1) \text{ op } \text{comp}_{\text{spec}}^{\varphi,k}(S_2) \end{aligned}$$

■ **Figure 4** The specification compiler $\text{comp}_{\text{spec}}^{\varphi,k}(S)$, which is parameterized over $\text{comp}_{\text{expr}}^{\varphi}$ (which can be either $\text{comp}_{\text{a}}^{\varphi}$ or $\text{comp}_{\text{b}}^{\varphi}$, depending on the type of expressions e). op is either \wedge or \vee .

the Coq definition of p_1 with v is a true **PROP**. We can define a program logic S for the source language this way by using the atoms in Figure 3 to embed arithmetic and boolean expressions in Coq propositions. We add conjunction and disjunction connectives at the logic level. We can define T for the target language similarly. We then use this to construct the following specification grammars:

$$S ::= S_e \mid S_1 \wedge S_2 \mid S_1 \vee S_2 \qquad T ::= (n, T_e) \mid T_1 \wedge T_2 \mid T_1 \vee T_2 \quad (1)$$

where S_e and T_e are instances of the logic described in Figure 3 using **IMP** and **STACK** arithmetic and boolean expressions respectively.

Because the minimum stack size required by the compilation might not be captured by language expressions contained within the formula itself, we also want to specify a minimum stack size in **STACK** specifications. This is represented by the following judgment:

$$\frac{|\sigma| \geq n \quad \sigma \models T_e}{\sigma \models (n, T_e)} \text{ STACK BASE}$$

We made the decision to allow function calls within specifications. This is not essential to our approach – one could disallow effectful constructs from expressions as in **CLight** [6]. For the current system, we find it more natural to reason about effectful expressions in **IMP**.

Compiling Specifications. We can reuse $\varphi : V \rightarrow \{1, \dots, |V|\}$ and the expression compilers from Section 2.1 to define a specification compiler (see Figure 4): recurse over the source logic formula and compile the leaves, i.e., **IMP** expressions. If k is the number of function arguments, give each assertion a minimal stack size, $|V| + k$, to ensure well-formedness of the resulting **STACK** expressions within the specification, which is given as the maximum value of φ plus k , where k is the number of arguments. Note that this definition is parameterized over an expression compiler, which need not be fully correct. To guarantee correctness of a translated proof in the sense that the target proof “proves the same thing”, users must show that the specification compiler must be sound with respect to the user’s source specification (see Definition 3 and Section 3.2.2). This ensures that the compiled proof proves an analogous property even when the program is compiled incorrectly.

2.3 Proofs

Our logics are based on standard Hoare logic and are proven sound in Coq. Automatically ensuring that the rule of consequence’s implications are preserved by compilation would usually require correctness of compilation. To remove this requirement, we modify the rule of consequence so that implications must be in an *implication database* I , which is a list of pairs of specifications that satisfy the following definition:

► **Definition 2.** I is valid if for each pair (P, Q) in I , $\forall \sigma, \sigma \models P \Rightarrow \sigma \models Q$.

■ **Table 1** Proof obligations and their relationship to the requirements for instantiating and invoking proof compilers (PC) for each of our workflows, and what properties may be guaranteed for TREE by these proof obligations. P means a user proof is required, A means that the plugin will attempt an automated check, × means the condition is not required, and - means the condition is not applicable to that column. “Trees WF” means the compiled code and assertions within the STACK Hoare tree have the right syntactic shape for Hoare rule application. “Valid Tree” means that the tree is a valid STACK Hoare proof (which is implied by a typechecked certificate). “CGC” indicates what is needed to ensure that once a certificate is generated and typechecks, that it is correct, i.e., preserves the meaning of the pre and postcondition. Since CC is correct-by-construction, all of the proof obligations are required.

		TREE						CC	
		Create PC	Invoke		Guaranteeing Properties			Create PC	Invoke PC
			PC	Plugin	Trees WF	Valid Tree	CGC		
Comp.	Comm.	×	-	-	A	A	-	P	-
User	Spec DB	-	×	P	×	P	-	-	P
	Pre/Post	-	×	×	×	×	P	-	P
	IMP WF	-	×	×	-	-	-	-	P
	preservesStack	-	×	×	A	A	-	-	P

This implication database, which is present for both IMP and STACK, serves to (1) identify which implications must be preserved through compilation, and (2) make it easy to identify which source implication corresponds to which target implication across compilation. For the STACK logic, as a simplifying assumption, we further require all expressions in assignments, if conditions, or while conditions to be side effect-free, i.e., preserve the stack.

3 Compiling Proofs

POTPIE’s two workflows share the same goal: to produce a term at the target representing a proof tree for the desired Stack-level property. To achieve this, both workflows have their own soundness theorems (Section 3.1), which need certain properties to be true of compiled programs and specifications. The workflows obtain these in different ways. Before being called, CC requires the user to prove certain equational properties about the compiler (Section 3.2.1) and well-formedness properties of the source program and proof (Section 3.2.3), and combines these to acquire the required syntactic and stack-preserving conditions for applying STACK Hoare rules. TREE simply compiles the Hoare proof tree, and its plugin performs an automated check (that can possibly fail) of whether the compiled tree is a valid Hoare proof. Additionally, both workflows require the user to manually translate the implication databases (Section 3.2.2) to retrieve STACK-level rule-of-consequence applications. A breakdown of which proof obligations are required for which workflow and the guarantees they provide can be found in Table 1. None of these proof obligations require full semantic preservation; they allow for some miscompilation of programs as long as compilation does not break the (possibly partial) specification.

3.1 Soundness Theorems and Overview

Consider the IMP Hoare triple $\{5 < 10\}x := 5\{x < 10\}$, which can be derived via a simple application of the IMP-level assignment rule. If we map x to stack slot #1, the “natural” translation of this IMP triple is the STACK triple $\{5 < 10\}\#1 := 5\{\#1 < 10\}$, which can be derived via STACK’s assignment Hoare rule. This translation seems “natural” for two

reasons: it is derived using the “same” rules, and it is proving the “same” thing. We use the former to compile the proofs, and we use the latter to define a notion of *soundness* for specification translation (30) (31), which each workflow can guarantee in a different way:

► **Definition 3.** For a given P , a specification compilation function $\text{comp}_{\text{spec}}^{\varphi,k}$ is sound with respect to P if for all σ, Δ, σ_s such that $(\sigma, \Delta) \approx_{\varphi} \sigma_s$, we have $\sigma, \Delta \models P \Leftrightarrow \sigma_s \models \text{comp}_{\text{spec}}^{\varphi,k}(P)$.

We can also define an informal notion of soundness for a proof compiler:

► **Definition 4.** Given an IMP Hoare proof pf that proves the triple $\{P\}c\{Q\}$, a proof compiler PC is sound with regards to it if $PC(pf) = pf'$ and pf' proves the triple $\{\text{comp}_{\text{spec}}(P)\}\text{comp}_{\text{code}}(c)\{\text{comp}_{\text{spec}}(Q)\}$.

Combining both notions of soundness lets us arrive at our definition of *soundness for a proof compiler*: if a specification and proof compiler are sound with regards to a specification and proof in the sense of Definitions 3 and 4, then the compiled version of that proof is both a valid proof at the target and proves the same thing that the source proof proved. The TREE workflow can achieve these guarantees in piecewise progression when certain proof obligations are met, and CC always guarantees both when it is called. The form Definition 4 takes in our implementation is a method of constructing a term of type `hl_stk` (the STACK correct-by-construction Hoare proof type) from a term of type `hl_Imp_Lang`.

Tree Proof Compiler. The TREE workflow utilizes a proof compiler that separates proof and compilation, and has two components: a compiler that produces a proof tree (2) and a Coq plugin, implemented in OCaml (5), that checks the proof tree’s validity (6). The compiler is parameterized over the code and specification compilers from IMP to STACK. The proof tree compiler component is sound in the sense that if the proof obligations for the CC proof compiler are satisfied, then it will always produce a sound tree (12). The plugin can be used on any STACK proof tree and can optionally produce a certificate, which can be used to produce a STACK Hoare logic proof via this theorem (13):

```

1 Theorem valid_tree_can_construct_hl_stk
2   (P Q: AbsState) (i: imp_stack) (facts': implication_env_stk)
3   (fenv': fun_env_stk) (T: stk_hoare_tree):
4     ∇ (V: stk_valid_tree P i Q facts' fenv' T), (* certificate type*)
5     hl_stk P i Q facts' fenv'.

```

An instance of Definition 4 can be retrieved by an appropriate substitution of variables.

We note that TREE is not *complete*: the requisite target-level properties could be true, and yet TREE will still fail. This can occur in the case of mutually recursive functions, along with some edge cases that we talk more about in Section 5.1. While TREE requires fewer proof obligations, it also provides fewer guarantees. One such guarantee it lacks is preservation of the pre and postcondition, i.e., specification-preserving compilation. This and other guarantees can be gained by showing the proof obligations indicated in Table 1.

CC Proof Compiler. This workflow is correct by construction. Given an IMP Hoare proof (`hl_Imp_Lang`) along with the CC proof obligations (described in Section 3.2), CC produces a STACK Hoare proof (`hl_stk`) of the same property (1) (some detail is omitted for brevity):

```

1 Definition proof_compiler :
2   ∇ (P Q: AbsEnv) (i: imp_Imp_Lang) (fenv: fun_env) (facts: implication_env)
3   (var_to_stack_map: list string) (num_args: nat)

```

$$\text{comp}_{\text{spec}}^{\varphi,k}(P[x \rightarrow a]) = (\text{comp}_{\text{spec}}^{\varphi,k}(P))[\varphi(x) \rightarrow \text{comp}_a^{\varphi}(a)] \quad (2)$$

$$\text{comp}_{\text{spec}}^{\varphi,k}((p_1 [b]) \wedge P) = (k + |V|, (p_1 [\text{comp}_b^{\varphi}(b)]) \wedge \text{comp}_{\text{spec}}^{\varphi,k}(P)) \quad (3)$$

$$\text{comp}_{\text{code}}^{\varphi,k}(x := a) = \#\varphi(x) := \text{comp}_a^{\varphi}(a) \quad (4)$$

$$\text{comp}_{\text{code}}^{\varphi,k}(\text{skip}) = \text{skip} \quad (5)$$

$$\text{comp}_{\text{code}}^{\varphi,k}(i_1; i_2) = \text{comp}_{\text{code}}^{\varphi,k}(i_1); \text{comp}_{\text{code}}^{\varphi,k}(i_2) \quad (6)$$

$$\text{comp}_{\text{code}}^{\varphi,k}(\text{if } b \text{ then } i_1 \text{ else } i_2) = \text{if } \text{comp}_b^{\varphi}(b) \text{ then } \text{comp}_{\text{code}}^{\varphi,k}(i_1) \text{ else } \text{comp}_{\text{code}}^{\varphi,k}(i_2) \quad (7)$$

$$\text{comp}_{\text{code}}^{\varphi,k}(\text{while } b \text{ do } i) = \text{while } \text{comp}_b^{\varphi}(b) \text{ do } \text{comp}_{\text{code}}^{\varphi,k}(i) \quad (8)$$

■ **Figure 5** Equations compilers must satisfy to be used to instantiate a proof compiler.

```

4 (proof: hl_Imp_Lang P i Q facts fenv) (translate_facts: valid_imp_trans_def),
5 (* well-formedness conditions and specification translation soundness *) →
6 hl_stk (comp P) (comp i) (comp Q) (comp facts) (comp fenv).

```

Since the CC proof compiler is correct-by-construction, the type signature in the above Coq code guarantees the validity of the produced target Hoare proof. However, as compared to TREE, CC requires far more proof obligations before a CC proof compiler can even be instantiated, with invocation requiring several on top of the instantiation burden.

3.2 Proof Obligations

POTPIE’s workflows both require some proof obligations in order to get target-level correctness guarantees. Table 1 breaks down these requirements for both workflows.

3.2.1 Commutativity Equations – CC Only

These code and specification compiler proof obligations relate the compiled programs and specifications. CC requires that proof-compileable IMP programs and specifications satisfy the equations in Figure 5 – TREE has no such requirement (Table 1) and will simply fail if these equations don’t hold. For example, consider the substitution performed by the assignment rule. Given some P , in order to compile an application of the assignment rule, we want (2) to hold. If we have this equality, we have the following, where $P' = \text{comp}_{\text{spec}}^{\varphi,k}(P)$:

$$\text{comp}_{\text{pf}}^{\varphi,k}(\{P[x \rightarrow a]\} x := a \{P\}) = \left\{ P'[\varphi(x) \rightarrow \text{comp}_{\text{code}}^{\varphi,k}(a)] \right\} \varphi(x) := a \left\{ P' \right\}$$

This compiler proof obligation lets a CC proof compiler mechanically apply the Hoare rules. In practice, as long as the program compilers are executable, these conditions are provable using **reflexivity**. These equations are the reason for the control-flow restrictions mentioned in the introduction and in Section 7. These equations also ensure that the specification compiler is “aware” of the way that expressions are compiled. For example, consider a code compiler that adds 1 to assignment statements’ right hand sides. This breaks the compilation of the assignment rule, as the specification compiler is “unaware” of a transformation that affects a Hoare rule application. Equations 2-4 and 7-8 in Figure 5 are to prevent such cases.

3.2.2 Specification Translation Conditions – Tree & CC

As we described in Section 2.3, the rule of consequence is the only Hoare rule that depends on the semantics of the program, and thus would require a completely correct compiler pass

to completely automate. Our solution is to have the user specify which implications they are using in their Hoare proof in an implication database. Then the user proves that these implications are compiled soundly $\textcircled{7}$ (this is the “Spec DB” proof obligation in Table 1):

► **Definition 5.** *Given φ , k , and a function environment, an IMP implication $P \Rightarrow Q$ has a valid translation if for all σ, Δ, σ_s , if $(\sigma, \Delta) \approx_\varphi \sigma_s$, then $\sigma_s \models \text{comp}_{\text{spec}}^{\varphi, k}(P) \Rightarrow \text{comp}_{\text{spec}}^{\varphi, k}(Q)$.*

While it lets us construct a proof in the target about the compiled program, it does not necessarily construct a proof of *the same* property, as the meaning of the precondition and postcondition could be destroyed by, for instance, compiling them both to \perp .

To prevent this, another proof obligation is to prove the pre/postcondition of the IMP Hoare proof sound with regards to the specification compiler (Definition 3). This guarantees that while program behavior can change, the specification remains the same. This is in Table 1 as the “Pre/Post” row. While it is required by CC, it is optional for TREE but is needed to guarantee correctness of a certificate, hence the P in the CGC column of Table 1.

These conditions only need for compilation to preserve Definitions 3 and 5 and require no proofs of *language-wide* properties, nor of *full compiler correctness*. Rather, they require specific correctness properties for a finite set of assertions. In practice, we have found these proofs to be repetitive, and have built some tactics to solve these goals $\textcircled{28}$ $\textcircled{29}$. We have not built proof automation to generate a given proof’s implication database as a verification condition but we suspect this could be done via a weakest precondition calculation.

3.2.3 Well-formedness Conditions – CC Only

The last set of user proof obligations is specific to our choice of languages and logics. Specifically, while the syntax of IMP prevents most type errors, there are other ways a program can be malformed, e.g., calling a function with an incorrect number of arguments. These obligations show that all components of the source proof be *well-formed*. Additionally, any compiled functions should preserve the stack, so as to meet the `preservesStack` condition of the STACK logic. We have largely automated these proof burdens in our case studies.

4 Case Studies

We have two sets of case studies that highlight the trade-offs of the POTPIE framework:

1. **Partial Correctness with Incorrect Compilation** (Section 4.1): We prove meaningful partial correctness properties of arithmetic approximation functions that are slightly incorrectly compiled. This set of case studies highlights two benefits of POTPIE:
 - a. **Specification-Preserving Compilation:** We invoke POTPIE with a slightly buggy program compiler to produce proofs that meaningfully preserve the correctness specifications down to the target level. Importantly, we obtain these meaningful target-level correctness proofs of our specification even though the program compiler *does not* preserve the full semantic behavior of the arithmetic approximation functions.
 - b. **Compositional Proof Compilation.** We use POTPIE to separately compile the correctness proofs of helper functions common to both approximation functions. Composition of those helper proofs within the target-level proof of the arithmetic function comes essentially “for free,” modulo termination conditions.
2. **PotPie Three Ways** (Section 4.2): We instantiate POTPIE with three different variants of a program compiler (**incomplete**, **incorrect**, and **correct but unverified**), and briefly explore the trade-offs of each of these instantiations.

■ **Table 2** The lines of code, number of theorems, and the time it took for the TREE plugin to generate and check our case studies in Section 4.1. “Core” refers to proving the source Hoare triple. “Tree” refers to how much work it took to get to the point where one could call the TREE plugin (which is different from calling the tree compiler, which is simply a one-liner), and “TreeC” the *additional* effort needed to ensure correctness. “CC” gives how much *more* work it would take to be able to use the CC workflow after ensuring tree compilation correctness.

	Multiplication				Exponentiation				Series				Square Root			
	Core	Tree	TreeC	CC	Core	Tree	TreeC	CC	Core	Tree	TreeC	CC	Core	Tree	TreeC	CC
LOC	209	104	56	508	478	107	54	362	679	174	45	630	406	154	43	286
Theorems	3	1	2	28	9	1	2	26	14	1	2	48	6	1	2	29
TREE CG (s)		0.172			0.154				2.781			4.279				
TREE Check (s)		0.131			0.098				0.534			1.946				

4.1 Partial Correctness with Incorrect Compilation

We have written and proven correct two mathematics approximation programs in IMP. Both approximation programs use common helper functions, which we also prove correct (Section 4.1.1). We then build on and compose the helper proofs to prove our approximation programs correct up to specification even in the face of incorrect compilation (Sections 4.1.2 and 4.1.3). Our incorrect compiler has the following bug, miscompiling $<$ to \leq :

$$\text{comp}_{\text{badb}}^{\varphi}(a_1 < a_2) \triangleq \text{comp}_a^{\varphi}(a_1) \leq \text{comp}_a^{\varphi}(a_2)$$

$\text{comp}_{\text{badb}}$ is a buggy boolean expression compiler that turns our less-than macro into a less-than-or-equal-to expression. While we do not have a less than operator in the IMP language, we have a less than macro defined as $a_1 < a_2 \triangleq \neg(a_1 \leq a_2 \wedge a_2 \leq a_1)$. For simplicity, we will use $<$ in this paper. The resulting program compiler ⑧ is correct for programs that do not contain $<$, and we use it throughout this subsection. We give a short summary of the proof effort that it took to prove these case studies in Table 2.

4.1.1 Helper Functions

We describe how we compile proofs about two helper functions: multiplication and exponentiation. For clarity, we omit environments in the lemmas we state here.

Multiplication. The first helper function is a multiplication function, which behaves as expected (code in green is actually wrapped Coq terms, whereas code in black is an expression in our language substituted into a Coq term as per the semantics of our logic in Figure 3):

```

1 { T }
2 x := param 0; y := 0;
3 while (1 ≤ x) do
4   y := y + param 1;
5   x := x - 1;
6 { y = (param 0) · (param 1) }
```

The proof of this IMP Hoare triple is straightforward since the body of the function does not encounter the incorrect behavior of the compiler. By combining this triple with a termination proof, we are able to generate a helper lemma ⑨ that relates applications of the IMP multiplication function to Coq’s `Nat.mul`:

```
Lemma mult_aexp_wrapper a1 a2 n1 n2: a1 ↓ n1 → a2 ↓ n2 → mult(a1, a2) ↓ (n1 * n2)%nat.
```

This lemma lets us reason more directly about `nats`. We use this lemma in the subsequent case studies, demonstrating how POTPIE enables us to reuse the source Hoare proof of this triple to get the target-level version of this lemma *almost* for free – we still have to reprove termination at the target level, something we hope to address in future work.

Exponentiation. Exponentiation is similarly straightforward, except we use multiplication as defined above as a function in its body and thus must use the multiplication function wrapper to prove the loop invariant, and we obtain the following wrapper [\(10\)](#):

```
Lemma exp_aexp_wrapper : forall a1 a2 n1 n2, a1 ↓ n1 → a2 ↓ n2 → exp(a1, a2) ↓ n2n1.
```

4.1.2 Geometric Series

One example use case for partial correctness specifications is floating point estimation of mathematical functions, like $\sin(x)$ and e^x , by way of computing infinite series with well-behaved error terms. Since floating point numbers are unable to represent all of the reals, we must approximate these functions within some error bound. As a simple version of this use case, we consider a program for calculating the geometric series $\sum_{i=1}^{\infty} \frac{1}{x^i}$ within an error bound of $\epsilon = \frac{\delta_n}{\delta_d}$. We require $x \geq 2$ so that the series converges, which simplifies some of our assertions for this example. While this is a toy example that would be easier to compute in its closed form – the series $\sum_{i=0}^{\infty} a \cdot r^i$ is known to converge to $\frac{a}{1-r}$ for $|r| < 1$, it suffices as a simple example of using POTPIE with an interesting partial specification. We cover a more realistic example in Section 4.1.3. The program we use to compute this series is as follows:

```
1 { 2 ≤ x ∧ x = x ∧ δn ≠ 0 ∧ δd ≠ 0 ∧ 1 = 1; ∧ x = x ∧ 2 = 2 }
2   x := x; // the series denominator
3   rn := 1; // the result numerator
4   rd := x; // the result denominator (for i = 1)
5   i := 2; // the next exponent
6   { rn · xi - rn · xi-1 = rd · xi-1 - rd ∧ x = x ∧ 2 ≤ x ∧ 2 ≤ i } // loop invariant
7   // the loop condition is equivalent to ε < 1/(x-1) - rn/rd, and 1/(x-1) = ∑i=1∞ 1/xi
8   while (mult(rn, δd · (x-1)) + mult(rd, δn · (x-1)) < mult(rd, δd)) do
9     d := exp(x, i);
10    rn := frac_add_numerator(rn, rd, 1, d); // a/b + c/d = (ad + cb)/(bd)
11    rd := frac_add_denominator(rd, d); // fraction addition denominator
12    i := i + 1;
13 { ¬ (mult(rn, δd · (x-1)) + mult(rd, δn · (x-1)) < mult(rd, δd))
14   ∧ (rn · xi - rn · xi-1 = rd · xi-1 - rd ∧ x = x ∧ 2 ≤ x ∧ 2 ≤ i) } // loop
15 { δd · rd ≤ δn · (x-1) · rd + δd · (x-1) · rn } // program postcondition: 1/(x-1) - rn/rd ≤ δn/δd
```

For brevity, we omit assertions outside of the pre/postcondition, loop invariant, and loop postcondition. We show wrapped Coq Props and arithmetic terms in green, i.e. $\delta_n \cdot (x-1)$. Terms in black are IMP expressions. Note that we encounter the bug in our program compiler, which miscompiles the `<` in the while loop conditional. However, we are still able to compile this program and its proof to STACK because (1) the pre/postconditions' meaning is preserved by compilation, and (2) the implication database is still valid, i.e., every compiled IMP implication is still an implication in STACK.

To see (1), we will need to look at the underlying representation of our assertions. As given in Figure 3, our precondition and postcondition actually have the following form:

```
(fun x' rn' rd' i' => 2 ≤ x' ∧ x' = x ∧ δn ≠ 0 ∧ δd ≠ 0 ∧ rn' = 1 ∧ rd' = x ∧ i' = 2) x 1 x 2
(fun rn' rd' => δd · rd' ≤ δn · (x-1) · rd' + δd · (x-1) · rn') rn rd
```

Everything after the anonymous function is actually an expression in the IMP language. These are the only parts of the assertions that are compiled by the specification compiler.

For instance, x is a constant arithmetic expression in IMP, which wraps Coq's `nat` type. The arithmetic compiler, `compa`, from Figure 2 compiles these to `nat` constants in the STACK language. For the variables `rn` and `rd`, `compaφ,k(rn) = #φ(rn)`. After compiling, we get the postcondition $\delta_d \cdot \#5 \leq \delta_n \cdot (x-1) \cdot \#5 + \delta_d \cdot (x-1) \cdot \#2$, or symbolically: $\frac{1}{x-1} - \frac{\#2}{\#5} \leq \frac{\delta_n}{\delta_d}$.

For (2), we have to show that every implication in the IMP implication database is compiled to a valid implication in STACK. The implication most relevant to the successful compilation of the proof is the last one, which implies the program's postcondition. Since the IMP loop condition `<` gets compiled to `<=` in STACK, our negated loop condition becomes

$$\neg (\text{mult}(\#2, \delta_d \cdot (x-1)) + \text{mult}(\#5, \delta_n \cdot (x-1)) \leq \text{mult}(\#5, \delta_d))$$

This is equivalent to the below inequality (where \equiv denotes “is numerically equivalent to”), which still implies the compiled postcondition. This is easily proved with the `Psatz.lia` tactic.

$$\text{mult}(\#5, \delta_d) < \text{mult}(\#2, \delta_d \cdot (x-1)) + \text{mult}(\#5, \delta_n \cdot (x-1)) \equiv \frac{1}{x-1} - \frac{\#2}{\#5} < \frac{\delta_n}{\delta_d}$$

4.1.3 Square Root

The second approximation program we consider interacts with the same miscompilation and still meaningfully preserves the source specification. Given numbers $a, b, \epsilon_n, \epsilon_d$, we consider a square root approximation program that calculates some x, y such that $|\frac{x^2}{y^2} - \frac{a}{b}| \leq \frac{\epsilon_n}{\epsilon_d}$. We can project the postcondition entirely into Coq terms, multiplying through both sides by the denominator so we can express it in our language. After writing the program, we come up with the following loop condition, which represents $\frac{\epsilon_n}{\epsilon_d} < \left| \frac{x^2}{y^2} - \frac{a}{b} \right|$ (`.` is syntactic sugar for `mult`, and `<` is actually the IMP less-than macro):

$$\text{loop_cond} \triangleq (y \cdot y \cdot b \cdot \epsilon_n < y \cdot y \cdot a \cdot \epsilon_d - x \cdot x \cdot a \cdot \epsilon_d) \vee (y \cdot y \cdot b \cdot \epsilon_n < x \cdot x \cdot b \cdot \epsilon_d - y \cdot y \cdot a \cdot \epsilon_d)$$

Our IMP square root program and specification is given by the following.

```

1 {T}
2   x      := a;   y      := mult(2, b);
3   inc_n  := a;   inc_d  := mult(2, b);
4   while (loop_cond) do
5     inc_d := mult(2, inc_d);
6     if (mult(mult(y, y), mult(a, εd)) ≤ mult(mult(x, x), mult(b, εd)))
7       then x := frac_sub_numerator(x, y, inc_n, inc_d);
8     else x := frac_add_numerator(x, y, inc_n, inc_d);
9     y := frac_add_denominator(y, inc_d);
10 { ¬loop_condition ∧ T } ⇒
11 { ((x · x · b · εd) - (y · y · a · εd) ≤ y · y · b · εn) ∧ ((y · y · a · εd) - (x · x · b · εd) ≤ y · y · b · εn) }
```

Most of the rules of consequence are straightforward. The only nontrivial implication involved is the final rule of consequence for the postcondition. The loop's postcondition is $\neg \left(\frac{\epsilon_n}{\epsilon_d} < \left| \frac{x^2}{y^2} - \frac{a}{b} \right| \right) \equiv \left| \frac{x^2}{y^2} - \frac{a}{b} \right| \leq \frac{\epsilon_n}{\epsilon_d}$, which directly gets us the program postcondition.

During compilation, the loop condition is miscompiled: the program compiler changes `<` to `≤`. This results in the following target loop condition, where again, `mult` is represented by `.`. Note this is not green since it represents an expression in STACK, not a Coq one.

$$\text{stk_loop_cond} \triangleq \#1 \cdot \#1 \cdot b \cdot \epsilon_n \leq \#1 \cdot \#1 \cdot a \cdot \epsilon_d - \#4 \cdot \#4 \cdot b \cdot \epsilon_d \\ \vee \#1 \cdot \#1 \cdot b \cdot \epsilon_n \leq \#4 \cdot \#4 \cdot b \cdot \epsilon_d - \#1 \cdot \#1 \cdot a \cdot \epsilon_d$$

Compared to the target program and proof, the main difference is in the final application of the rule of consequence, where the incorrect behavior of the compiler appears and changes the semantics of the loop condition. The programs have meaningfully different semantics, and those meaningfully different semantics do manifest in the application of the while rule.

```

1 {(T,T)}
2   push; push; push; push;
3   #4 := a; #1 := mult(2, b);
4   #3 := a; #2 := mult(2, b);
5   {4, T}
6   while (stk_loop_cond) do
7     #2 := mult(2, #2);
8     if (mult(mult(#1, #1), mult(a, εd)) ≤ mult(mult(#4, #4), mult(a, εd)))
9     then #4 := frac_sub_numerator(#4, #1, #3, #2);
10    else #4 := frac_add_numerator(#4, #1, #3, #2);
11    #1 := frac_add_denominator(#1, #2)
12 {(4, ¬target_loop_condition) /\ (4,T)} ==>
13 {4, (#4·#4·b·εd) - (#1·#1·a·εd) ≤ (#1·#1·b·εn) ∧ ((#1·#1·a·εd) - (#4·#4·b·εd) ≤ #1·#1·b·εn)}

```

While the loop condition is indeed miscompiled, the postcondition uses Coq’s \leq , so the postcondition is *not*. Even though the unsound behavior of the compiler changes the semantics of the loop invariant, it is not enough to break the implication between the loop condition and the Coq-wrapped loop condition. Further, because of the way that the postcondition projects into Coq, the final implication is almost completely provable via applications of helper lemmas from Section 4.1.1 and the tactics `inversion` and `Psatz.lia`.

4.2 PotPie Three Ways

POTPIE makes it easy to swap out control-flow-preserving program compilers and still reuse the same infrastructure. We instantiate POTPIE with three variants of a program compiler, and use these on three small programs: `shift` (left-shift) (14), `max` (15) (16), and `min` (17):

1. An **incomplete program compiler** (18) that is missing entire cases of the source language grammar. Only `shift` can be compiled using the incomplete proof compiler.
2. An **incorrect program compiler** (19) that contains a mistake and an unsafe optimization, in a similar vein to the previous examples. We can compile `max` using it, but not `min`.
3. An **unverified correct program compiler** (20) that always preserves program and specification behavior. This can be used to proof compile all of the programs.

These examples show we are able to instantiate the POTPIE framework for several different compilers, and POTPIE is compatible with correct compilers as well. We are able to invoke the CC and TREE compilers with all of these case studies as well.

5 Implementation

While much of our proof development for POTPIE is implemented in Coq, the TREE plugin is implemented in OCaml (Section 5.1). We prove that POTPIE is sound for both workflows (Section 5.2) and keep POTPIE’s *trusted computing base* small (Section 5.3).

5.1 The Tree Plugin

The TREE plugin is implemented in OCaml, and consists of about 2.2k lines of code (LOC). Much of the code (~ 1.1 k LOC) is simply copied from the reusable plugin library `coq-plugin-lib`¹ and updated to Coq 8.16.1. Additionally, such a plugin only has to be created *once* per target language-logic pair, and is *completely independent* from compilation. Indeed, the plugin can be called on any STACK Hoare tree – the tree need not be the result of compilation. While

¹ <https://github.com/uwplse/coq-plugin-lib>

■ **Table 3** The proof engineering effort that went into stating and formalizing POTPIE, including the infrastructure to support the code and spec languages, logics, the compilers, the case studies, and automation. Here, “Thms” means the number of **Theorems** and **Lemmas**, while “Specs” means the number of **Definitions**, **Fixpoints**, and **Inductives**. “WF” stands for well-formed, “Insts.” for instantiations of CC compilers, “Cases” for our case studies, and “Auto” for automation. “Base Props” refers to code related to the base assertions seen in Figure 3.

	IMP			STACK			Base Props	Compiler				Insts.	Cases	Auto	Misc	Total
	Lang	Logic	WF	Lang	Logic	WF		Code	Spec	TREE	CC					
LOC	808	1948	3605	2593	1077	5635	941	1102	159	780	3045	2133	6971	2914	3225	36936
Thms	15	67	103	91	17	204	37	44	2	17	93	52	288	31	105	1166
Specs	43	32	51	29	51	63	31	25	14	13	40	100	238	50	107	887

Table 1 indicates that the plugin automates a check for the commutativity equations from Section 3.2.1, this is because the properties checked by the plugin *imply* the commutativity equations for the included TREE proof compiler in our code ②– it never actually checks the commutativity equations themselves. This makes TREE more flexible than the CC approach.

The plugin is called on a STACK tree, function environment, implication database (with proof of its validity), and list of functions. Here we call it on our multiplication example:

```

1 Certify (MultTargetTree.tree) (MultTargetTree.fenv) (ProdTargetTree.facts)
2   (MultValidFacts.valid_facts) (MultTargetTree.funcs) as mult.
3 Check mult.

```

`mult` contains the answer returned by the plugin. If the plugin is set to generate certificates and it is successful, `mult` has type `stk_valid_tree`. Otherwise, `mult` is a Coq `bool`.

The plugin recurses over the input tree and attempts to construct the certificate ②1. This may fail if the tree is malformed or there are mutually recursive functions. As we saw in Section 2.2, the STACK logic requires that all expressions preserve the stack, which is represented by the relation `exp_stack_pure_rel` ③. However, due to the semantics of STACK functions, we need to know that all function calls preserve the stack, and showing that `exp_stack_pure_rel` is true in the presence of mutually recursive functions would lead to an infinite loop. If certificate generation fails, the plugin tries to provide a boolean answer as a fallback mechanism. It does this by checking each function for stack-preserving behavior modulo the behavior of other functions ②3, then checking the proof tree recursively ②4.

As we saw in Table 2, the certificate generator and tree checking algorithms are fairly performant. This is due to several caching and reduction algorithm optimizations we made. Before applying optimizations, the series and square root examples took *>10 minutes* to generate certificates, and now take *<5 seconds*. The main bottleneck was Coq’s δ -reductions, which unfold constants. Our plugin provides an option to treat certain functions as “opaque” inside the plugin ②7, leaving their constants folded and speeding up normalization. This *does not* change the user’s Coq environment. The plugin also uses unification (for example, to match with constructors of option types ③2) to avoid all but one call to normalization, which we found to significantly improve performance.

5.2 Formal Proof

Our Coq formalization includes two proofs of soundness, one for each of the workflows, as well as all of the case studies from Section 4. The CC soundness proof ① takes the form of a correct-by-construction function that takes a source Hoare proof, the well-formedness conditions, and the implication translation, and produces a verified Hoare proof in the target, as described in Section 3.1. For TREE, we prove that if all of the obligations for CC are

satisfied, then the compiled tree is valid (12). As we mentioned in Section 3.1, we additionally show that when the OCaml plugin (5) generates a certificate that typechecks, the certificate can be used to obtain an `hl_stk` proof.

We loosely based our code on Xavier Leroy’s course on mechanized semantics [30]. The LOC numbers for our proof development in Table 3 are large when compared to the size of Leroy’s course materials, but there are several key differences. First, our languages include functions, making our semantics more difficult to reason about than the course’s semantics. The trade-off is that functions give us the opportunity to reason about the composition of programs and their proofs (Section 4.1). Second, our target language is far less well-behaved than either of the languages in the course. Third, POTPIE supports two different workflows, two separate proof compilers that work to get guarantees even for incorrect compilation.

5.3 Trusted Computing Base (TCB)

POTPIE’s two workflows for proof compilation have different TCBs and provide different levels of guarantees. The CC proof compiler’s TCB consisting of the Coq kernel, the mechanized semantics, the definition of the Hoare triple, and two localized Uniqueness of Identity Proofs (UIP) axioms for reasoning about the equalities between dependent types. UIP, which is consistent with Coq, states that any two equality proofs are equal for *all* types – we instead assume that equality proofs are equal to each other for *two particular types*, `AbsEnvs` (25) (the implementation of *SM* from Section 2.2) and function environments (26). This does not imply universal UIP but is similarly convenient for proof engineering. Whenever all of its proof obligations can be satisfied, the correct-by-construction proof compiler is guaranteed to produce a correct proof. However, the resulting proof object may not be independent from the source semantics, due to various opaque proof terms that cannot be further reduced.

The TREE plugin can either generate a certificate or run a check on a proof tree, returning its validity as a boolean. The *certificate generator* has a strictly smaller TCB than CC since it does not assume any form of UIP. The certificate generator works by generating a term of type `stk_valid_tree` (22). Since this term must still be type-checked in Coq for it to be considered valid, this does not add to the TCB. The TREE *boolean proof tree checker* has its own “kernel,” also implemented in OCaml, for checking proof trees, which adds to its TCB. While it does not imply formal correctness, it can boost confidence in compiled proofs.

6 Related Work and Discussion

Early work on compiling proofs positioned itself as an extension of **proof-carrying code** [35]. A 2005 paper [4] stated a theorem relating source and target program logics. Early work [33] transformed Hoare-style proofs about Java-like programs to proofs about bytecode implemented in XML. Later work [37] implemented **proof-transforming compilation**, transforming proof objects from Eiffel to bytecode, and formalizing the specification compiler in Isabelle/HOL, with a hand-written proof of correctness of the proof compiler. Subsequent work [16] showed how to embed the compiled bytecode proofs into Isabelle/HOL. Our work is the first we know of to formally verify the correctness of the proof compiler, and to use it to support specification-preserving compilation in the face of incorrect program compilation. Existing work on **certificate translation** [3, 26], which is similar but focuses on compiler optimizations, may help us relax control-flow restrictions.

There is a lens through which our work is related to **type-preserving compilation**: compiling programs in a way that preserves their types. There is work on this defined on a subset of Coq for CPS [7] and ANF [21] translations. As the source and target languages

both have dependent types, this can likewise be used to compile proofs while preserving specifications. A similar line of work can be found for compilations of proof languages in Metamath Zero [9]. Our work focuses on compiling program logic proofs instead.

Our work implements a certified **proof transformation** in Coq for an embedded program logic. Proof transformations were introduced in 1987 to bridge automation and usability [39], and have since been used for proof generalization [15, 20, 17], reuse [31], and repair [41].

The golden standard for correct compilation is **certified compilation**: formally proving compilers correct. The CompCert verified C compiler [29, 28] lacks bugs present in other compilers [47]. The CakeML [25] verified implementation of ML includes a verified compiler. Oeuf [32] and CertiCoq [2] are certified compilers for Coq’s term language Gallina. Certified compilation is desirable when possible, but real compilers may be unverified, incomplete, or incorrect. Our work complements certified compilation by exploring an underexplored part of the design space of compiler correctness: compilation that is **specification-preserving** for a given source program and (possibly partial) specification, even when the compilation may not be fully **meaning-preserving** for that program. The original CompCert paper [28] brought up the possibility of specification-preserving compilation as part of a design space that is *complementary* to, not in competition with, certified compilation. We agree; it expands the space of guarantees one can get for compiled programs – even when those programs are incorrectly compiled. It also expands the means by which one may get said guarantees.

Our work implements a kind of **certifying compilation**: producing compiled code and a proof that its compilation is correct. For example, COGENT’s certifying compiler proves that, for a given program compiled from COGENT to C, target code correctly implements a high-level semantics embedded in Isabelle/HOL [1, 42]. Certifying compilation shares the benefit that the compiler may be incorrect or incomplete, yet still produce proofs about the compiled program. Most prior work on certifying compilation that we are aware of targets general properties (like type safety) rather than program-specific ones. One exception is *Rupicola* [40], a framework for correct but incomplete compilation from Gallina to low-level code using proof search, which focuses on preservation of program-specific specifications proven at the source level like we do. But it does not appear to address the case when the program itself is incorrectly compiled, nor the case where there already exists an unverified complete program compiler. Our work adds to the space of certifying compilation by preserving program-specific partial specifications proven at the source level even when the program itself is compiled incorrectly, with the added benefit of compositionality.

One immensely practical method for showing that programs compiled with unverified compilers preserve behavior is **translation validation**. In translation validation, the compiler produces a proof of the correctness of a particular program’s compilation, which then needs to be checked [36]. Our work is in a similar spirit, but distinguishes itself in that our method does not rely on functional equivalence for the particular compiled program. Our method makes it possible to show that a compiler preserves a partial specification when the program is miscompiled in ways that are not relevant to the specification.

Section 4.1.1 shows in a limited context our method’s potential for **compositionality**. Similar motivation is behind (much more mature) work in compositional certified compilation [46, 14, 19]. DimSum [43] defines an elegant and powerful language-and-logic-agnostic framework for language interoperability, though to get guarantees, it leans heavily on data refinement arguments that show a simulation property stronger than what our framework requires. We hope that in the future, we will make our compositional workflow more systematic and fill the gap of compositional multi-language reasoning in a relaxed correctness setting – by linking compiled *proofs* directly in a common target logic. Similar motivations

are behind linking types [38], which are extensions to type systems for reasoning about correct linking in a multilanguage setting. We expect tradeoffs similar to those between our work and type-preserving compilation to arise in this setting.

Frameworks based on embedded **program logics** (e.g., Iris [18, 24], VST-Floyd [8], Bedrock [12, 13], YNot [34], CHL [11], SEPREF [27], and CFML [10]) help proof engineers write proofs in a proof assistant about code with features that the proof assistant lacks. C programs verified in the VST program logic are, by composition with CompCert, guaranteed to preserve their specifications even after compilation to assembly code [5]. Our work aims to create an alternative toolchain for preserving guarantees across compilation that allows the program compiler to be unverified or even incorrect, even for the program being compiled. Relative to practical frameworks like Iris and VST, the program logics we use for this are much less mature. We hope to extend our work to more practical logics and lower-level target languages in the future, so that users of toolchains like VST can get guarantees about compiled programs even in the face of incorrect compilation.

7 Conclusion

We showed how compiling proofs across program logics can empower proof engineers to reason directly about source programs yet still obtain proofs about compiled programs – even when they are incorrectly compiled. Our implementation POTPIE and its two workflows, CC and TREE, are formally verified in Coq, providing guarantees that compiled proofs not only prove their respective specifications, but also are correctly related to the source proofs. Our hope is to provide an alternative to relying on verified program compilers without sacrificing important correctness guarantees of program specifications.

Future Work. In this work, we have not tackled the problem of control flow optimizations. We believe the challenges of bridging abstraction levels and verifying control flow-modifying optimizations are mostly orthogonal, and that the latter is out of our scope. In future work, we would like to investigate ways our work could be composed with control flow optimizations. For example, we may be able to leverage Kleene algebras with tests (KAT) [22] to reason about control flow optimizations. An optimization pass could extract a proof subtree and return the optimized subprogram, while preserving semantic equality via KAT. This approach may even be able to leverage a Hoare triple’s preconditions to apply optimizations that would be otherwise unsound [23]. For an example of KATs applied to existing compiler optimizations, see existing work [22]. Beyond relaxing control flow restrictions, other next steps include supporting more source languages and logics, supporting additional linking of target-level proofs, implementing optimizing compilers, and bringing the benefits of proof compilation to more practical frameworks.

We also have not addressed the issue of scalability. As we outlined in Section 1, that was not in the scope of this paper. We do however have some ideas for expanding scalability. There are two main issues of scale: (1) applying the methodology here to more complex programming languages and program logics, and (2) how easily proof compilers can be implemented and used. For more complex languages and logics, we are currently implementing a language with pointers and an accompanying separation logic, as well as a stack language with stack pointer expressions. This will give us a better idea of the effort involved to scale to more languages. As for implementing and using proof compilers, we found that the TREE version of the proof compiler was very easy to write, and the plugin consists of only 1.1k new LOC as we saw in Section 5.1. We believe that significant parts of that code could have been automatically generated as well, which would further decrease the time needed to create such a proof compiler. We are excited to explore these directions, as well as others, in the future.

References

- 1 Sidney Amani, Alex Hixon, Zilin Chen, Christine Rizkallah, Peter Chubb, Liam O'Connor, Joel Beeren, Yutaka Nagashima, Japheth Lim, Thomas Sewell, Joseph Tuong, Gabriele Keller, Toby Murray, Gerwin Klein, and Gernot Heiser. Cogent: Verifying high-assurance file system implementations. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 175–188, Atlanta, GA, USA, April 2016. doi:10.1145/2872362.2872404.
- 2 Abhishek Anand, Andrew W. Appel, Greg Morrisett, Matthew Weaver, Matthieu Sozeau, Olivier Savary Belanger, Randy Pollack, and Zoe Paraskevopoulou. CertiCoq: A verified compiler for Coq. In *CoqPL*, 2017. URL: <http://www.cs.princeton.edu/~appel/papers/certicoq-coqpl.pdf>.
- 3 Gilles Barthe, Benjamin Grégoire, César Kunz, and Tamara Rezk. Certificate translation for optimizing compilers. *ACM Trans. Program. Lang. Syst.*, 31(5), July 2009. doi:10.1145/1538917.1538919.
- 4 Gilles Barthe, Tamara Rezk, and Ando Saabas. Proof obligations preserving compilation. In *Formal Aspects in Security and Trust: Thrid International Workshop, FAST 2005, Newcastle upon Tyne, UK, July 18-19, 2005, Revised Selected Papers*, pages 112–126. Springer, 2005.
- 5 Lennart Beringer and Andrew W. Appel. Abstraction and subsumption in modular verification of C programs. *Formal Methods in System Design*, 58(1):322–345, October 2021. doi:10.1007/s10703-020-00353-1.
- 6 Sandrine Blazy and Xavier Leroy. Mechanized Semantics for the Clight Subset of the C Language. *Journal of Automated Reasoning*, 43(3):263–288, October 2009. doi:10.1007/s10817-009-9148-3.
- 7 William J. Bowman, Youyou Cong, Nick Rioux, and Amal Ahmed. Type-preserving cps translation of Σ and Π types is not possible. *Proc. ACM Program. Lang.*, 2(POPL), December 2017. doi:10.1145/3158110.
- 8 Qinxiang Cao, Lennart Beringer, Samuel Gruetter, Josiah Dodds, and Andrew W. Appel. VST-Floyd: A Separation Logic Tool to Verify Correctness of C Programs. *Journal of Automated Reasoning*, 61(1):367–422, June 2018. doi:10.1007/s10817-018-9457-5.
- 9 Mario Carneiro. Metamath zero: Designing a theorem prover prover. In Christoph Benzmüller and Bruce Miller, editors, *Intelligent Computer Mathematics*, pages 71–88, Cham, 2020. Springer International Publishing.
- 10 Arthur Charguéraud. Characteristic formulae for the verification of imperative programs. *SIGPLAN Not.*, 46(9):418–430, September 2011. doi:10.1145/2034574.2034828.
- 11 Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. Using crash hoare logic for certifying the fscq file system. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP '15*, pages 18–37, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2815400.2815402.
- 12 Adam Chlipala. Mostly-automated verification of low-level programs in computational separation logic. *SIGPLAN Not.*, 46(6):234–245, June 2011. doi:10.1145/1993316.1993526.
- 13 Adam Chlipala. The bedrock structured programming system: Combining generative metaprogramming and hoare logic in an extensible program verifier. *SIGPLAN Not.*, 48(9):391–402, September 2013. doi:10.1145/2544174.2500592.
- 14 Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. Deep specifications and certified abstraction layers. *SIGPLAN Not.*, 50(1):595–608, January 2015. doi:10.1145/2775051.2676975.
- 15 Robert W Hasker and Uday S Reddy. Generalization at higher types. In *Proceedings of the Workshop on the λ Prolog Programming Language*, pages 257–271, 1992.
- 16 Bruno Hauser. Embedding proof-carrying components into Isabelle. Master's thesis, ETH, Swiss Federal Institute of Technology Zurich, Institute of Theoretical . . . , 2009.
- 17 Einar Broch Johnsen and Christoph Lüth. Theorem reuse by proof term transformation. In *Theorem Proving in Higher Order Logics: 17th International Conference, TPHOLs 2004*,

- Park City, Utah, USA, September 14-17, 2004. Proceedings*, pages 152–167. Springer, Berlin, Heidelberg, 2004. doi:10.1007/978-3-540-30142-4_12.
- 18 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. *SIGPLAN Not.*, 50(1):637–650, January 2015. doi:10.1145/2775051.2676980.
 - 19 Jérémie Koenig and Zhong Shao. Compcerto: Compiling certified open c components. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2021*, pages 1095–1109, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3453483.3454097.
 - 20 Thomas Kolbe and Christoph Walther. *Proof Analysis, Generalization and Reuse*, pages 189–219. Springer, Dordrecht, 1998. doi:10.1007/978-94-017-0435-9_8.
 - 21 Paulette Koronkevitch, Ramon Rakow, Amal Ahmed, and William J. Bowman. Anf preserves dependent types up to extensional equality. *Journal of Functional Programming*, 32:e12, 2022. doi:10.1017/S0956796822000090.
 - 22 Dexter Kozen and Maria-Cristina Patron. Certification of compiler optimizations using kleene algebra with tests. In John Lloyd, Veronica Dahl, Ulrich Furbach, Manfred Kerber, Kung-Kiu Lau, Catuscia Palamidessi, Luís Moniz Pereira, Yehoshua Sagiv, and Peter J. Stuckey, editors, *Computational Logic — CL 2000*, pages 568–582, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg. doi:10.1007/3-540-44957-4_38.
 - 23 Dexter Kozen and Jerzy Tiuryn. On the completeness of propositional hoare logic. *Information Sciences*, 139(3):187–195, 2001. Relational Methods in Computer Science. doi:10.1016/S0020-0255(01)00164-5.
 - 24 Robbert Krebbers, Amin Timany, and Lars Birkedal. Interactive proofs in higher-order concurrent separation logic. *SIGPLAN Not.*, 52(1):205–217, January 2017. doi:10.1145/3093333.3009855.
 - 25 Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML: A verified implementation of ML. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL ’14*, pages 179–191, New York, NY, USA, 2014. ACM. doi:10.1145/2535838.2535841.
 - 26 César Kunz. *Certificate Translation Alongside Program Transformations*. PhD thesis, ParisTech, Paris, France, 2009.
 - 27 Peter Lammich. Refinement to imperative HOL. *J. Autom. Reason.*, 62(4):481–503, April 2019. doi:10.1007/s10817-017-9437-1.
 - 28 Xavier Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *33rd ACM symposium on Principles of Programming Languages*, pages 42–54. ACM Press, 2006. URL: <http://xavierleroy.org/publi/compiler-certif.pdf>.
 - 29 Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, July 2009. doi:10.1145/1538788.1538814.
 - 30 Xavier Leroy. Coq development for the course “Mechanized semantics”, 2019-2021. URL: <https://github.com/xavierleroy/cdf-mech-sem>.
 - 31 Nicolas Magaud. Changing data representation within the Coq system. In *International Conference on Theorem Proving in Higher Order Logics*, pages 87–102. Springer, 2003.
 - 32 Eric Mullen, Stuart Pernsteiner, James R. Wilcox, Zachary Tatlock, and Dan Grossman. Oeuf: Minimizing the Coq extraction TCB. In *CPP*, pages 172–185, New York, NY, USA, 2018. ACM. doi:10.1145/3167089.
 - 33 Peter Müller and Martin Nordio. Proof-transforming compilation of programs with abrupt termination. In *SAVCBS*, pages 39–46, January 2007. doi:10.1145/1292316.1292321.
 - 34 Aleksandar Nanevski, Greg Morrisett, Avraham Shinnar, Paul Govereau, and Lars Birkedal. Ynot: Dependent types for imperative programs. *SIGPLAN Not.*, 43(9):229–240, September 2008. doi:10.1145/1411203.1411237.

- 35 George C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, pages 106–119, New York, NY, USA, January 1997. Association for Computing Machinery. doi:10.1145/263699.263712.
- 36 George C. Necula. Translation validation for an optimizing compiler. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 83–94, New York, NY, USA, 2000. Association for Computing Machinery. doi:10.1145/349299.349314.
- 37 Martin Nordio, Peter Müller, and Bertrand Meyer. Formalizing proof-transforming compilation of Eiffel programs. Technical report, ETH Zurich, 2008.
- 38 Daniel Patterson and Amal Ahmed. Linking Types for Multi-Language Software: Have Your Cake and Eat It Too. In Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi, editors, *2nd Summit on Advances in Programming Languages (SNAPL 2017)*, volume 71 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 12:1–12:15, Dagstuhl, Germany, 2017. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.SNAPL.2017.12.
- 39 Frank Pfenning. *Proof Transformations in Higher-Order Logic*. PhD thesis, Carnegie Mellon University, 1987.
- 40 Clément Pit-Claudel, Jade Philipoom, Dustin Jamner, Andres Erbsen, and Adam Chlipala. Relational compilation for performance-critical applications: Extensible proof-producing translation of functional models into low-level code. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022, pages 918–933, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3519939.3523706.
- 41 Talia Ringer. *Proof Repair*. PhD thesis, University of Washington, 2021.
- 42 Christine Rizkallah, Japheth Lim, Yutaka Nagashima, Thomas Sewell, Zilin Chen, Liam O'Connor, Toby Murray, Gabriele Keller, and Gerwin Klein. A framework for the automatic formal verification of refinement from Cogent to C. In *Interactive Theorem Proving*, pages 323–340, Cham, 2016. Springer. doi:10.1007/978-3-319-43144-4_20.
- 43 Michael Sammler, Simon Spies, Youngju Song, Emanuele D'Oswaldo, Robbert Krebbers, Deepak Garg, and Derek Dreyer. Dimsum: A decentralized approach to multi-language semantics and verification. *Proc. ACM Program. Lang.*, 7(POPL), January 2023. doi:10.1145/3571220.
- 44 Audrey Seo, Christopher Lam, Dan Grossman, and Talia Ringer. uwplse/potpie. Software, swbId: swb:1:dir:5b78a12be7ef95b92fe5db2acf903d436e951851 (visited on 2024-08-21). URL: <https://github.com/uwplse/potpie/tree/v0.5>.
- 45 The Coq Development Team. The coq proof assistant, July 2023. doi:10.5281/zenodo.8161141.
- 46 Yuting Wang, Pierre Wilke, and Zhong Shao. An abstract stack based approach to verified compositional compilation to machine code. *Proc. ACM Program. Lang.*, 3(POPL), January 2019. doi:10.1145/3290375.
- 47 Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 283–294, New York, NY, USA, 2011. Association for Computing Machinery. doi:10.1145/1993498.1993532.

Redex2Coq: Towards a Theory of Decidability of Redex's Reduction Semantics

Mallku Soldevila   

FAMAF, UNC, Córdoba, Argentina
CONICET, Buenos Aires, Argentina

Rodrigo Ribeiro   

DECOM, UFOP, Ouro Preto, Brazil

Beta Ziliani   

FAMAF, UNC, Córdoba, Argentina
Manas.Tech, Buenos Aires, Argentina

Abstract

We propose the first step in the development of a tool to automate the translation of Redex models into a semantically equivalent model in Coq, and to provide tactics to help in the certification of fundamental properties of such models.

The work is based on a model of Redex's semantics developed by Klein *et al.* In this iteration, we were able to code in Coq a primitive recursive definition of the matching algorithm of Redex, and prove its correctness with respect to the original specification. The main challenge was to find the right generalization of the original algorithm (and its specification), and to find the proper well-founded relation to prove its termination.

Additionally, we also adequate some parts of our mechanization to prepare it for the future inclusion of Redex features absent in Klein *et al.*, such as the Kleene's closure operator.

2012 ACM Subject Classification Theory of computation → Formal languages and automata theory; Theory of computation → Rewrite systems; Theory of computation → Interactive proof systems

Keywords and phrases Coq, PLT Redex, Reduction semantics

Digital Object Identifier 10.4230/LIPIcs.ITP.2024.34

Supplementary Material *Software (Source code)*: <https://github.com/Mallku2/redex2coq> [15] archived at `swh:1:dir:eea74f34ec4a10a6e7315b1d5d3f89327bce18a5`

Funding *Mallku Soldevila*: CONICET, Argentina.

1 Introduction

Redex [5] is a DSL built on top of the Racket programming language, which allows for the mechanization of reduction semantics models and formal systems. It includes a variety of tools for testing the models, including: unit testing; random testing of properties; and a stepper for step-by-step reduction sequences. Given its toolkit, Redex has been successfully used for the mechanization of large semantics models of real programming languages (*e.g.*, JavaScript [6, 11]; Python [12]; Scheme [8]; and Lua [17, 16, 14]).

The approach of Redex to semantics engineering involves a lightweight development of models that focuses on a quick transition between specification of models and testing of their properties. These virtues of Redex enable it as a useful tool with which to perform the first steps of a formalization effort. Nonetheless, when a given model seems to be thoroughly tested and mature, one still might need to prove its desired properties, since no amount of testing can guarantee the absence of errors [3].

Redex does not offer tools for formal verification of a given model, and there are no fully developed automatic tools to export the model into some proof assistant. Hence, for verification purposes, it is common for a given model to be written again entirely into a proof



© Mallku Soldevila, Rodrigo Ribeiro, and Beta Ziliani;
licensed under Creative Commons License CC-BY 4.0

15th International Conference on Interactive Theorem Proving (ITP 2024).

Editors: Yves Bertot, Temur Kutsia, and Michael Norrish; Article No. 34; pp. 34:1–34:18

Leibniz International Proceedings in Informatics



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

assistant. Besides being a time-consuming process, another downside is that the translation into the proof assistant may be guided just by an intuitive understanding of the behavior of the mechanization in Redex. And that intuitive understanding could differ from the actual behavior of the model in Redex. This is so, since the tool implements a particular meaning of reduction semantics with evaluation contexts, offering an expressive language to the user that includes several features, useful to express concepts like context-dependent syntactic rules. The actual semantics of this language may not coincide with what the researcher understands, as exposed in [4].

In this work, we propose to build a tool to automatically translate a given model in Redex into an equivalent model in Coq, where the interpretation of the resulting model is done through a shallow embedding in Coq of Redex’s actual semantics, as formalized in [4]. That is, we propose a *Redex within Coq* approach, where the pattern-matching engine is an algorithm verified against a formal specification of the semantics of each pattern. In addition, we propose to develop reasoning tools within Coq to help the user verify a model just in terms of the same concepts from the Redex formalism. The approach does not limit the kind of patterns that can be represented, nor the structure of the grammars that can be translated (beyond Redex’s own limitations). The downside is that, in order to help the user verify properties of a given model, we need to develop our own theory about patterns in Redex.

Summary of the Contributions

In this work we present a first step into the development of a tool to automate the translation of a Redex model into a semantically equivalent model in Coq, and to provide automation to the proof of essential properties of such models. The present work is heavily based on the model of Redex’s semantics developed by [4] (which we will denote as RedexK). Essential to RedexK are a specification of the process of matching between Redex patterns and terms, and an algorithmic interpretation of this specification.

The contributions of the present work are:

- We mechanize a modified version of RedexK in Coq. In the process, we develop a proof of termination for the matching algorithm, which enables its mechanization into Coq as a regular primitive recursion.
- We modify RedexK to prepare it for the future addition of features, like the Kleene’s closure operator, and the development of tactics to decide about properties of reduction semantics models.
- We prove soundness properties of the matching algorithm with respect to its specification.
- We verify the correspondence between our modified specification of matching and the original version presented in RedexK.

The reader is invited to download the accompanying source code from <https://github.com/Mallku2/redex2coq>.

The remainder of this paper is structured as follows: §2 presents a brief introduction to reduction semantics, as presented in Redex; §3 offers a general overview of our mechanization in Coq; §4 presents the main soundness results proved within our mechanization; §5 discuss about related work from the literature of the area; finally, §6 summarizes the results presented in this paper and discusses future venues of research enabled by this first iteration of our tool.


```

(define-language lambda
[e ::= x (e e) v] [v ::= (λ x e)] [x ::= variable-not-otherwise-mentioned] [E ::= hole (E e) (v E)]
[reduction-relation lambda #:domain e
[-> (in-hole E ((λ x e) v)) (in-hole E (substitute e x v)) beta_contraction]]
(define-metafunction lambda
fv : e -> (x ...)
[(fv x) (x)]
[(fv (e1 e2)) (x1 ... x2 ...) (where (x1 ...) (fv e1)) (where (x2 ...) (fv e2))]

```

■ **Figure 1** Definition of a language in Redex.

2 Redex

In this section, we present a brief introduction to Redex’s main concepts, limiting our attention to the concepts that are relevant to our tool in this first iteration of the development. As a running example, we show how to mechanize in Redex a fragment of λ -calculus with normal order call-by-value reduction. For a better introduction to these topics, the reader can consult [5, 7] and the original paper on which our mechanization is based [4].

Redex can be viewed as a particular implementation of Reduction Semantics with Evaluation Contexts (RS), in which semantical aspects of computations are described as relations over syntactic elements (terms) of the language.

As a simple introductory example, Figure 1 shows part of a specification for a call-by-value λ -calculus. The grammar of the language is defined with the first command, `define-language`. The language called `lambda` contains non-terminals `e` (representing any λ -term), `v` (values; in this case only λ -abstractions), `x` (variables; defined with pattern `variable-not-otherwise-mentioned`, meaning the symbols that are not used as literals elsewhere in the language) and `E` (evaluation contexts, to be explained below). The right-hand side of the productions of each non-terminal are shown to the right of the `::=` symbol.

The productions of non-terminal `E` indicate that an evaluation context could be a single hole, or a context of the form $E' e$, where E' is another evaluation context; or a context of the form $v E'$. Note that the consequence of this definition is that we are imposing normal-order reduction.

The reduction relation is defined with the keyword `reduction-relation`. It defines a relation between terms (`e`), from the previously defined `lambda` language, consisting of a single contraction, `beta_contraction`. This rule explains two things: how β -contractions are done; and the order in which those contractions can occur, effectively imposing the order of evaluation. The rule states that if a term can be decomposed into context `E` and an abstraction application $((\lambda x e) v)$ (pattern `(in-hole E ((λ x e) v))`), then, the original term reduces to the phrase resulting from plugging the result of substituting `x` by `v` in `e` into the context `E` (pattern `(in-hole E (substitute e x v))`).

As an example, consider the term $((\lambda w w) (\lambda y y)) (\lambda z z)$. In order to match the left-hand side of the rule, it decomposes the term into context $E = \text{hole } (\lambda z z)$, matching `x` with `w`, `e` with `w`, and `v` with $(\lambda y y)$. The result is the term $(\lambda y y) (\lambda z z)$.

We won’t delve into the details of the `substitute` meta-function, but it will be useful to explain one of its components: the list of *free variables* of a term, `fv`, partially shown in Figure 1. This meta-function is defined using the `define-metafunction` keyword. The signature of the function, `fv : e → (x ...)`, states that `fv` receives a λ -term, and returns a list of 0 or more variables (pattern `x ...`, to be explained below). After the signature, we have 2 equations explaining which are the free variables: in a term that is a single variable `x` or an application $e_1 e_2$. For reasons of space, we do not show equations referring to the cases where the term under consideration is a λ -abstraction.


```

Inductive term := lit_term : lit → term | list_term_c : list_term → term
                | ctxtxt_term : ctxtxt → term
with list_term := nil_term_c : list_term | cons_term_c : term → list_term → list_term
with ctxtxt := hole_ctxtxt_c : ctxtxt | list_ctxtxt_c : list_ctxtxt → ctxtxt
with list_ctxtxt := hd_ctxtxt : ctxtxt → list_term → list_ctxtxt
                | tail_ctxtxt : term → list_ctxtxt → list_ctxtxt.

```

■ **Figure 2** Language of terms.

The pattern $p \dots$ is called the *Kleene’s closure* of a pattern p , and expresses the idea of “zero or more terms” that match a given pattern p . For example, the first **where** clause of the second equation imposes a condition that holds only when the expression $\text{fv } e_1$ matches the pattern $x_1 \dots$, meaning that $\text{fv } e_1$ must evaluate to a list of 0 or more variables. Redex binds that list with $x_1 \dots$, and we can use this pattern to refer to this list. In particular, in this case we return $x_1 \dots$ followed by the variables resulting from evaluating $\text{fv } e_2$ (that is, $x_2 \dots$). As a last comment, it is possible to express context-dependent restrictions by using specific indexes: for example, pattern $(x_1 x_1)$ only matches a list of two equal variables; and pattern $(x_! x_!)$ only matches a list of two different variables.

3 Expressing Redex in Coq

In this section, we introduce the main ideas behind our implementation in Coq. Later, in §4, we describe the main soundness properties that we mechanized.

Coq’s literals and constructions will be presented with Coq’s **concrete syntax**, using listings or embedded in the text itself. Elements belonging to our meta-language (for example, some variables quantified over terms or patterns) will be presented with usual Latex’s math fonts. Further notation will be introduced when needed.

3.1 Language of Terms and Patterns

We begin the presentation by introducing our mechanized version of the language of terms and patterns. We ask for some reasonable decidability properties about the language that we use to describe a given reduction semantics model. These standard properties will be useful to develop our mechanization in its present version, and more so in the prospective future of the development.

3.1.1 Terms

The module type **Symbols** describes abstractly the atomic elements of the language of terms and patterns: literals (**lit**), non-terminals (**nonterm**), and *pattern variables* **var**, which also play the role of sub-indexes in the patterns. We require that these types are also instances of the **stdpp**’s typeclass **EqDecision** [18]. This encompasses showing that definitional equality between atomic elements is decidable. Details can be found in file **patterns_terms.v**.

In RedexK, terms are classified according to their structure, or if they act as a context or not. According to their structure, terms are classified as atomic literals or with a binary-tree structure. In our case, we will generalize the notion of “terms with structure”. One of the most prominent features absent in RedexK is the Kleene’s closure operator, which matches (or describes) lists of zero or more terms. In order to be able to include this feature in a future iteration of our model, we begin by generalizing the notion of structured terms. We

```

Inductive pat := lit_pat : lit → pat | hole_pat : pat
| list_pat_c : list_pat → pat | name_pat : var → pat → pat
| nt_pat : nonterm → pat | inhole_pat : pat → pat → pat
with list_pat := nil_pat_c : list_pat | cons_pat_c : pat → list_pat → list_pat.

```

■ **Figure 3** Language of patterns.

will allow them to be lists of 0 or more terms. Non-empty lists can also be considered as binary trees, but where the right sub-tree of a given node is always a list. We will enforce that shape through types.

The language of terms is presented in Figure 2. A term consisting of a literal is built with constructor `lit_term`, while structured terms are captured and enforced through a type, `list_term`. Structured terms can be an empty list, built with `nil_term_c` (which would be denoted simply as `()` in Redex), or a list with one term as its head, and some list as its tail, using constructor `cons_term_c`. For example, a Redex pattern like `(x x)`, for some literal `x`, would be built as: `cons_term_c (lit_term x) (cons_term_c (lit_term x) nil_term_c)`. Finally, we define an injection into terms, `list_term_c`.

The other kind of terms considered in RedexK are contexts. Contexts include information about where to find the hole, to help the algorithms of decomposition and plugging. That information consists in a path from the root of the term (seen as a tree) to the leaf that contains the hole. To that end, RedexK defines a notion of context that, if it is not just a single hole, contains a *tag* indicating where to look for the hole: either into the left or the right sub-tree of the context. We preserve the same idea, adapted to our presentation of structured terms.

We introduce the type `ctxt`, to represent and enforce through types the notion of contexts. These contexts can be just a single hole (`hole_ctxt_c`; denoted with the pattern `hole` in Redex, as shown in §2) or a list of terms with some position marked with a hole. In order to guarantee the presence of a hole into this last kind of contexts, we introduce the type `list_ctxt`. These contexts can point into the first position of a given list (`hd_ctxt`; like in `(hole (λ y y))`) or the tail (`tail_ctxt`; like in `((λ w w) hole)`). Finally, we have the injections from `list_ctxt` into `ctxt` (`list_ctxt_c`), and from `ctxt` into `term` (`ctxt_term`). These injections, naturally, are used later as coercions.

3.1.2 Patterns

As mentioned in §2, Redex offers a language of patterns with enough expressive power to state context-dependent restrictions. We mechanize the same language of patterns as presented in RedexK, with the required change to accommodate our generalization done to structured terms, as explained in the previous sub-section. The language of patterns is presented in Figure 3.

Pattern `lit_pat` *l* matches only a single literal *l*. Pattern `hole_pat` matches a context that is just a single hole. In order to describe the new category of structured terms that we presented in the previous subsection, we add a new category of patterns enforced through type `list_pat`. From this category of patterns, pattern `nil_pat_c` matches a list of 0 terms, while pattern `cons_pat_c` *p_{hd}* *p_{tl}* matches a list of terms, whose first term matches pattern *p_{hd}*, and whose tail matches the pattern *p_{tl}*. Finally, we have an injection from this category of patterns into the type `pat`: `list_pat_c`.

Context-dependent restrictions are imposed through pattern `name_pat x p`. This pattern matches a term t that, in turn, must match pattern p . As a result, the pattern `name_pat x p` introduces a context-dependent restriction in the form of a *binding*, that assigns *pattern variable* x to term t . Data-structures to keep track of this information will be introduced later, but for the moment, just consider that during matching some structures are used to keep track of all of this context-dependent restrictions that have the form of a binding between a pattern variable and a term. If, at the moment of introducing the binding to x , there exists another binding for the same variable but with respect to a term different than t , the whole matching fails. Note that this semantics accounts for the behavior of the pattern `(x_1 x_1)`, mentioned in §2. Also, a pattern like `(x!_ x!_)`, also mentioned in §2, could be described in terms of similar concepts, though it is currently not supported in RedexK nor within our mechanization.

Pattern `nt_pat e` matches a term t , if there exists a production from non-terminal e , whose right-hand-side is a pattern p that matches term t .

Finally, pattern `inhole_pat p_c p_h` matches some term t , if t can be decomposed between some context C , that matches pattern p_c , and some term t' , that matches pattern p_h . It should be possible to plug t' into context C , recovering the original term t . Note that the information contained in the tag of each kind of non-empty context, that indicates where to find the hole, helps in this process: at each step the process looks, either, into the head of the context or into its tail.

3.1.3 Decidability of predicates about terms and patterns

We want to put particular emphasis on the development of tools to recognize the decidability of predicates about terms and patterns. This could serve as a good foundation for the future development of tactics to help the user automate as much as possible the process of proving arbitrary statements about the user's reduction semantics models.

As a natural consequence of our first assumptions about the atomic elements of the languages of terms and patterns, presented in §3.1.1, we can also prove decidability results about definitional equalities among terms and patterns. Another straightforward consequence involves the decidability of definitional equalities between values of the many data-structures involved in the process of matching. Future efforts will be put in developing further this minimal theory about decidability (see §6).

3.1.4 Grammars

The notion of grammar in Redex, as presented in §2, is modeled in RedexK as a finite mapping between non-terminals and sets of patterns. Our intention is not to force some particular representation for grammars, beyond the previous description. As a first step, we axiomatize some assumptions about grammars through a module type. We begin by defining a production of the grammar, simply, as a pair inhabiting `nonterm * pat`, and we define a `productions` type as a list of type production. We also ask for the existence of computational type `grammar`, a constructor for grammars (inhabiting `productions → grammar`), the possibility of testing *membership* of a production with respect to a grammar, and to be possible to *remove* a production from a grammar (`remove_prod`). We ask for some notion of *length* of grammars, and that `remove_prod` actually affects that length in the expected way. This will be useful to guarantee the termination property of the matching algorithm (see §3.2.1). Finally, we ask for some reasonable decidability properties for these types and operations: decidability of definitional equalities among values of the previous types, and, naturally, for the testing of membership of a production with respect to a given grammar.

Abstracting these previous types and properties in a module type (`Grammar`), could serve in the future when developing further our theory of decidability for the notion of RS implemented in Redex. As a simple example, separating the type `productions` from the actual definition of the type `grammar`, allows for the encapsulation of properties in the type `grammar` itself, that specifies something about the inhabitants of `productions`. Some decidability results depend on a grammar whose productions are restricted in some particular way.¹

For this first iteration, we provide an instantiation of the previous module type with a grammar implemented using a list of productions: module `GrammarLists` from `grammar.v`. Here, the type `grammar` does not impose new properties over the inhabitants of type `productions`. We also provide a minimal theory to reason about *grammars as lists*, that helps in proving the required termination and soundness properties of the matching algorithm. This is required since our previous axiomatization of grammars, through module type `Grammar`, is not strong enough to prove every desired property of our algorithm. A goal for a next iteration would be to take advantage of the experience with this development, and strengthen our axiomatization of grammars.

3.2 Matching and Decomposition

The first challenge we encountered when trying to mechanize RedexK, was finding a primitive recursive algorithm to express matching and decomposition. The original algorithm from RedexK is not a primitive recursion, for reasons that will be clear below. However, the theory developed in the paper to check the soundness of this algorithm and to characterize the inputs over which it converges to a result, helped us to recapture the matching and decomposition process as a *well-founded recursion*.

3.2.1 Well-founded Relation Over the Domain of Matching/Decomposition

In Coq, a well-founded recursion is presented as a primitive recursion over the evidence of *accessibility* of a given element (from the domain of the well-founded recursion), with respect to a given *well-founded relation* R . That is, it is a primitive recursion over the proof of a statement that asserts that, from a given actual parameter x over which we are evaluating a function, there is only a finite quantity of elements which are *smaller* than x , according to relation R . These smaller elements are the ones over which the function can be evaluated recursively.

The actual steps of matching/decomposition will be presented in detail below. But, for the moment, in pursuing a well-founded recursive definition for the matching/decomposition process, let us observe that, for a given grammar G , pattern p and term t , the matching/decomposition of t with p involves, either:

1. Steps where the input term t is *decomposed* or *consumed*.
2. Steps where there is no input consumption, but, either:
 - a. The pattern p is decomposed or consumed.
 - b. The productions of the grammar G are considered, searching for a suitable pattern that allows matching to proceed.

¹ For example, while the general language intersection problem for context-free grammars (CFG) is non-decidable, the intersection problem between a regular CFG and a non-recursive CFG is decidable [9].

Step 1 corresponds, for example, to the case where t is a list of terms of the form `cons_term_c` t_{hd} t_{tl} , and p is a list of patterns of the form `cons_pat_c` p_{hd} p_{tl} . Here, the root of each tree (t and p) match, and the next step involves checking if t_{hd} matches pattern p_{hd} , and if t_{tl} matches p_{tl} .

Step 2a corresponds, for example, to the case where pattern p has the form `name_pat` x p' : as described in §3.1.2, the next step in matching/decomposition involves checking if pattern p' matches t . Here, the step does not involve consumption of input term t , but it does involve a recursive call to matching/decomposition over a proper sub-pattern of p .

Finally, step 2b corresponds to the case of pattern `nt_pat` n , which implies looking for productions of n in G that match t . Here, there is no reduction of terms and this process does not necessarily imply the reduction of patterns.

If not because for the pattern `nt_pat`, it could be easily argued that the process previously described is indeed an algorithm.² Now, if we do take into account `nt_pat` patterns, termination in the general case no longer holds. In particular, non-termination can be observed with a *left-recursive* grammar G and a given non-terminal n that witnesses the left-recursion of G . Matching pattern `nt_pat` n , following the described process, could get stuck repeating the step of searching into the productions of n , without any consumption of input: from pattern `nt_pat` n we could reach to the same pattern `nt_pat` n .

The previous problem with left-recursion is described in [4]. There, the property of left-recursion is captured by providing a relation \rightarrow_G that order patterns as they appear during the previously described phase of the matching process when the input term is not being consumed, but there is a decomposition of a pattern and/or searching into the grammar, looking for a proper production to continue the matching. Then, a left-recursive grammar would make the chains of the previous relation to contain a repeated pattern.

Then, if, for a non-left-recursive grammar G it is the case that $p \not\rightarrow_G^+ p$ for any pattern p (where \rightarrow_G^+ is the transitive closure of \rightarrow_G), it must be the case that also `nt_pat` $n \not\rightarrow_G^+ \text{nt_pat } n$, for a non-terminal n from G . This means that, when searching for productions of n in G , and as long as the matching/decomposition is in the stage captured by \rightarrow_G , it should be possible to *discard* the productions from the grammar G being tested.

The previous observation helps us argue that, provided that G is non-left-recursive, when the matching process enters the stage of non-consumption of input, this phase will eventually finalize: either, the pattern under consideration is totally decomposed and/or we run out of productions from G . In what follows, we will assume *only* non-left-recursive grammars. This does not impose a limitation over our model of Redex, since it only allows such kind of grammars.

We will exploit the previous to build a well-founded relation over the domain of our matching/decomposition function. The technique that we will use will consist in, first, modeling each phase in isolation through a particular relation. There will be a relation $\langle_{\text{t}} : \text{term} \rightarrow \text{term} \rightarrow \text{Prop}$ explaining what happens to the input when it is being consumed, and a relation $\langle_{\text{p} \times \text{g}} : \text{pat} \times \text{grammar} \rightarrow \text{pat} \times \text{grammar} \rightarrow \text{Prop}$, explaining what happens to the pattern and the grammar when there is no consumption of input. We will also prove the well-foundedness of each relation. The final well-founded relation for the matching/decomposition function will be the *lexicographic product* of the previous relations, a well-known method to build new well-founded relations out of other such relations [10]. We will parameterize this relation by the original grammar, to be able to recover the original productions when needed (see §3.2.4 for details). For a given grammar g , we will denote this last relation with $\langle_{\text{t} \times \text{p} \times \text{g}}^g$. Note that its type will be $\text{term} \times \text{pat} \times \text{grammar} \rightarrow \text{term} \times \text{pat} \times \text{grammar} \rightarrow \text{Prop}$.

² *Algorithm* as an effective procedure that also terminates on every input.

$$\begin{array}{l}
(p_c, G) <_{p \times g} (\text{inhole_pat } p_c \ p_h, G) \qquad (p_h, G) <_{p \times g} (\text{inhole_pat } p_c \ p_h, G) \\
(p, G) <_{p \times g} (\text{name_pat } \times \ p, G) \qquad \frac{p \in G(n) \quad G' = G \setminus (n, p)}{(p, G') <_{p \times g} (\text{nt_pat } n, G)}
\end{array}$$

■ **Figure 4** Consumption of pattern and productions.

For a tuple (t, p, G) to be related with another *smaller* tuple (t', p', G') , according to $<_{t \times p \times g}^g$, it must happen that $t' <_t t \vee (t' = t \wedge (p', G') <_{p \times g} (p, G))$. This expresses the situations where there is actual progress in the matching/decomposition algorithm towards a result: either there is consumption of input or the phase of production searching and decomposition of the pattern progresses towards its completion. Note that this definition shows that the lexicographic product is a more general relation, that contains chains of tuples that do not necessarily model what happens during matching and decomposition: if $t' <_t t$, then $(t', p', G') <_{t \times p \times g}^g (t, p, G)$, for some grammar g , regardless of what (p', G') and (p, G) actually are. Later, when presenting the relations that form this lexicographic product, we will also specify which are the actual chains that we will consider when modeling the process of matching and decomposition. We will refer to this last kind of chains as the *chains of interest*.

3.2.2 Input consumption

We define the relation $<_t$ to be exactly $<_{\text{subt}}$, where $<_{\text{subt}}$ will denote the relation `subterm_rel` : `term` \rightarrow `term` \rightarrow `Prop`, that links a term with each of its sub-terms. This describes an order that coincides with that in which the input is consumed, for the actual specification of matching and decomposition. This does not avoid for more exotic patterns, that could be introduced in the future, to have a different behavior on input consumption. Hence, the distinction between what constitutes a relation like $<_t$ and what simply is $<_{\text{subt}}$.

3.2.3 Pattern and production consumption

The specification of $<_{p \times g}$, shown in Figure 4, matches the cases 2a and 2b described in §3.2.1. Recall that, in this case, the algorithm entered a phase where the pattern is being decomposed or productions from some non-terminal are being tested, to see if matching/decomposition can continue. Matching a term t with a pattern of the form `inhole_pat` $p_c \ p_h$, means trying to decompose the term between some context that matches pattern p_c , and some sub-term of t that matches pattern p_h . In doing so, the first step involves a decomposition process (to be specified later in §3.2.5), that begins working over the whole term t , and with respect to just the sub-pattern p_c . Hence, this step does not involve input consumption, but it does involve considering a reduced pattern: p_c . We just capture this simple fact through $<_{p \times g}$, by stating that $(p_c, G) <_{p \times g} (\text{inhole_pat } p_c \ p_h, G)$ holds, for any grammar G . Note that we preserve the grammar.

In the particular case that p_c matches `hole_ctxt_c`, then there is no actual decomposition of the term t . This means that, when looking for said sub-term of t that matches pattern p_h , we will still be considering the whole input term t . Again, we just capture this simple fact by stating that $(p_h, G) <_{p \times g} (\text{inhole_pat } p_c \ p_h, G)$ holds, for any grammar G .

$$\begin{array}{c}
\frac{\rho \in G'(n) \quad G \vdash t : \rho_{G' \setminus (n, \rho)} \mid b}{G \vdash t : (\text{nt_pat } n)_{G'} \mid \emptyset} \\
\\
\frac{G \vdash t_{hd} : (\rho_{hd})_G \mid b_{hd} \quad G \vdash t_{tl} : (\rho_{tl})_G \mid b_{tl}}{G \vdash \text{cons_term_c } t_{hd} \ t_{tl} : (\text{cons_pat_c } \rho_{hd} \ \rho_{tl})_{G'} \mid b_{hd} \sqcup b_{tl}} \\
\\
\frac{G \vdash t = C[[t_h]] : (\rho_c)_{G'} \mid b_c \quad t_h <_{\text{subt}} t \quad G \vdash t_h : (\rho_h)_G \mid b_h}{G \vdash t : (\text{inhole_pat } \rho_c \ \rho_h)_{G'} \mid b_c \sqcup b_h}
\end{array}$$

■ **Figure 5** Generalized specification of matching.

The case for the pattern `name_pat` $x \ \rho$ can be explained on the same basis as with the previous cases.

Finally, the last case refers to the pattern `nt_pat` n : it involves considering each production of non-terminal n in G (which are denoted as $G(n)$). Here it is assumed that G contains the correct set of productions that remain to be tested (an invariant property about G through our algorithm). Then, we continue the process considering a grammar G' that contains every production from G , except for (n, ρ) : the already considered production of non-terminal n with right-hand-side ρ . We denote it stating that G' equals the expression $G \setminus (n, \rho)$.

3.2.4 Specification of matching

We now explain our specification for matching and decomposition, which is a slight generalization from that of RedexK [4]. In the original specification, the judgment about matching has the form $G \vdash t : \rho \mid b$, stating that term t matches pattern ρ , under the productions from grammar G , producing the bindings b (which could be an empty set of bindings, denoted with \emptyset). A seemingly obvious fact is that the non-terminals that may appear on pattern ρ will be interpreted in terms of the productions from G . In our presentation, we relax this assumption, and allow the non-terminals to be interpreted in terms of some arbitrary grammar G' , which in practice will be a subset of G .

Therefore, our judgment is of the form $G \vdash t : \rho_{G'} \mid b$, with the particular difference that, *initially*, we interpret the non-terminals from ρ with grammar G' . Only when input consumption begins, we restore the original grammar G . Figure 5 presents a simplified fragment of our formal system. Following a top-down order, the first rule applies when a term t matches a pattern `nt_pat` n , when the non-terminals of this pattern (in this case, just n) are *initially* interpreted in terms of the productions of G' : then, that matching is successful if there exists some $\rho \in G'(n)$, such that t matches ρ , when its non-terminals are *initially* interpreted under the productions from the grammar $G' \setminus (n, \rho)$. Recall that this means that this last grammar will be used as long as there is no input consumption, or there is no other occurrence of a pattern `nt_pat`. Again, we are following the chains from $<_{\text{p} \times \text{g}}$. Also, the non-left-recursivity of the grammars being considered guarantees that this replacement of the grammars is semantics-preserving: we will not need another production from n , as long as there is no input consumption. Finally, note that this match does not produce bindings.

$$\begin{array}{c}
\frac{G \vdash t_{hd} = C[[t'_{hd}]] : (\rho_{hd})_G \mid b_{hd} \quad G \vdash t_{tl} : (\rho_{tl})_G \mid b_{tl}}{G \vdash \text{cons_term_c } t_{hd} \ t_{tl} = (\text{hd_contxt } C \ t_{tl})[[t'_{hd}]] : (\text{cons_pat_c } \rho_{hd} \ \rho_{tl})_{G'} \mid b_{hd} \sqcup b_{tl}} \\
\\
\frac{G \vdash t = C_c[[t_c]] : (\rho_c)_{G'} \mid b_c \quad t_c <_{\text{subt}} t \quad G \vdash t_c = C_h[[t_h]] : (\rho_h)_G \mid b_h}{G \vdash t = (C_c ++ C_h)[[t_h]] : (\text{inhole_pat } \rho_c \ \rho_h)_{G'} \mid b_c \sqcup b_h}
\end{array}$$

■ **Figure 6** Generalized specification of decomposition.

The second rule can be understood in terms of the previously introduced concepts. Note that, for each recursive proof of matching over sub-terms and sub-patterns, we *re-install* the original grammar G . We denote with \sqcup the union of bindings, which is undefined if the same name is bound to different terms.

The last case in Figure 5 refers to the matching of a term t with a pattern of the form `inhole_pat` $\rho_c \ \rho_h$. This operation is successful when we can decompose term t between some context that matches pattern ρ_c , and some sub-term, that matches pattern ρ_h . In order to fully formalize what this matching means, we need to explain what *decomposition* means. RedexK specifies this notion through another formal system, whose adaptation to our work we present in the following sub-section. The original system allows us to build proofs for judgments of the form $G \vdash t = C[[t']] : \rho \mid b$, meaning that we can decompose term t , between some context C , that matches pattern ρ , and some sub-term t' . The decomposition produces bindings b , and the non-terminals from pattern ρ are interpreted through the productions present in grammar G . In our case, we modify this judgment by generalizing it in the same way done for the matching judgment: $G \vdash t = C[[t']] : \rho_{G'} \mid b$, including the possible interpretation of non-terminals in ρ , initially, using grammar G' .

Returning to the case about `inhole_pat` patterns in Figure 5, note that our intention is to distinguish the case where the decomposition step actually consumes some portion from t (shown in the rule), from the case where it does not (not shown in Figure 5). The first situation (described in the rule for `inhole_pat`) means that context C is not simply a hole, and t_h is an actual proper sub-term of t : *i.e.*, $t_h <_{\text{subt}} t$. Also, note that the decomposition is proved interpreting (initially) the non-terminals from ρ_c with production from the arbitrary grammar G' ($(\rho_c)_{G'}$). And the proof of the matching between t_h and ρ_h is done interpreting the non-terminals of this last pattern with productions from the original grammar G ($(\rho_c)_G$). On the contrary, when the decomposition step does not consume some input (pattern ρ_c matches against a hole, and the resulting term t_h is exactly t), the proof of the matching between t_h and ρ_h is done considering the arbitrary grammar G' .

3.2.5 Specification of decomposition

The final part of the specification concerns the decomposition judgment required for the `inhole_pat` pattern. We already mentioned what it does and how it is generalized; we proceed to explain the relevant rules listed in Figure 6.

The first rule explains the decomposition of a list of terms `cons_term_c` $t_{hd} \ t_{tl}$, between a context that matches a list of patterns `cons_pat_c` $\rho_{hd} \ \rho_{tl}$, and some sub-term. In the particular case of the first rule, the hole of the resulting context is pointing to somewhere in the head of the list of terms. This information is indicated by the constructor of the resulting context: `hd_contxt` $C \ t_{tl}$, where C is some context that must match pattern ρ_{hd} ,

```

Definition binding := var * term.
Inductive decom_ev : term → Set :=
| empty_d_ev : forall (t : term), decom_ev t
| nonempty_d_ev : forall t (c : ctxt) sub,
  {sub = t ∧ c = hole_ctxt_c} + {subterm_rel sub t} → decom_ev t.
Inductive mtch_ev : term → Set :=
  mtch_pair : forall t, decom_ev t → list binding → mtch_ev t.

```

■ **Figure 7** Mechanization of decomposition and matching results.

as indicated in the premise of the inference rule. Note that the whole premise is stating that the decomposition occurs in the head of the list of terms (t_{hd}), and the resulting sub-term is t'_{hd} . Then, the side-condition from the inference rule states that the tail of the original input term, t_{tl} , must match the tail of the list of patterns p_{tl} . Finally, note that in the decomposition through sub-pattern p_{hd} , and the matching sub-pattern p_{tl} , the non-terminals of these patterns are interpreted in terms of productions from the original grammar, G .

With respect to the remaining rule, the case of the `inhole_pat` pattern, it handles the matching of pattern `inhole_pat` (`inhole_pat` p_c p_h) $p_{h'}$ with some term t . The semantics of this case involves a first step of decomposition of t between some context that matches sub-pattern `inhole_pat` p_c p_h , and some sub-term that matches sub-pattern $p_{h'}$. In the rule shown in Figure 6, we are describing what it means, in this situations, that first step of decomposing t in terms of a context that matches pattern `inhole_pat` p_c p_h . Since the whole pattern must match some context, it means that, both, p_c and p_h , are patterns describing contexts. Note that we distinguish the case where p_c produces an empty context, from the case where it does not (not shown in Figure 6). This distinction allows us to recognize whether we should interpret non-terminals from patterns through the original grammar G or the arbitrary grammar G' .

The last piece of complexity of the rule for the `inhole_pat` pattern resides in the actual context that results from the decomposition. Here, the authors of RedexK, expressed this context as the result of plugging one of the obtained contexts within the other, denoted with the expression $C_c ++ C_h$: this represents the context obtained by plugging context C_h within the hole of context C_c , following the information contained in the constructor of this last context to find its actual hole. For reasons of space we elide this definition, though it presents no surprises.

3.2.6 Matching and decomposition algorithm

We close this section presenting a simplified description of the matching and decomposition algorithm adapted for its mechanization in Coq. We remind the reader that this algorithm is just a modification of the one proposed for RedexK [4].

The previous specification of the algorithm cannot be used directly to derive an actual effective procedure to compute matching and decomposition. In particular, the rules for decomposition of lists of terms (second and third rules from Figure 6) do not suggest effective meanings to determine whether to decompose on the head, and match on the tail, or vice versa. To solve this issue and the complexity problem that could arise from trying to naively perform both kinds of decomposition simultaneously, the algorithm developed for RedexK performs matching and decomposition simultaneously, sharing intermediate results.

Supporting Data-Structures

In Figure 7 we show some of the implemented data-structures used to represent the results returned by RedexK's algorithm. The result of a matching/decomposition of a term t (with some given pattern) will be represented through a value of type `mtch_ev t`. Making the type dependent on t is done for soundness checking.

For reasons of brevity, when presenting the algorithm we will avoid the actual concrete syntax from our mechanization. A value of type `mtch_ev t` will be denoted as (d, b) , where d is a value of type `decom_ev t` (explained below), and b is a list of bindings (also shown in Figure 7). For a value of the list type `mtch_powset_ev t`, we will denote it decorating it with its dependence on the value t : $[(d, b), \dots]_t$

Values inhabiting type `decom_ev t` represent a decomposition of a given term t , between a context and a sub-term. We include in the value some evidence of the soundness of the decomposition: a sub-term $subt$ extracted in the decomposition is either t itself, or a proper sub-term of t .

Since a value of type `mtch_ev t` could represent a single match or a single decomposition, following [4] we distinguish an actual match using an empty decomposition `empty_d_ev t`. Otherwise, a decomposition is represented through the value `nonempty_d_ev t C subt ev`, for context C , sub-term $subt$ and soundness evidence ev . We denote such values as $(C, subt)_t^{ev}$.

Matching and Decomposition Algorithm as a Least-Fixed-Point

We capture the intended matching/decomposition algorithm as the least fixed-point of a *generator function* or *functional* of the following type:

```
forall (g1 : grammar) (tpg1 : (term * pat * grammar)),
  (forall tpg2 : (term * pat * grammar),
    matching_tuple_order g1 tpg2 tpg1 → list (mtch_ev (fst tpg2)))
  → list (mtch_ev (fst tpg1))
```

The family of generator functions M_{ev_gen} of this type is parameterized over grammars and tuples of terms and patterns. Also, these functions receive a candidate of matching/decomposition that they will *improve*: they will construct the result by optionally calling the candidate over tuples that are provably smaller than the given tuple $tpg1$, according to the well-founded order (`matching_tuple_order g1 tpg2 tpg1`, see §3.2.1). Hence, M_{ev_gen} will build a function that performs the matching indicated in $tpg1$, using, if necessary, a candidate function that performs matching for tuples smaller than $tpg1$.

Figure 8 shows 2 of the equations that capture M_{ev_gen} . The first equation explains the matching and/or decomposition of a list of terms (`cons thd ttl`) with a list of patterns (`cons phd ptl`). We describe by comprehension the list of results. Note that, to explain this case, we need to consider the approximation function M_{ap} that M_{ev_gen} receives as its last parameter. We begin by using M_{ap} to compute matching and decomposition for *smaller* tuples: $tp_{hd} = (t_{hd}, p_{hd}, g_1)$ and $tp_{tl} = (t_{tl}, p_{tl}, g_1)$. Note that, given that these tuples represent a matching/decomposition over a proper sub-term of the input term, we consider the original grammar g_1 (first parameter of M_{ev_gen}). In order to be able to fully evaluate M_{ap} , we need to build proofs lt_{hd} and lt_{tl} of type $tp_{hd} <_{t \times p \times g}^{g_1} tp_{cons}$ and $tp_{tl} <_{t \times p \times g}^{g_1} tp_{cons}$, respectively, where tp_{cons} is the original tuple over which we evaluate M_{ev_gen} . Then, for each value of type `mtch_ev thd` and `mtch_ev ttl` of the results obtained from evaluating M_{ap} , the algorithm queries if they are decompositions or not, and if it is possible to combine these results, using the helper function `select`.

$$\begin{aligned}
M_{\text{ev_gen}}(g_1, (t, p, g_2), M_{\text{ap}}) &= [(d, b) \mid d \in \text{select}(t_{hd}, d_{hd}, t_{tl}, d_{tl}, t, \text{sub}), \\
&\quad \text{sub} : \text{subterms } t \ t_{hd} \ t_{tl}, \quad b = b_{hd} \sqcup b_{tl}, \\
&\quad (d_{hd}, b_{hd})_{t_{hd}} \in M_{\text{ap}}(tp_{hd}, lt_{hd}), \quad (d_{tl}, b_{tl})_{t_{tl}} \in M_{\text{ap}}(tp_{tl}, lt_{tl}), \\
&\quad lt_{hd} : tp_{hd} <_{t \times p \times g}^{g_1} tp_{\text{cons}}, \quad lt_{tl} : tp_{tl} <_{t \times p \times g}^{g_1} tp_{\text{cons}}, \\
&\quad tp_{\text{cons}} = (t, p, g_2), \quad tp_{hd} = (t_{hd}, p_{hd}, g_1), \quad tp_{tl} = (t_{tl}, p_{tl}, g_1)]_t \\
&\quad \text{with } t = \mathbf{cons} \ t_{hd} \ t_{tl} \quad p = \mathbf{cons} \ p_{hd} \ p_{tl} \\
M_{\text{ev_gen}}(g_1, (t, p, g_2), M_{\text{ap}}) &= [(d, b) \mid d = \text{combine}(t, C, t_c, \text{ev}, d_h), \\
&\quad b = b_c \sqcup b_h, \quad (d_h, b_h)_{t_c} \in M_{\text{ap}}(tp_h, lt_h), \\
&\quad lt_h : tp_h <_{t \times p \times g}^{g_1} tp_{\text{in-hole}}, \quad tp_h = (t_c, p_h, g_h), \\
&\quad g_h \text{ according to Figure 5,} \\
&\quad ((C, t_c)^{\text{ev}}, b_c)_t \in M_{\text{ap}}(tp_c, lt_c), \quad lt_c : tp_c <_{t \times p \times g}^{g_1} tp_{\text{in-hole}}, \\
&\quad tp_{\text{in-hole}} = (t, p, g_2), \quad tp_c = (t, p_c, g_2)]_t \\
&\quad \text{with } p = \mathbf{in-hole} \ p_c \ p_h
\end{aligned}$$

■ **Figure 8** Generator function for the matching and decomposition algorithm.

The original `select` helper function from RedexK receives as parameters t_{hd} , d_{hd} , t_{tl} and d_{tl} . It analyses d_{hd} and d_{tl} : if none of them represent actual decompositions, then the whole operation will be considered just a matching of the original list of terms and `select` must build an *empty* decomposition of the proper type to represent this. If only d_{hd} is a decomposition, then the whole operation is interpreted as a decomposition of the original list of terms on the head of the list. In that case, `select` builds a value of type `decom_ev` ($\mathbf{cons} \ t_{hd} \ t_{tl}$).

The remaining equation, that of the **in-hole** pattern, can be understood on the same basis as the previous one, requiring only some explanation for the auxiliary function `combine`: it helps in deciding if the result is a decomposition against pattern **in-hole**, or if it is just a match against said pattern, depending on whether d_h is a decomposition or not.

Finally, we define the desired matching/decomposition algorithm, M_{ev} , as the least fixed-point of the previous generator function. For reasons of space, we do not show its definition, but it presents no surprises. The resulting implementation can be seen on file `./match_impl.v`.

3.3 Semantics for Context-Sensitive Reduction Rules

The last component of RedexK consists in a semantics for context-sensitive reduction rules, with which we define semantics relations in Redex. The proposed semantics makes use of the introduced notion of matching, to define a new formal system that explains what it means for a given term to be *reduced*, following a given semantics rule. We have mechanized the previous formal system, though, for reasons of space, we do not introduce it here in detail. The reader is invited to look at the mechanization of this formal system, in module `./reduction.v`.

```

Theorem completeness_Mev :  $\forall G1 G2 p t \text{sub\_t } b C,$ 
  (G1 |- t : p, G2 | b  $\rightarrow$  In (mch_pair t (empty_d_ev t) b) (M_ev G1 (t, (p, G2))))
 $\wedge$ 
  (G1 |- t1 = C [ t2 ] : p, G2 | b  $\rightarrow$   $\exists$  (ev_decom : {sub_t = t} + {subterm_rel sub_t t}),
    In (mch_pair t (nonempty_d_ev t C sub_t ev_decom) b) (M_ev G1 (t, (p, G2)))).

Theorem from_orig :  $\forall G t p b,$ 
  non_left_recursive_grammar  $\rightarrow$ 
  G |- t : p | b  $\rightarrow$  G |- t : p, G | b
with from_orig_decomp :  $\forall G C t1 t2 p b,$ 
  non_left_recursive_grammar  $\rightarrow$ 
  G |- t1 = C [ t2 ] : p | b  $\rightarrow$  G |- t1 = C [ t2 ] : p, G | b.

```

■ **Figure 9** The statement of completeness of M_{ev} and completeness of our formal systems, in Coq.

3.4 Extra Material

In the README.md file of the repository the interested reader will find the correspondence between the source code and this paper. Additionally, besides from the results shown here, we included a mechanization of a lambda-calculus with normal-order reduction similar to the one presented in §2. It serves mainly to showcase the actual capabilities of Redex that are mechanized in the present version of the tool, and how to invoke them to implement a reduction-semantics model. We note that the performance of our implementation of the matching/decomposition algorithm is subpar. In particular, the resources in time and space consumed for matching grow too fast to be able to test even some simple patterns. The amount of information built and carried within the algorithm to guarantee soundness properties could be playing some part, and it could be addressed by code extraction, or by the implementation within Coq of a simpler pattern-matching engine, in correspondence with the verified version. Though, it is an issue we plan to better study and tackle in a future iteration of the tool. We note that this is not a problem observed in Redex itself.

4 Soundness and Completeness of Matching

In the original paper of RedexK the authors prove the correspondence between the algorithm and its specification. In our mechanization we reproduced this result, for the least-fixed-point of $M_{ev_gen} g (t, p, g')$ and our extended definition of matching (§3.2.4). In what follows, $M_{ev} g (t, p, g')$ represents the least-fixed-point of $M_{ev_gen} g (t, p, g')$. Naturally, for a given grammar g , the original intention of matching and decomposition corresponds to $M_{ev} g (t, p, g)$. We show the statement of completeness of the algorithm in Figure 9. Note that we represent and manipulate results returned from M_{ev} through Coq's standard library implementation of lists. Also, the shape of the tuples of terms, patterns and grammars, is the result of the way in which we build our lexicographic product: the product between a relation with domain **term**, and a relation with domain **pat** \times **grammar**. Completeness can be proved by *rule induction* on the evidences of match and decomposition.

The converse, the soundness property, is not shown, but it is the expected converse of the completeness statement. The proof presents no surprises: since we have a well-founded recursion over the tuples from **term** \times **pat** \times **grammar**, we also have an induction principle to reason over them.

We also verified the correspondence between our specifications and the original formal systems from the paper. We can’t do it for the general case: we followed the proposal of the authors of RedexK, explained in §3.2, and only consider those grammars that are *non-left-recursive*. In Coq, we name this predicate `non_left_recursive_grammar` (see file `wf_rel.v`).

We show in Figure 9 the completeness result mapping our formal systems with the original ones from RedexK. Note the hypothesis `non_left_recursive_grammar`, which asks for every grammar to be non-left-recursive. We do not parameterize this predicate over a specific grammar, since, during the proofs, we may obtain several different grammars by removing productions from the original grammar, and we still need to show that these “intermediate” grammars are non-left-recursive. A more elegant solution could be, first, to prove that by removing a production from a non-left-recursive grammar, we still get a non-left-recursive grammar; and, second, to parameterize `non_left_recursive_grammar` over G . We left this as a future work.

For the converse, soundness, we need to restrict the result to those grammars G' (over which we begin interpreting the non-terminals) which are *smaller or equal* (`gleq`) to the original one G : that is, every production in G' is also in G (see `./verification/match_spec_equiv.v` for more details).

5 Related Work

Redex-Plus [19] is, to the best of our knowledge, the only tool proposed to export Redex models to proof assistants. The approach followed involves translating a given model, first, into an intermediate representation where some elements of the model are described through types. For example, non-terminals of a grammar are captured as types, and the right-hand-side of each production is captured as a constructor of the corresponding type. Having an intermediate representation of a Redex model allows Redex-Plus to export to several different targets: Agda, Coq, Beluga and SMT-LIB. It can handle definitions of languages, meta-functions and formal systems.

The downside of Redex-Plus approach is that it limits the scope of Redex patterns that can be supported, and restricts the structure of the grammars that are allowed. From its reference manual: “*In general, only patterns that can be represented in proof assistants are supported*”. In particular, it is not possible to have *overlapping* non-terminals: that is, different non-terminals that can generate the same phrases (for example, a syntactic category *value* as a sub-category of *terms*). The reason is that, since each non-terminal is represented through a type, a given “phrase” (value) cannot inhabit such two different non-terminals (types).

In our case, every pattern inhabits the same “pattern” type, and everything about their semantics (pattern matching) is mechanized within Coq itself. This approach does not limit the kind of Redex patterns or structure of grammars that can be represented within Coq. Nonetheless, the technique employed by Redex-Plus is an interesting take at representing elements of a Redex model into a proof assistant, that should be able to better leverage the type system of the target proof assistant.

Another difference with Redex-Plus is that its translation and representation of patterns in the target proof assistant involves an informal Racket implementation. The implementation is not verified against some formal specification of the semantics of patterns. In [19] it is argued that the patterns supported in the current version have a well-understood semantics, and that it should be possible to accurately translate them into proof assistants. This seems

to hold for the subset of patterns currently supported by Redex-Plus, though it remains to be seen what would happen when more complex features are added. In our case, we follow the concerns raised in [4] and we offer a pattern-matching algorithm mechanically verified against its specification. And, since we implement Redex itself within Coq, the translation between a Redex pattern and its representation within Coq is a straightforward process.

CoLoR [2] is a mechanization in Coq of the theory of well-founded rewriting relations over the set of first-order terms, applied to the automatic verification of termination certificates. It presents a formalization of several fundamental concepts of rewriting theory, and the mechanization of several results and techniques used by termination provers. Its notion of terms includes first-order terms with symbols of fixed and varyadic arity, strings, and simply typed lambda terms. CoLoR does not implement a language of patterns offering support for context-sensitive restrictions, something that is ubiquitous in a Redex mechanization. Also, Redex is not focused just on well-founded rewriting relations.

Sieczkowski et. al present in [13] a verified implementation in Coq of the technique of *refocusing*, with which it is possible to extract abstract machines from a specification of a reduction semantics that satisfies certain characteristics. In order to characterize a reduction semantics that can be *automatically refocused*, the authors provide an axiomatization capturing the sufficient conditions. Hence, the focus is put in allowing the representation of a certain class of reduction semantics rather than allowing for the mechanization of arbitrary models, as is the case with Redex. Nonetheless, future development of our tool could take advantage of this library, since testing of Redex's models that are proved to be deterministic could make use of an optimization as refocusing, to extract interpreters that run efficiently in comparison with the expensive computation model of reduction semantics.

Matching logic is a formalism used to specify logical systems and their properties. It is mechanized in Coq in [1], including its syntax, semantics, formal system and the corresponding soundness result. At its heart, matching logic has a notion of patterns and pattern matching. Redex could be explained as a matching logic, with formulas that represent Redex's patterns to capture languages and relations, and whose model refer to the terms (or structures containing terms) that match against these patterns. While this representation could be of interest for the purpose of studying the underlying semantics of Redex, this is not satisfactory for the purpose of providing users with a direct explanation in Coq of their mechanization in Redex.

6 Conclusion

We adapted RedexK [4] to be able to mechanize it into Coq. In particular, we obtained a primitive recursive expression of its matching algorithm; we introduced modifications to its language of terms and patterns, to better adapt it to the future inclusion of features of Redex absent in RedexK; we reproduced the soundness results shown in [4], but adapted to our mechanization, while also verifying the expected correspondence between our adapted formal systems, that capture matching and decomposition, and the originals from the cited work.

A natural next step in our development could consist in the addition of automatic routines to transpile a Redex model into an equivalent model in Coq. In order to be practical, we also must extend the language with capabilities of Redex absent in RedexK.

Finally, the user can specify properties expressed using judgments about matching, or in terms of the results of the matching algorithm. To prove these properties, the user have some results at hand (for example, soundness and completeness of the algorithm, and of our formal specification of matching with regard to the original specification), but a richer theory is in order.

References

- 1 Péter Berezky, Xiaohong Chen, Dániel Horpácsi, Lucas Peña, and Jan Tušil. Mechanizing matching logic in coq. In Vlad Rusu, editor, *Proceedings of the Sixth Working Formal Methods Symposium*, "Al. I. Cuza University", Iasi, Romania, 19-20 September, 2022, volume 369 of *Electronic Proceedings in Theoretical Computer Science*, pages 17–36. Open Publishing Association, 2022. doi:10.4204/EPTCS.369.2.
- 2 Frédéric Blanqui and Adam Koprowski. Color: a coq library on well-founded rewrite relations and its application to the automated verification of termination certificates. *Mathematical Structures in Computer Science*, 21(4):827–859, 2011. doi:10.1017/S0960129511000120.
- 3 Brown PLT Group. Mechanized lambdajcs. <https://blog.brownplt.org/2012/06/04/lambdajcs-coq.html>, 2012.
- 4 Steven Jaconette Casey Klein, Jay McCarthy and Robert Bruce Findler. A semantics for context-sensitive reduction semantics. In *APLAS'11*, 2011.
- 5 M. Felleisen, R. B. Findler, and M. Flatt. *Semantics Engineering with PLT Redex*. The MIT Press, 2009.
- 6 A. Guha, C. Saftoiu, and S. Krishnamurthi. The essence of JavaScript. In *ECOOP '10*, 2010.
- 7 Casey Klein, John Clements, Christos Dimoulas, Carl Eastlund, Matthias Felleisen, Matthew Flatt, Jay A. McCarthy, Jon Raskind, Sam Tobin-Hochstadt, and Robert Bruce Findler. Run your research: On the effectiveness of lightweight mechanization. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 285–296, New York, NY, USA, 2012. ACM. doi:10.1145/2103656.2103691.
- 8 Jacob Matthews and Robert Bruce Findler. An operational semantics for scheme. *Journal of Functional Programming*, 2007.
- 9 Mark-Jan Nederhof and Giorgio Satta. The language intersection problem for non-recursive context-free grammars. *Inf. Comput.*, 192(2):172–184, August 2004. doi:10.1016/j.ic.2004.03.004.
- 10 Lawrence C. Paulson. Constructing recursion operators in intuitionistic type theory. *J. Symb. Comput.*, 2(4):325–355, December 1986.
- 11 J. G. Politz, M. J. Carroll, B. S. Lerner, J. Pombrio, and S. Krishnamurthi. A tested semantics for getters, setters, and eval in JavaScript. In *DLS '12*, 2012.
- 12 J. G. Politz, A. Martinez, M. Milano, S. Warren, D. Patterson, J. Li, A. Chitipothu, and S. Krishnamurthi. Python: The full monty: A tested semantics for the Python programming language. In *OOPSLA '13*, 2013.
- 13 Filip Sieczkowski, Małgorzata Biernacka, and Dariusz Biernacki. Automating derivations of abstract machines from reduction semantics: A generic formalization of refocusing in coq. In *Proceedings of the 22nd International Conference on Implementation and Application of Functional Languages*, IFL'10, pages 72–88, Berlin, Heidelberg, 2010. Springer-Verlag.
- 14 M. Soldevila, B. Ziliani, and B. Silvestre. From specification to testing: Semantics engineering for lua 5.2. *Journal of Automated Reasoning*, 2022.
- 15 Mallku Soldevila, Rodrigo Ribeiro, and Beta Ziliani. redex2coq. Software, CONICET (Argentina), swHId: swH:1:dir:eea74f34ec4a10a6e7315b1d5d3f89327bce18a5 (visited on 2024-08-20). URL: <https://github.com/Mallku2/redex2coq>.
- 16 Mallku Soldevila, Beta Ziliani, and Daniel Fridlender. Understanding Lua's garbage collection: Towards a formalized static analyzer. In *Proceedings of the 22nd International Symposium on Principles and Practice of Declarative Programming*, PPDP 2020, 2020.
- 17 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 Dynamic Languages Symposium*, DLS 2017, 2017.
- 18 The Coq-std++ Team. An extended “standard library” for Coq, 2020. Available online at <https://gitlab.mpi-sws.org/iris/stdpp>.
- 19 Junfeng Xu. *Redex-plus: a metanotation for programming languages*. PhD thesis, University of British Columbia, 2023. doi:10.14288/1.0435516.

Formal Verification of the Empty Hexagon Number

Bernardo Subercaseaux ✉ 

Carnegie Mellon University, Pittsburgh, PA, USA

Wojciech Nawrocki ✉ 

Carnegie Mellon University, Pittsburgh, PA, USA

James Gallicchio ✉ 

Carnegie Mellon University, Pittsburgh, PA, USA

Cayden Codel ✉ 

Carnegie Mellon University, Pittsburgh, PA, USA

Mario Carneiro ✉ 

Carnegie Mellon University, Pittsburgh, PA, USA

Marijn J. H. Heule ✉ 

Carnegie Mellon University, Pittsburgh, PA, USA

Abstract

A recent breakthrough in computer-assisted mathematics showed that every set of 30 points in the plane in general position (i.e., no three points on a common line) contains an empty convex hexagon. Heule and Scheucher solved this problem with a combination of geometric insights and automated reasoning techniques by constructing CNF formulas ϕ_n , with $O(n^4)$ clauses, such that if ϕ_n is unsatisfiable then every set of n points in general position must contain an empty convex hexagon. An unsatisfiability proof for $n = 30$ was then found with a SAT solver using 17 300 CPU hours of parallel computation. In this paper, we formalize and verify this result in the Lean theorem prover. Our formalization covers ideas in discrete computational geometry and SAT encoding techniques by introducing a framework that connects geometric objects to propositional assignments. We see this as a key step towards the formal verification of other SAT-based results in geometry, since the abstractions we use have been successfully applied to similar problems. Overall, we hope that our work sets a new standard for the verification of geometry problems relying on extensive computation, and that it increases the trust the mathematical community places in computer-assisted proofs.

2012 ACM Subject Classification Theory of computation → Logic and verification

Keywords and phrases Empty Hexagon Number, Discrete Computational Geometry, Erdős-Szekeres

Digital Object Identifier 10.4230/LIPIcs.ITP.2024.35

Supplementary Material *Software (Source Code):*

<https://github.com/bsubercaseaux/EmptyHexagonLean/tree/itp2024> [37]
archived at `swh:1:dir:29dc0e7145296997bcb1230b4e03cd14c8d75617`

Funding Supported by the National Science Foundation (NSF) grant CCF-2229099.

1 Introduction

Mathematicians are often rightfully skeptical of proofs that rely on extensive computation (e.g., the controversy around the four color theorem [42]). Nonetheless, many mathematically-interesting theorems have been resolved with the help of computers. SAT solving in particular has been a powerful tool for mathematics, successfully resolving Keller’s conjecture [2], the packing chromatic number of the infinite grid [35], the Pythagorean triples problem [20], Lam’s problem [3], and one case of the Erdős discrepancy conjecture [25]. Notably, all of these proofs follow the same two-step structure:



© Bernardo Subercaseaux, Wojciech Nawrocki, James Gallicchio, Cayden Codel, Mario Carneiro, and Marijn J. H. Heule;

licensed under Creative Commons License CC-BY 4.0

15th International Conference on Interactive Theorem Proving (ITP 2024).

Editors: Yves Bertot, Temur Kutsia, and Michael Norrish; Article No. 35; pp. 35:1–35:19



Leibniz International Proceedings in Informatics

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

35:2 Formal Verification of the Empty Hexagon Number

- **(Reduction)** Show that the mathematical theorem of interest is true if a concrete propositional formula ϕ is unsatisfiable.
- **(Solving)** Show that ϕ is indeed unsatisfiable.

Formal methods researchers have developed techniques that make the *solving* step reliable, reproducible, and trustworthy. For example, modern SAT solvers produce proofs of unsatisfiability in formal proof systems such as DRAT [43] that can in turn be checked with verified proof checkers such as `cake_lpr` [41]. These tools ensure that when a SAT solver declares a formula ϕ to be unsatisfiable, the formula is indeed unsatisfiable. In contrast, the *reduction* step is not as trustworthy, as it can use problem-specific mathematical insights that, when left unverified, threaten the correctness of the proof. A perfect example of a complex reduction can be found in a recent breakthrough in discrete computational geometry due to Heule and Scheucher [21]. They constructed (and solved) a formula ϕ whose unsatisfiability implies that every set of 30 points in the plane, without three in a common line, must contain an empty convex hexagon. However, as is common with such results, their reduction argument was only sketched, relied heavily on intuition, and had several gaps.

In this paper we complete and formalize their reduction in the Lean theorem prover [10]. We do so by connecting existing geometric definitions in the mathematical proof library `mathlib` [29] to the unsatisfiability of a particular SAT instance, thus setting a new standard for verifying results which rely on extensive computation. Our formalization is publicly available at <https://github.com/bsubercaseaux/EmptyHexagonLean/tree/itp2024>.

Verification of SAT proofs. Formal verification makes the SAT *solving* step trustworthy. For example, theorem provers and formal methods tools have been used to verify solvers [27,31,33] and proof checkers [26,41]. However, the *reduction* step has not received similar scrutiny, with only a few reductions having been verified. For instance, Cruz-Filipe and coauthors [8,9] used the Coq proof assistant to verify the reduction of the Pythagorean triples problem [20] to SAT, and Delemazure and colleagues [11] used Isabelle/HOL to verify SAT-based results in social choice theory for which minimal unsatisfiable sets of clauses were too large to extract human-readable proofs. More generally, Giljegård and Wennerbreck [16] built a CakeML library of verified SAT encodings, which they used to write verified SAT reductions for different puzzles, such as the *N-queens* problem. In this paper, we use reduction verification techniques based on those of Codel, Avigad, and Heule [6], which they developed in Lean.

Formal verification for SAT-based combinatorial geometry was pioneered by Marić [28]. He formally verified a reduction of a case of the Happy Ending Problem (see below) to SAT in Isabelle/HOL. We compare our work to his in Section 7.

Lean. Initially developed by Leonardo de Moura in 2013 [10], the Lean theorem prover has become a popular choice for formalizing modern mathematical research. Recent successes include the *Liquid Tensor Experiment* [5] and the proof of the polynomial Freiman-Ruzsa conjecture [17,34], both of which brought significant attention to Lean. A major selling point for Lean is the `mathlib` project [29], a monolithic formalization of foundational mathematics. By relying on `mathlib` for definitions, lemmas, and proof tactics, mathematicians can focus on the interesting components of a formalization while avoiding duplication of proof efforts. In turn, by making a formalization compatible with `mathlib`, future proof efforts can rely on work done today. In this spirit, we connect our results to `mathlib` as much as possible.

The Empty Hexagon Number. In the 1930s, Erdős and Szekeres, inspired by Esther Klein, showed that for any $k \geq 3$, one can find a sufficiently large number n such that every n points in the plane in *general position* (i.e., with no three points collinear) contain a convex k -gon,

i.e., a convex polygon with k vertices [13]. The minimal such n is denoted $g(k)$. The same authors later showed that $g(k) > 2^{k-2}$ and conjectured that this bound is tight [12]. Indeed, it is known that $g(5) = 9$ and $g(6) = 17$, with the latter result obtained by Szekeres and Peters 71 years after the initial conjecture via exhaustive computer search [40]. Larger cases remain open, with $g(k) \leq 2^{k+o(k)}$ being the best known upper bound [22, 38]. This problem is now known as the *Happy Ending Problem*, as it led to the marriage of Klein and Szekeres.

In a similar spirit, Erdős defined $h(k)$ to be the minimal number of points in general position that is guaranteed to contain a k -hole, or *empty k -gon*, meaning a convex k -gon with no other point inside. It is easy to check that $h(3) = 3$ and $h(4) = 5$. In 1978, Harborth established that $h(5) = 10$ [19]. Surprisingly, in 1983, Horton discovered constructions of arbitrarily large point sets that avoid k -holes for $k \geq 7$ [23]. Only $h(6)$ remained. The *Empty Hexagon Theorem*, establishing $h(6)$ to be finite, was proven independently by Gerken [15] and Nicolás [30] in 2006. In 2008, Valtr narrowed the range of possible values down to $30 \leq h(6) \leq 1717$, where the problem remained until the breakthrough by Heule and Scheucher [21], who used a SAT solver to prove that $h(6) \leq 30$, a result we refer to as the *Empty Hexagon Number*.

2 Outline of the proof

We will incrementally build sufficient machinery to prove the following theorem.

► **Theorem.** *Any finite set of 30 or more points in the plane in general position has a 6-hole.*

Outline of the proof. We begin Section 3 with a precise statement in Lean of the above theorem and involved geometric terms. In a nutshell, the proof consists of building a CNF formula ϕ_n such that from any set S of n points in general position without a 6-hole we can construct a satisfying assignment τ_S for ϕ_n . Then, checking that ϕ_{30} is unsatisfiable implies that no such set S of size 30 exists, thus implying the theorem. In order to construct ϕ_n , one must first discretize the continuous space \mathbb{R}^2 . *Triple orientations*, presented in Section 4, are a way to achieve this. Concretely, any three points p, q, r in general position correspond to either a clockwise turn, denoted by $\sigma(p, q, r) = -1$, or a counterclockwise turn, denoted by $\sigma(p, q, r) = +1$, depending on whether r is above the directed line \overrightarrow{pq} or not. In this way, every set S of points in general position induces an assignment $\sigma_S : S^3 \rightarrow \{-1, +1\}$ of triple orientations. We show in Section 4 that whether S contains a k -hole (i.e., $\text{HasEmptyKGon } k \ S$) depends entirely on σ_S . As each orientation $\sigma(p, q, r)$ can only take two values, we can represent each orientation $\sigma(p, q, r)$ with a boolean variable. Any set of points S in general position thus induces an assignment τ_S over its *orientation variables*. Because $\text{HasEmptyKGon } k \ S$ depends only on σ_S , it can be written as a boolean formula over the orientation variables. Unfortunately, it is practically infeasible to determine if such a formula is satisfiable with a naïve encoding. In order to create a better encoding, Section 5 shows that one can assume, without loss of generality, that the set of points S is in *canonical position*. Canonicity eliminates a number of symmetries from the problem – ordering, rotation, and mirroring – significantly reducing the search space. In Section 6, we show the correctness of the efficient encoding of Heule and Scheucher [21] for constructing a smaller CNF formula ϕ_n . Concretely, we show that any finite set of n points in canonical position containing no 6-hole would give rise to a propositional assignment τ_S satisfying ϕ_n . However, ϕ_{30} (depicted in Section 6) is unsatisfiable; therefore no such set of size 30 exists and the theorem follows by contradiction. As detailed in Section 6, to establish unsatisfiability of ϕ_{30} we passed the formula produced by our verified encoder to a SAT solver, and used a verified proof checker to certify the correctness of the resulting unsatisfiability proof. The construction of ϕ_n and τ_S involves sophisticated optimizations which we justify using geometric arguments. ◀

3 Geometric Preliminaries

We identify points with elements of \mathbb{R}^2 . Concretely, `abbrev Point := EuclideanSpace \mathbb{R}` (Fin 2). The next step is to define what it means for a k -gon to be *empty* (with respect to a set of points) and *convex*, which we do in terms of `mathlib` primitives.

```

/-- 'EmptyShapeIn S P' means that 'S' carves out an empty shape in 'P':
the convex hull of 'S' contains no point of 'P' other than those already in 'S'. -/
def EmptyShapeIn (S P : Set Point) : Prop :=
  ∀ p ∈ P \ S, p ∉ convexHull  $\mathbb{R}$  S

/-- 'ConvexIndep S' means that 'S' consists of extremal points of its convex hull,
i.e., the point set encloses a convex polygon. -/
def ConvexIndep (S : Set Point) : Prop :=
  ∀ a ∈ S, a ∉ convexHull  $\mathbb{R}$  (S \ {a})

/-- 'ConvexEmptyIn S P' means that 'S' forms a convex empty polygon in 'P' -/
def ConvexEmptyIn (S P : Set Point) : Prop :=
  ConvexIndep S ∧ EmptyShapeIn S P

/-- 'HasEmptyKGon k P' means that 'P' has a convex, empty 'k'-gon -/
def HasEmptyKGon (k : Nat) (P : Set Point) : Prop :=
  ∃ S : Finset Point, S.card = k ∧ ↑S ⊆ P ∧ ConvexEmptyIn S P

```

Let `SetInGenPos` be a predicate that states that a set of points is in *general position*, i.e., no three points lie on a common line (made precise in Section 4). With this we can already state the core theorem.

```

theorem hole_6_theorem : ∀ (pts : Finset Point),
  SetInGenPos pts → pts.card = 30 → HasEmptyKGon 6 pts

```

At the root of the encoding of Heule and Scheucher is the idea that the `ConvexEmptyIn` predicate can be determined by analyzing only triangles. In particular, that a set s of k points in a pointset S form an empty convex k -gon if and only if all the $\binom{k}{3}$ triangles induced by vertices in s are empty with respect to S . This is discussed informally in [21, Section 3, Eq. 4]. Concretely, we prove the following theorem:

```

theorem ConvexEmptyIn.iff_triangles {s : Finset Point} {S : Set Point}
  (sS : ↑s ⊆ S) (sz : 3 ≤ s.card) :
  ConvexEmptyIn s S ↔
  ∀ (t : Finset Point), t.card = 3 → t ⊆ s → ConvexEmptyIn t S

```

Proof sketch. We first prove a simple monotonicity lemma: if `ConvexIndep(s)`, then `ConvexIndep(s')` for every $s' \subseteq s$, and similarly `EmptyShapeIn(s, S)` \Rightarrow `EmptyShapeIn(s', S)` for every set of points S . By instantiating this monotonicity lemma over all subsets $t \subseteq s$ with $|t| = 3$ we get the forward direction of the theorem. For the backward direction it is easier to reason contrapositively: if the `ConvexIndep` predicate does not hold of s , or if s is not empty w.r.t. S , then we want to show that there is a triangle $t \subseteq s$ that is also not empty w.r.t. S . To see this, let H be the convex hull of s , and then by Carathéodory's theorem (cf. `theorem convexHull_eq_union` from `mathlib`), every point in H is a convex combination of at most 3 points in s , and consequently, of exactly 3 points in s . If s is non-empty w.r.t. S , then there is a point $p \in S \setminus s$ that belongs to H , and by Carathéodory, p is a convex combination of 3 points in $s \setminus \{a\}$, and thus lies inside a triangle $t \subseteq s$ (Figure 1a). If s does

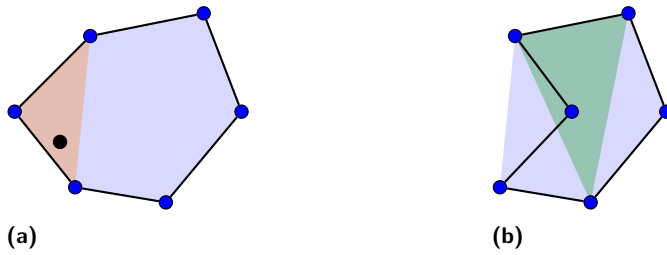


Figure 1 Illustration of the proof for `ConvexEmptyIn.iff_triangles`. The left subfigure shows how a point in $S \setminus s$ that lies inside s will be inside one of the triangles induced by the convex hull of s (orange triangle). The right subfigure shows how if the `ConvexIndep` predicate does not hold of s , then some point $a \in s$ will be inside one of the triangles induced by the convex hull of $s \setminus \{a\}$.

not hold `ConvexIndep`, then there is a point $a \in s$ such that $a \in \text{convexHull}(s \setminus \{a\})$, and by Carathéodory again, a is a convex combination of 3 points in s , and thus lies inside a triangle $t \subseteq s \setminus \{a\}$ (Figure 1b). ◀

In the next section, we show how to use boolean variables to encode which triangles (and, by the above theorem, which k -holes) are empty in a pointset.

4 Triple Orientations

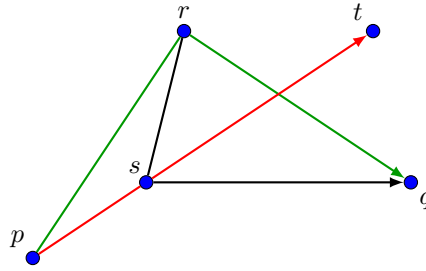
An essential step for obtaining computational proofs of geometric results is *discretization*: problems concerning the existence of an object \mathcal{O} in a continuous search space like \mathbb{R}^2 must be reformulated in terms of the existence of a discrete, finitely-representable object \mathcal{O}' that a computer can search for. It is especially challenging to discretize problems in which the desired geometric object \mathcal{O} is characterized by very specific coordinates of points, thus requiring the computer to use floating-point arithmetic, which suffers from numerical instability. Fortunately, this is not the case for Erdős-Szekeres-type problems such as determining the value of $h(k)$, as their properties of interest (e.g., convexity and emptiness) can be described in terms of axiomatizable relationships between points and lines (e.g., point p is above the line \vec{qr} , lines \vec{qr} and \vec{st} intersect, etc.) that are invariant under rotation, translation, and even small perturbations of the coordinates. We can discretize these relationships with boolean variables, thus making us agnostic to the specific coordinates of the points. The combinatorial abstraction that has been most widely used in Erdős-Szekeres-type problems is that of *triple orientations* [21,32], also known as *signotopes* [14,36], Knuth’s *counterclockwise relation* [24], or *signatures* [39]. Given points p, q, r , their *triple-orientation* is defined as:

$$\sigma(p, q, r) = \text{sign det} \begin{pmatrix} p_x & q_x & r_x \\ p_y & q_y & r_y \\ 1 & 1 & 1 \end{pmatrix} = \begin{cases} 1 & \text{if } p, q, r \text{ are oriented counterclockwise,} \\ 0 & \text{if } p, q, r \text{ are collinear,} \\ -1 & \text{if } p, q, r \text{ are oriented clockwise.} \end{cases}$$

We define σ in Lean using `mathlib`’s definition of the determinant.

```
inductive Orientation : Type where
| cw      -- Clockwise turn
| ccw     -- Counter-clockwise turn
| collinear -- Collinear
```


35:6 Formal Verification of the Empty Hexagon Number



■ **Figure 2** Illustration of triple orientations, where $\sigma(p, r, q) = -1$, $\sigma(r, s, q) = 1$, and $\sigma(p, s, t) = 0$.

```

noncomputable def  $\sigma$  (p q r : Point) : Orientation :=
  let det := Matrix.det !![p.x, q.x, r.x ; p.y, q.y, r.y ; 1, 1, 1]
  if 0 < det then ccw
  else if det < 0 then cw
  else collinear

```

Using the function σ we can define the notion of *general position* for collections (e.g., finite sets, lists, etc.) of points, simply postulating that $\sigma(p, q, r) \neq 0$ for every three distinct points p, q, r in the collection. Furthermore, we can start formalizing sets of points that are *equivalent* with respect to their triple orientations, and consequently, properties of pointsets that are fully captured by their triple orientations (*orientation properties*).

```

structure  $\sigma$ Equiv (S T : Set Point) where
  f : Point  $\rightarrow$  Point
  bij : Set.BijOn f S T
  parity : Bool
   $\sigma$ _eq :  $\forall$  (p  $\in$  S) (q  $\in$  S) (r  $\in$  S),  $\sigma$  p q r = parity  $\wedge\wedge\wedge$   $\sigma$  (f p) (f q) (f r)

```

```

def OrientationProperty (P : Set Point  $\rightarrow$  Prop) :=
   $\forall$  {{S T}}, S  $\simeq\sigma$  T  $\rightarrow$  P S  $\rightarrow$  P T -- ' $\simeq\sigma$ ' is infix notation for ' $\sigma$ Equiv'

```

Our notion of σ equivalence allows for all orientations to be flipped. The $\wedge\wedge\wedge$ (xor) operation does nothing when `parity` is false, and negates the orientation when `parity` is true. See Section 5 for more details.

To illustrate how these notions will be used, let us consider the property $\pi_k(S) \triangleq$ “pointset S contains an empty convex k -gon”, formalized as `HasEmptyKGon`.

Based on `ConvexEmptyIn.iff_triangles`, we know that $\pi_k(S)$ can be written in terms of whether certain triangles are empty w.r.t S . We can define triangle membership using σ , and prove its equivalence to the geometric definition.

```

/-- 'Means that 'a' is in the triangle 'pqr', possibly on the boundary. -/
def PtInTriangle (a : Point) (p q r : Point) : Prop :=
  a  $\in$  convexHull  $\mathbb{R}$  {p, q, r}

/-- 'Means that 'a' is in the triangle 'pqr' strictly, not on the boundary. -/
def  $\sigma$ PtInTriangle (a p q r : Point) : Prop :=
   $\sigma$  p q a =  $\sigma$  p q r  $\wedge$   $\sigma$  p a r =  $\sigma$  p q r  $\wedge$   $\sigma$  a q r =  $\sigma$  p q r

theorem  $\sigma$ PtInTriangle_iff {a p q r : Point} (gp : InGenPos4 a p q r) :
   $\sigma$ PtInTriangle a p q r  $\leftrightarrow$  PtInTriangle a p q r

```


Heule and Scheucher used the orientation-based definition [21] and, as it is common in the area, its equivalence to the *ground-truth* mathematical definition was left implicit. This equivalence, formalized in `theorem σ PtInTriangle_iff` is not trivial to prove: the forward direction in particular requires reasoning about convex combinations and determinants. Using the previous theorem, we can generalize to k -gons as follows.

```
def  $\sigma$ IsEmptyTriangleFor (a b c : Point) (S : Set Point) : Prop :=
   $\forall$  s  $\in$  S,  $\neg$  $\sigma$ PtInTriangle s a b c

def  $\sigma$ HasEmptyKGon (n : Nat) (S : Set Point) : Prop :=
   $\exists$  s : Finset Point, s.card = n  $\wedge$   $\uparrow$ s  $\subseteq$  S  $\wedge$   $\forall$  (a  $\in$  s) (b  $\in$  s) (c  $\in$  s),
  a  $\neq$  b  $\rightarrow$  a  $\neq$  c  $\rightarrow$  b  $\neq$  c  $\rightarrow$   $\sigma$ IsEmptyTriangleFor a b c S

theorem  $\sigma$ HasEmptyKGon_iff_HasEmptyKGon {n : Nat} (gp : ListInGenPos pts) :
   $\sigma$ HasEmptyKGon n pts.toFinset  $\leftrightarrow$  HasEmptyKGon n pts.toFinset
```

Then, because `σ HasEmptyKGon` is ultimately defined in terms of `σ` , we can prove

```
lemma OrientationProperty_ $\sigma$ HasEmptyKGon {n : Nat} : OrientationProperty
  ( $\sigma$ HasEmptyKGon n)
```

Which in combination with `theorem σ HasEmptyKGon_iff_HasEmptyKGon`, provides

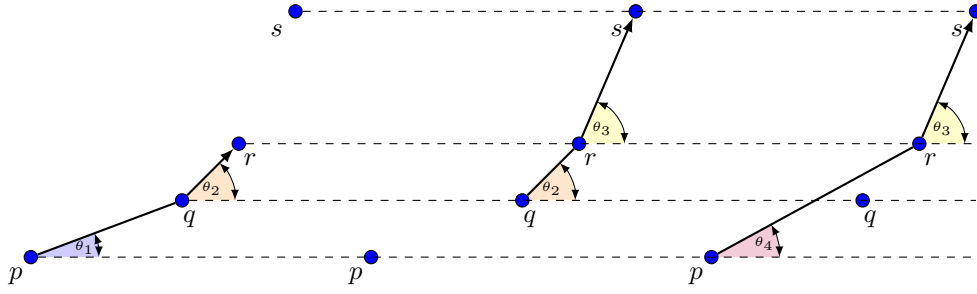
```
theorem OrientationProperty_HasEmptyKGon {n : Nat} : OrientationProperty
  (HasEmptyKGon n)
```

The previous theorem is important for two reasons. First, if `σ` is invariant under certain point transformations (e.g., rotations, translations, etc.), then any orientation property is invariant under the same transformations. This is a powerful tool for performing symmetry breaking (see Section 5). For a concrete example, consider a proof of an Erdős-Szekeres-type result that starts by saying “*we assume without loss of generality that points p_1, \dots, p_n all have positive y -coordinates.*” Since `σ` is invariant under translation, we can see that this assumption indeed does not impact the validity of the proof.

Second, as introduced at the beginning of this section, SAT encodings for Erdős-Szekeres-type problems use triple orientations to capture properties like convexity and emptiness, thus discretizing the problem. Because we have proved that `$\pi_k(S)$` is an orientation property, the values of `σ` on the points in `S` contain enough information to determine whether `$\pi_k(S)$` holds. Therefore, we have proved that given n points, it is enough to analyze the values of `σ` over these points, a discrete space with at most 2^{n^3} possibilities, instead of grappling with a continuous search space on n points, $(\mathbb{R}^2)^n$. This is the key idea that will allow us to transition from the finitely-verifiable statement “*no set of triple orientations over n points satisfies property π_k* ” to the desired statement “*no set of n points satisfies property π_k* .”

4.1 Properties of orientations

We now prove, assuming points are sorted left-to-right (which is justified in Section 5), that certain *σ -implication-properties* hold. Consider four points p, q, r, s with $p_x < q_x < r_x < s_x$. If p, q, r are oriented counterclockwise, and q, r, s are oriented counterclockwise as well, then it follows that p, r, s must be oriented counterclockwise (see Figure 3). We prove a number of properties of this form:



■ **Figure 3** Illustration for $\sigma(p, q, r) = 1 \wedge \sigma(q, r, s) = 1 \implies \sigma(p, r, s) = 1$. As we have assumptions $\theta_3 > \theta_2 > \theta_4$ by the forward direction of the *slope-orientation equivalence*, we deduce $\theta_3 > \theta_4$, and then conclude $\sigma(p, r, s) = 1$ by the backward direction of the *slope-orientation equivalence*.

theorem σ_prop_1 (h : Sorted₄ p q r s) (gp : InGenPos₄ p q r s) :
 $\sigma\ p\ q\ r = ccw \rightarrow \sigma\ q\ r\ s = ccw \rightarrow \sigma\ p\ r\ s = ccw$

[...]

theorem σ_prop_3 (h : Sorted₄ p q r s) (gp : InGenPos₄ p q r s) :
 $\sigma\ p\ q\ r = cw \rightarrow \sigma\ q\ r\ s = cw \rightarrow \sigma\ p\ r\ s = cw$

Our proofs of these properties are based on an equivalence between the orientation of a triple of points and the *slopes* of the lines that connect them. Namely, if p, q, r are sorted from left to right, then (i) $\sigma(p, q, r) = 1 \iff \text{slope}(\overrightarrow{pq}) < \text{slope}(\overrightarrow{pr})$ and (ii) $\sigma(p, q, r) = 1 \iff \text{slope}(\overrightarrow{pr}) < \text{slope}(\overrightarrow{qr})$. By first proving these *slope-orientation* equivalences we can then easily prove σ_prop_1 and others, as illustrated in Figure 3.

These properties will be used in Section 6 to justify clauses (4) and (5) of the SAT encoding; these clauses are commonly added in orientation-based SAT encodings to reduce the search space by removing some “unrealizable” orientations [21, 32, 36, 39].

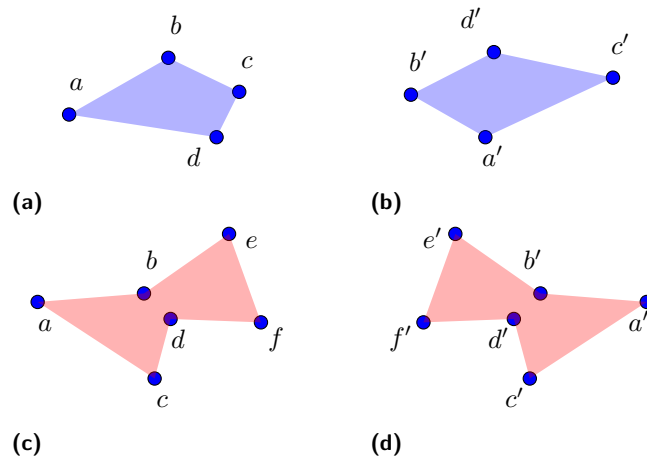
$$(\neg o_{a,b,c} \vee \neg o_{a,c,d} \vee o_{a,b,d}) \wedge (o_{a,b,c} \vee o_{a,c,d} \vee \neg o_{a,b,d})$$

5 Symmetry Breaking

Symmetry breaking plays a key role in SAT solving by reducing the search space of satisfying assignments for a formula [1, 7], thus making a wider range of formulas practical to solve. For example, if one proves that all satisfying assignments to a formula ϕ have either (i) $x_1 = 0, x_2 = 1$, or (ii) $x_1 = 1, x_2 = 0$, and that there is a bijection between satisfying assignments of forms (i) and (ii), then one can assume, *without loss of generality*, that $x_1 = 0, x_2 = 1$, and thus add unit clauses $\overline{x_1}$ and x_2 to the formula ϕ while preserving its satisfiability. There are several techniques that can automatically find symmetry-breaking clauses, such as structured bounded variable addition [18], but it is accepted wisdom in the SAT-solving community that problem-specific symmetry breaking is more effective.

In their proof of the Empty Hexagon Number, Heule and Scheucher showed that for any list of points in general position, there exists a list of points in *canonical position* with the same triple-orientations. Canonical position is defined as follows.

► **Definition 1** (Canonical Position). *A list of points $L = (p_1, \dots, p_n)$ is said to be in canonical position if it satisfies all of the following properties:*



■ **Figure 4** The pointsets depicted in Figures 4a and 4b are σ -equivalent with `parity := false` since the bijection f defined by $(a, b, c, d) \mapsto (b', d', c', a')$ satisfies $\sigma(p_i, p_j, p_k) = \sigma(f(p_i), f(p_j), f(p_k))$ for every $\{p_i, p_j, p_k\} \subseteq \{a, b, c, d\}$. On the other hand, no orientation-preserving bijection exists for Figures 4c and 4d, which are only σ -equivalent with `parity := true`.

(General Position) No three points are collinear, i.e., for all $1 \leq i < j < k \leq n$, we have $\sigma(p_i, p_j, p_k) \neq 0$.

(x-order) The points are sorted with respect to their x -coordinates, i.e., $x(p_i) < x(p_j)$ for all $1 \leq i < j \leq n$.

(CCW-order) All orientations $\sigma(p_1, p_i, p_j)$, with $1 < i < j \leq n$, are counterclockwise.

(Lex-order) The list of orientations $(\sigma(p_{\lfloor \frac{n}{2} \rfloor - 1}, p_{\lfloor \frac{n}{2} \rfloor}, p_{\lfloor \frac{n}{2} \rfloor + 1}), \dots, \sigma(p_2, p_3, p_4))$ is not lexicographically smaller than the list $(\sigma(p_{\lfloor \frac{n}{2} \rfloor + 1}, p_{\lfloor \frac{n}{2} \rfloor + 2}, p_{\lfloor \frac{n}{2} \rfloor + 3}), \dots, \sigma(p_{n-2}, p_{n-1}, p_n))$.

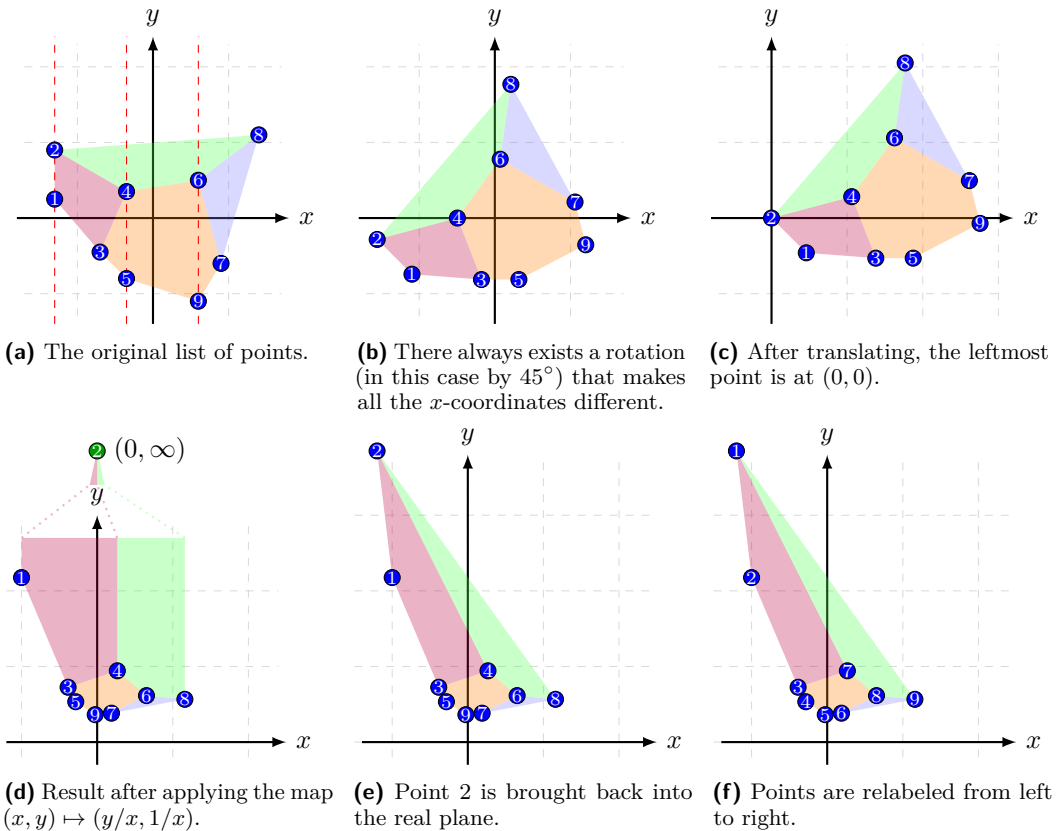
The three ordering properties each break a different symmetry. First, the x -order property breaks symmetry due to how we label the points by ensuring that the points are labeled from left to right. The x -order property also simplifies the encoding of clauses (1)–(5), as they rely on the points being sorted. Second, the CCW-order property breaks symmetry due to *rotation* by fixing the orientations involving the leftmost point p_1 .

Third, the lex-order property breaks symmetry due to *reflection*. Reflecting a set of points S over a line (e.g., with the map $(x, y) \mapsto (-x, y)$) preserves the presence of k -holes and convex k -gons. This operation does not quite preserve orientations, but rather flips them (clockwise orientations become counterclockwise and vice versa). Our definition of σ -equivalence includes a *parity* flag for this purpose: `parity := false` corresponds to the case that orientations are the same, and `parity := true` corresponds to the case that all orientations have been flipped. See the point sets in Figure 4 for an example.

The lex-order property, then, picks between a set of points and its reflection over $x = 0$. The vector of consecutive orientations from the middle to the left is assumed to be at least as big as the vector of consecutive orientations from the middle to the right. This constraint is not geometrically meaningful, but is easy to implement in the SAT encoding.

We prove that there always exists a σ -equivalent point set in canonical position.

```
theorem symmetry_breaking : ListInGenPos 1 →
  ∃ w : CanonicalPoints, Nonempty (1.toFinset ≈σ w.points.toFinset)
```



■ **Figure 5** Illustration of the proof of the symmetry breaking theorem. Note that the highlighted holes are preserved as σ -equivalence is preserved. For simplicity we have omitted the illustration of the Lex order property.

Proof Sketch. The proof proceeds in 6 steps, illustrated in Figure 5. In each of the steps, we construct a new list of points that is σ -equivalent to the previous one, with the last one being in canonical position.¹ The main justification for each step is that, given that the function σ is defined as a sign of the determinant, applying transformations that preserve (or, when `parity := true`, uniformly reverse) the sign of the determinant will preserve (or uniformly reverse) the values of σ .

For example, given the identity $\det(AB) = \det(A) \det(B)$, if we apply a transformation to the points that corresponds to multiplying by a matrix B such that $\det(B) > 0$, then $\text{sign}(\det(A)) = \text{sign}(\det(AB))$, and thus orientations will be preserved. **Step 1:** we transform the list of points so that no two points share the same x -coordinate. This can be done by applying a rotation to the list of points, which corresponds to multiplying by a rotation matrix. Rotations always have determinant 1. **Step 2:** we translate all points by a constant vector t , which corresponds to multiplying by a translation matrix, to bring the leftmost point p_1 to position $(0, 0)$. As a result, every other point has a positive x -coordinate.

¹ Even though we defined σ -equivalence for sets of points, our formalization goes back and forth between sets and lists. Given that symmetry breaking distinguishes between the order of the points e.g., x -order, this proof proceeds over lists. All permutations of a list are immediately σ -equivalent.

Let L_2 be the list of points excluding p_1 after Step 2. **Step 3:** we apply the projective transformation $f : (x, y) \mapsto (y/x, 1/x)$ to every point in L_2 , showing that this preserves orientations within L_2 . To see that this mapping is a σ -equivalence consider that

$$\begin{aligned} \text{sign det} \begin{pmatrix} p_x & q_x & r_x \\ p_y & q_y & r_y \\ 1 & 1 & 1 \end{pmatrix} &= \text{sign det} \left(\begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} p_y/p_x & q_y/q_x & r_y/r_x \\ 1/p_x & 1/q_x & 1/r_x \\ 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} p_x & 0 & 0 \\ 0 & q_x & 0 \\ 0 & 0 & r_x \end{pmatrix} \right) \\ &= \text{sign} \left(1 \cdot \det \begin{pmatrix} p_y/p_x & q_y/q_x & r_y/r_x \\ 1/p_x & 1/q_x & 1/r_x \\ 1 & 1 & 1 \end{pmatrix} \cdot p_x q_x r_x \right) = \text{sign det} \begin{pmatrix} p_y/p_x & q_y/q_x & r_y/r_x \\ 1/p_x & 1/q_x & 1/r_x \\ 1 & 1 & 1 \end{pmatrix}. \end{aligned}$$

To preserve orientations with respect to the leftmost point $(0, 0)$, we set $f((0, 0)) = (0, \infty)$, a special point that is treated separately as follows. As the function σ takes points in \mathbb{R}^2 as arguments, we need to define an extension $\sigma_{(0, \infty)}(q, r) = \begin{cases} 1 & \text{if } q_x < r_x \\ -1 & \text{otherwise.} \end{cases}$, We then show

that $\sigma((0, 0), q, r) = \sigma_{(0, \infty)}(f(q), f(r))$ for all points $q, r \in L_2$.

Step 4: we sort the list L_2 by x -coordinate in increasing order, thus obtaining a list L_3 . This can be done while preserving σ -equivalence because sorting corresponds to a permutation, and all permutations of a list are σ -equivalent by definition. **Step 5:** we check whether the Lex order condition above is satisfied in L_3 , and if it is not, we reflect the pointset, which preserves σ -equivalence with `parity := true`. Note that in such a case we need to relabel the points from left to right again.

Step 6: we bring point $(0, \infty)$ back into the range by first finding a constant c such that all points in L_3 are to the right of the line $y = c$, and then finding a large enough value M such that (c, M) has the same orientation with respect to the other points as $(0, \infty)$ did, meaning that $\sigma((c, M), q, r) = \sigma_{(0, \infty)}(q, r)$ for every $q, r \in L_3$.

Finally, we note that the list of points obtained in step 6 satisfies the CCW-order property by the following reasoning: if $1 < i < j \leq n$ are indices, then

$$\begin{aligned} \sigma(p_1, p_i, p_j) = 1 &\iff \sigma((c, M), p_i, p_j) = 1 \\ &\iff \sigma_{(0, \infty)}(p_i, p_j) = 1 && \text{(By step 6)} \\ &\iff (p_i)_x < (p_j)_x && \text{(By definition of } \sigma_{(0, \infty)}) \\ &\iff \text{true.} && \text{(By step 4, since points are sorted and } i < j) \end{aligned}$$

This concludes the proof. \blacktriangleleft

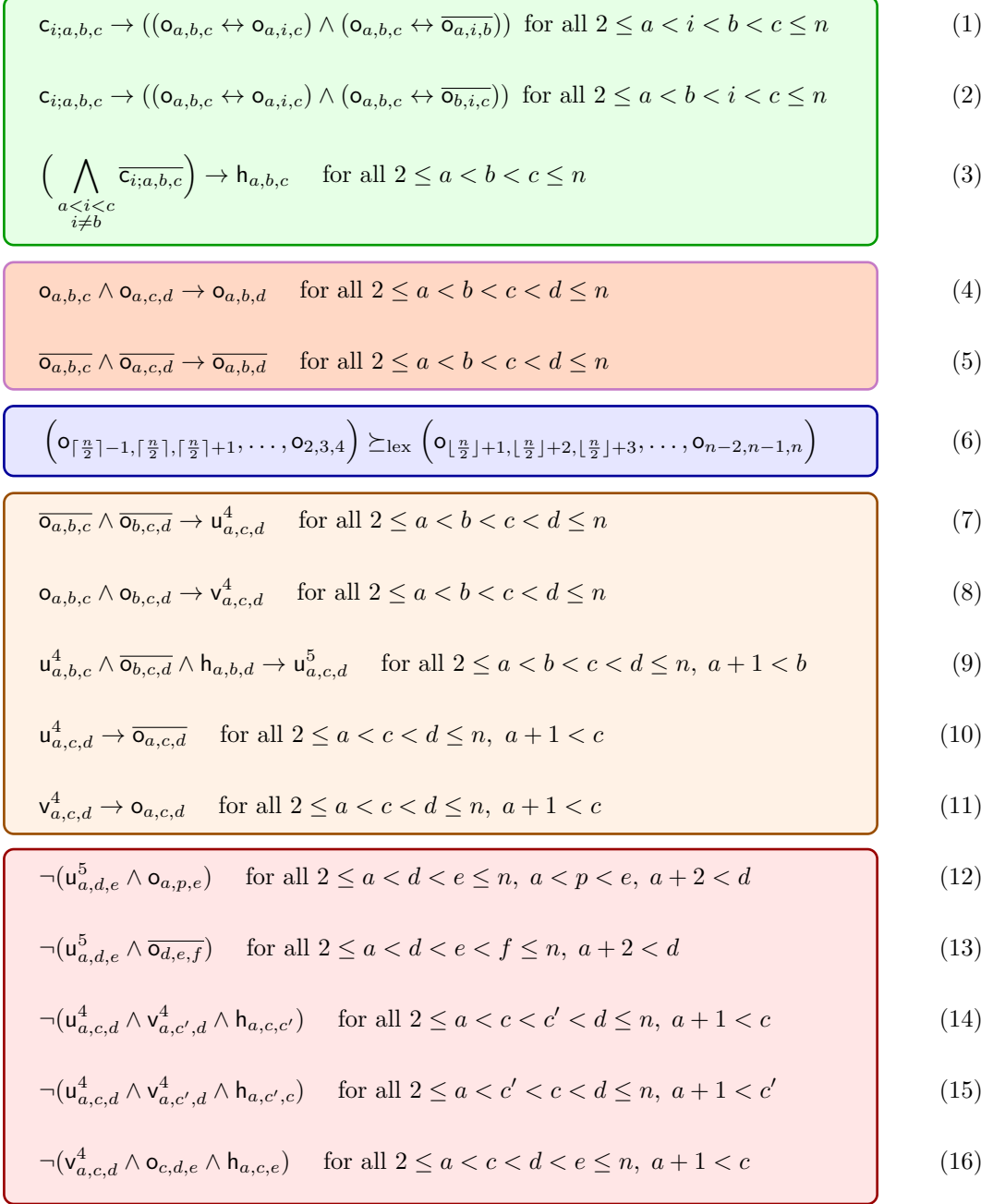
Compared to the symmetry-breaking transformation described by Heule and Scheucher, our transformation is simpler. Nonetheless, proving the above theorem in Lean was tedious, as we had to show that the properties from the previous steps were preserved at each new step, which carried substantial proof burden. In particular, steps 3 through 6 required careful bookkeeping and special handling of the distinguished point p_1 .

6 The Encoding and Its Correctness

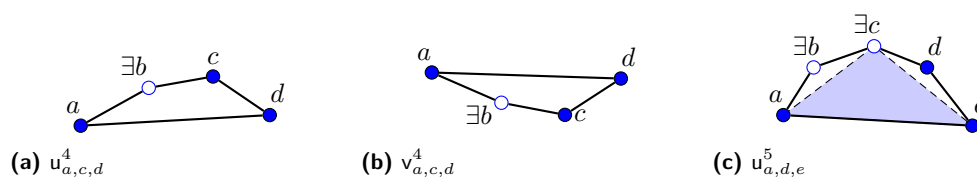
Having established the reduction to orientations, and the symmetry-breaking assumption of canonicity, we now turn to the construction of a CNF formula ϕ_n whose unsatisfiability would imply that every set of n points contains a 6-hole.² The formula is detailed in Section 6.

² Satisfiability of ϕ_n would *not* necessarily imply the existence of a point set without a 6-hole, due to the *realizability problem* (see e.g., [36]).

35:12 Formal Verification of the Empty Hexagon Number



■ **Figure 6** Encoding based on that of Heule and Scheucher for the Empty Hexagon Number [21]. Each line determines a set of clauses. Unsatisfiability of the formula below for $n = 30$ implies $h(6) \leq 30$, as detailed throughout the paper.



■ **Figure 7** Illustration of the 4-cap (7a), 4-cup (7b), and 5-cap (7c) variables. The highlighted region denotes an empty triangle.

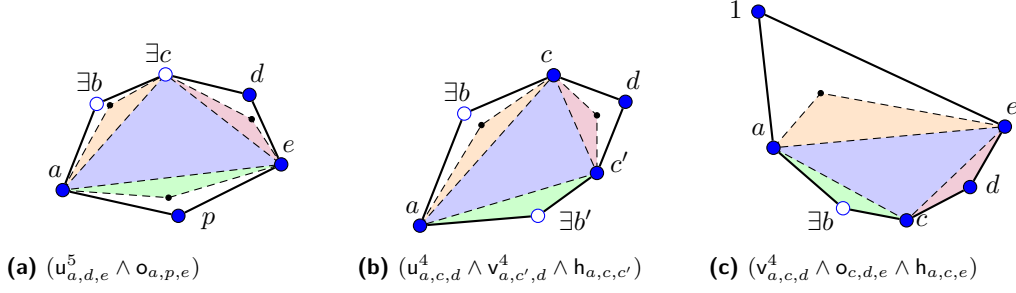
Variables. Let $S = (p_1, \dots, p_n)$ be the list of points in canonical position. We explain the variables of ϕ_n by specifying their values in the propositional assignment τ_S that is our intended model of ϕ_n corresponding to S . We then have:

- For every $2 \leq a < b < c \leq n$, $\circ_{a,b,c}$ is true iff $\sigma(p_a, p_b, p_c) = +1$.³
The first optimization observes that orientations are antisymmetric: if (p, q, r) is counterclockwise then (q, p, r) is clockwise, etc. Thus one only needs $\circ_{a,b,c}$ for ordered triples (a, b, c) , reducing the number of orientation variables by a factor of $3! = 6$ relative to using all triples. The second optimization uses the **CCW-order** property of canonical positions: since all $\circ_{1,a,b}$ are true, we may as well omit them from the encoding.
- Next, for every $a < b < c$ with $a < i < b$ or $b < i < c$, the variable $c_{i;a,b,c}$ is true iff $\sigma\text{PtInTriangle } S[i] S[a] S[b] S[c]$ holds. By $\sigma\text{PtInTriangle_iff}$, this is true exactly iff p_i is inside the triangle $p_a p_b p_c$. The reason for assuming (a, b, c) to be ordered is again symmetry: $p_a p_b p_c$ is the same triangle as $p_a p_c p_b$, etc. Furthermore thanks to the **x-order** property of canonical positions, if p_i is in the triangle then $x(p_a) < x(p_i) < x(p_c)$. This implies that $a < i < c$, leaving one case distinction permuting (i, b) .
- For every $a < b < c$, $h_{a,b,c}$ is true iff $\sigma\text{IsEmptyTriangleFor } S[a] S[b] S[c] S$ holds. By a geometro-combinatorial connection analogous to ones above, this is true iff $p_a p_b p_c$ is a 3-hole.
- Finally, one defines 4-cap, 5-cap, and 4-cup variables. For $a + 1 < c < d$, $u_{a,c,d}^4$ is true iff there is b with $a < b < c$ with $\sigma(p_a, p_b, p_c) = \sigma(p_b, p_c, p_d) = -1$. $v_{a,c,d}^4$ is analogous, except in that the two orientations are required to be counterclockwise. These are the 4-caps and 4-cups, respectively. The 5-cap variables $u_{a,d,e}^5$ are defined for $a + 2 < d < e$. We set $u_{a,d,e}^5$ to true iff there exists c with $a + 1 < c < d$ such that $u_{a,c,d}^4$, $\circ_{c,d,e}$, and $h_{a,c,e}$ are all true. Intuitively, 4-caps and 4-cups are clockwise and counterclockwise arcs of length 4, respectively, whereas 5-caps are clockwise arcs of length 5 containing a 3-hole. All three are depicted in Figure 7. The usage of these variables is crucial to an efficient encoding: we will show below that a hexagon can be covered by only 4 triangles, so one need not consider all $\binom{6}{3}$ triangles contained within it.

Satisfaction. We now have to justify that the clauses of ϕ_n are satisfied by the intended interpretation τ_S for a 6-hole-free point set S . The variable-defining clauses (1)–(3) and (7)–(11) follow essentially by definition combined with boolean reasoning. The orientation properties (4) and (5) have been established in the family of theorems σ_prop_i . The lexicographic ordering clauses (6) follow from the **Lex order** property of canonical positions. Thus we are left with clauses (12)–(16) which forbid the presence of certain 6-holes.⁴ We illustrate why

³ Since the point set is in general position, we have $\neg\circ_{a,b,c} \iff \sigma(p_a, p_b, p_c) = -1$.

⁴ They are intended to forbid *all* 6-holes, but proving completeness is not necessary for an unsatisfiability-based result.



■ **Figure 8** Illustration of some *forbidden configurations* that imply 6-holes. Figure 8a corresponds to the configuration forbidden by clause (12), Figure 8b to the one forbidden by clause (14), and Figure 8c to clause (16). All highlighted regions denote empty triangles.

clause (12) is true. The contrapositive is easier to state: if τ_S satisfies $u_{a,d,e}^5 \wedge o_{a,p,e}$, then S contains a 6-hole. The intuitive argument is depicted in Figure 8a. The clause directly implies the existence of a convex hexagon $apedcb$ such that ace is a 3-hole. It turns out that this is enough to ensure the existence of a 6-hole by “flattening” the triangles ape , edc , and cba , if necessary, to obtain empty triangles $ap'e$, $ed'c$, and $cb'a$, which can be assembled into a 6-hole $ap'ed'cb'$.

Justifying this formally turned out to be complex, requiring a fair bit of reasoning about point Arcs and σ_{CCWP} oints: lists of points winding around a convex polygon. Luckily, the main argument can be summarized in terms of two facts: (a) any triangle abc contains an empty triangle $ab'c$; and (b) empty shapes sharing a common line segment can be glued together. Formally, (a) can be stated as

```

theorem  $\sigma$ IsEmptyTriangleFor_exists (gp : ListInGenPos S)
  (abc : [a, b, c]  $\subseteq$  S) :  $\exists$  b'  $\in$  S,  $\sigma$  a b' c =  $\sigma$  a b c
   $\wedge$  (b' = b  $\vee$   $\sigma$ PtInTriangle b' a b c)  $\wedge$   $\sigma$ IsEmptyTriangleFor a b' c S.toFinset
    
```

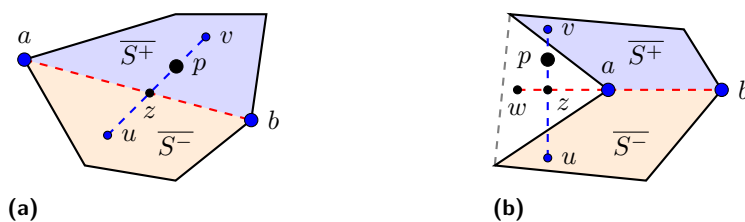
Proof. Given points p, q , say that $p \leq q$ iff p is in the triangle aqc . This is a preorder. Now, the set $S' = \{x \in S \mid \sigma(a, x, c) = \sigma(a, b, c) \wedge x \leq b\}$ is finite and so has a weakly minimal element b' , in the sense that no $x \in S'$ has $x < b'$. Emptiness of $ab'c$ follows by minimality. ◀

Moving on, (b) follows from a *triangulation lemma*: given any convex point set S and a line \overleftrightarrow{ab} between two vertices of S , the convex hull of S is contained in the convex hulls of points on either side of \overleftrightarrow{ab} . That is:

```

theorem split_convexHull (cvx : ConvexIndep S) :
   $\forall$  {a b}, a  $\in$  S  $\rightarrow$  b  $\in$  S  $\rightarrow$ 
  convexHull  $\mathbb{R}$  S  $\subseteq$  convexHull  $\mathbb{R}$  {x  $\in$  S |  $\sigma$  a b x  $\neq$  ccw}
   $\cup$  convexHull  $\mathbb{R}$  {x  $\in$  S |  $\sigma$  a b x  $\neq$  cw}
    
```

Proof. Let $S^+ = \{x \in S \mid \sigma(a, b, x) \geq 0\}$ and $S^- = \{x \in S \mid \sigma(a, b, x) \leq 0\}$ be the two sets in the theorem, and let $p \in \overline{S}$, where \overline{S} denotes the convex hull of S . Assume WLOG that $\sigma(a, b, p) \geq 0$. (We would like to show that $p \in \overline{S^+}$.) Now p is a convex combination of elements of S^+ and elements of S^- , so there exist points $u \in \overline{S^-}$ and $v \in \overline{S^+}$ such that p lies on the \overline{uv} line. Because $\{x \mid \det(a, b, x) \leq 0\} \supseteq S^-$ is convex, it follows that $\det(a, b, u) \leq 0$, and likewise $\det(a, b, v) \geq 0$, so they lie on opposite sides of the \overleftrightarrow{ab} line and hence \overline{uv} intersects \overleftrightarrow{ab} at a point z . The key point is that z must in fact be on the line



■ **Figure 9** Illustration of the proof for `split_convexHull`. (a) Given point p , we obtain points u and v inside the two halves and z as the point of intersection with the line \overline{ab} . (b) In this (contradictory) situation, the point z has ended up outside the segment \overline{ab} , because S is not actually convex. In this case we construct w such that z is on the \overline{wa} segment, and observe that w, z, a, b are collinear.

segment \overline{ab} ; assuming that this was the case, we could obtain z as a convex combination of a and b , and p as a convex combination of v and z , and since v is in $\overline{S^+}$ and $a, b \in S^+ \subseteq \overline{S^+}$ we can conclude $p \in \overline{S^+}$. To show that $z \in \overline{ab}$, suppose not, so that a lies between z and b (see Figure 9b). (The case where z is on the b side is similar.) We can decompose z as a convex combination of some $w \in \overline{S \setminus \{a\}}$ and a , which means that w, z, a, b are collinear and appear in this order on the line. Therefore a is a convex combination of w and b , which means that $a \in \overline{S \setminus \{a\}}$ which violates convexity of S . ◀

By contraposition, the triangulation lemma directly implies that if $\{x \in S \mid \sigma(a, b, x) \neq +1\}$ and $\{x \in S \mid \sigma(a, b, x) \neq -1\}$ are both empty shapes in P , then S is an empty shape in P .

6.1 Running the CNF

Having now shown that our main result follows if ϕ_{30} is unsatisfiable, we run a distributed computation to check its unsatisfiability. We solve the SAT formula ϕ_{30} produced by Lean using the same setup as Heule and Scheucher [21], although using different hardware: the Bridges 2 cluster of the Pittsburgh Supercomputing Center [4]. Following Heule and Scheucher, we partition the problem into 312 418 subproblems. Each of these subproblems was solved using CaDiCaL version 1.9.5. CaDiCaL produced an LRAT proof for each execution, which was validated using the `cake_lpr` verified checker on-the-fly in order to avoid writing/storing/reading large files. The total runtime was 25 876.5 CPU hours, or roughly 3 CPU years. The difference in runtime relative to Heule and Scheucher’s original run is purely due to the difference in hardware. Additionally, we validated that the subproblems cover the entire search space as Heule and Scheucher did [21, Section 7.3]. This was done by verifying the unsatisfiability of another formula that took 20 seconds to solve.

The artifact for this paper includes scripts to validate any individual subproblem, as well as the summary proof that the subproblems cover the search space. However, the unsatisfiability of ϕ_{30} depends on the unsatisfiability of *all* (hundreds of thousands of) subproblems. A skeptical reader might wish to examine the proof files for all subproblems, but we estimated the total proof size to be tens or hundreds of terabytes, far too much to reasonably store and distribute. Instead, the skeptical reader must run the entire 3 CPU year computation. We believe this trust story can be somewhat improved, but we leave such a challenge to future work.

7 Related Work

Our formalization is closely related to a prior development in which Marić put proofs of $g(6) \leq 17$ on a more solid foundation [28]. The inequality, originally obtained by Szekeres and Peters [40] using a specialized, unverified search algorithm, was confirmed by Marić using a formally-verified SAT encoding. Marić introduced an optimized encoding based on nested convex hull structures, which, when combined with performance advances in modern SAT solvers, significantly improved the search time over the unverified computation.

Our work focuses on the closely-related problem of determining k -hole numbers $h(k)$. Rather than devise a new SAT encoding, we use essentially the same encoding presented by Heule and Scheucher [21]. Interestingly, a formal proof of $g(6) \leq 17$ can be obtained as a corollary of our development. We can assert the hole variables $h_{a,b,c}$ as true while leaving the remainder of the encoding in Section 6 unchanged, which trivializes constraints about emptiness so that only the convexity constraints remain.⁵ The resulting CNF formula asserts the existence of a set of n points with no convex 6-gon. We checked this formula to be unsatisfiable for $n = 17$, giving the same result as Marić:

```
theorem gon_6_theorem : ∀ (pts : Finset Point),
  SetInGenPos pts → pts.card = 17 → HasConvexKGon 6 pts
```

Since both formalizations can be executed, we performed a direct comparison against Marić’s encoding. On a personal laptop, we found that it takes negligible time (below 1s) for our verified Lean encoder to output the full CNF. In contrast, Marić’s encoder, extracted from Isabelle/HOL code,⁶ took 437s to output a CNF. To improve encoding performance, Marić wrote a C++ encoder whose code was manually compared against the Isabelle/HOL specification. We do not need to resort to an unverified implementation.

As for the encodings, ours took 28s to solve, while the Marić encoding took 787s (both using `cadical`). This difference is likely accounted for by the relative size of the encodings, in particular their symmetry breaking strategies. For $k = 6$ and n points, the encoding of Heule and Scheucher uses $O(n^4)$ clauses, whereas the one of Marić uses $O(n^6)$ clauses. They are based on different ideas: the former as detailed in Section 5, whereas the latter on nested convex hulls. The different approaches have been discussed by Scheucher [32]. This progress in solve times represents an encouraging state of affairs; we are optimistic that if continued, it could lead to an eventual resolution of $g(7)$.

Further differences include what exactly was formally proven. As with most work in this area, we use the combinatorial abstraction of triple orientations. We and Marić alike show that point sets in \mathbb{R}^2 satisfy orientation properties (Section 4). However, our work goes further in building the connection between geometry and combinatorics: our definitions of convexity and emptiness (Section 3), and consequently the theorem statements, are geometric ones based on convex hulls as defined in Lean’s `mathlib` [29]. In contrast, Marić axiomatizes these properties in terms of σ . A skeptical reviewer must manually verify that these combinatorial definitions correspond to the desired geometric concept.

A final point of difference concerns the verification of SAT proofs. Marić fully reconstructs some of the SAT proofs on which his results depend, though not the main one for $g(6)$, in an NbE-based proof checker for Isabelle/HOL. We make no such attempt for the time being,

⁵ This modification was performed by an author who did not understand this part of the proof, nevertheless having full confidence in its correctness thanks to the Lean kernel having checked every assertion.

⁶ We used Isabelle/HOL 2016. Porting the encoder to more recent versions of the prover would require broader adaptations due to breaking changes in the HOL theories.

instead passing our SAT proofs through the formally verified proof checker `cake_lpr` [41] and asserting unsatisfiability of the CNF as an axiom in Lean. Thus we trust that the CNF formula produced by the verified Lean encoder is the same one whose unsatisfiability was checked by `cake_lpr`.

8 Concluding Remarks

We have proved the correctness of the main result of Heule and Scheucher [21], implying $h(6) \leq 30$. Given that the lower bound $h(6) > 29$ can be checked directly (see [21]), we conclude the result $h(6) = 30$ is indeed correct. We believe this work puts a *happy ending* to one line of research started by Klein, Erdős and Szekeres in the 1930s. Prior to formalization, the result of Heule and Scheucher relied on the correctness of various components of a highly sophisticated encoding that are hard to validate manually. We developed a significant theory of combinatorial geometry that was not present in `mathlib`. Beyond the main theorem presented here, we showed how our framework can be used for other related theorems such as $g(6) = 17$, and we hope it can be used for proving many further results in the area.

Our formalization required approximately 300 hours of work over 3 months by researchers with significant experience formalizing mathematics in Lean. The final version of our proofs consists of approximately 4.7k lines of Lean code; about 26% are lemmas that should be moved to upstream libraries, about 40% develops the theory of orientations in plane geometry, and the remaining 34% (1550 LOC) validates the symmetry breaking and SAT encoding.

We substantially simplified the symmetry-breaking argument presented by Heule and Scheucher, and derived in turn from Scheucher [32]. Moreover, we found a small error in their proof, as their transformation uses the mapping $(x, y) \mapsto (x/y, -1/y)$, and incorrectly assumes that x/y is increasing for points in CCW-order, whereas only the slopes y/x are increasing. Similarly, we found a typo in the statement of the `Lex order` condition that did not match the `(correct)` code of Heule and Scheucher. Our formalization corrects this.

Future Work. We hope to formally verify the result $h(7) = \infty$ due to Horton [23], and other results in Erdős-Szekeres style problems.

We also want to improve the trust story of importing “cube and conquer”-style results into an ITP. Importing these kinds of proofs is a significant engineering task when the proof certificate is hundreds of terabytes in size, as it was for this result (see Section 6.1). Although we are confident that our results are correct, more work needs to be done to strengthen the trust in this connection point.

References

- 1 A. Biere, M. Heule, H. van Maaren, and T. Walsh. *Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications*. IOS Press, NLD, 2009.
- 2 Joshua Brakensiek, Marijn Heule, John Mackey, and David Narváez. The Resolution of Keller’s Conjecture, 2023. [arXiv:1910.03740](https://arxiv.org/abs/1910.03740).
- 3 Curtis Bright, Kevin K. H. Cheung, Brett Stevens, Ilias S. Kotsireas, and Vijay Ganesh. A SAT-based resolution of Lam’s Problem. In *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021*, pages 3669–3676. AAAI Press, 2021. doi:10.1609/AAAI.V35I5.16483.
- 4 Shawn T. Brown, Paola Buitrago, Edward Hanna, Sergiu Sanielevici, Robin Scibek, and Nicholas A. Nystrom. *Bridges-2: A Platform for Rapidly-Evolving and Data Intensive Research*, pages 1–4. Association for Computing Machinery, New York, NY, USA, 2021.
- 5 Davide Castelvecchi. Mathematicians welcome computer-assisted proof in ‘grand unification’ theory. *Nature*, 595(7865):18–19, June 2021. doi:10.1038/d41586-021-01627-2.

- 6 Cayden Codel, Marijn J. H. Heule, and Jeremy Avigad. Verified Encodings for SAT Solvers. In Alexander Nadel and Kristin Yvonne Rozier, editors, *Proceedings of the 23rd conference on Formal Methods In Computer-Aided Design*, 2023.
- 7 James Crawford, Matthew Ginsberg, Eugene Luks, and Amitabha Roy. Symmetry-breaking predicates for search problems. In *Proc. KR'96, 5th Int. Conf. on Knowledge Representation and Reasoning*, pages 148–159. Morgan Kaufmann, 1996.
- 8 Luís Cruz-Filipe, João Marques-Silva, and Peter Schneider-Kamp. Formally Verifying the Solution to the Boolean Pythagorean Triples Problem. *J. Autom. Reason.*, 63(3):695–722, October 2019. doi:10.1007/s10817-018-9490-4.
- 9 Luís Cruz-Filipe and Peter Schneider-Kamp. Formally Proving the Boolean Pythagorean Triples Conjecture. In Thomas Eiter and David Sands, editors, *LPAR-21. 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, volume 46 of *EPiC Series in Computing*, pages 509–522. EasyChair, 2017. doi:10.29007/jvdj.
- 10 Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The Lean Theorem Prover (System Description). In Amy P. Felty and Aart Middeldorp, editors, *Automated Deduction - CADE-25*, pages 378–388, Cham, 2015. Springer International Publishing.
- 11 Théo Delemazure, Tom Demeulemeester, Manuel Eberl, Jonas Israel, and Patrick Lederer. Strategyproofness and proportionality in party-approval multiwinner elections. In Brian Williams, Yiling Chen, and Jennifer Neville, editors, *Thirty-Seventh AAAI Conference on Artificial Intelligence, AAAI 2023, Thirty-Fifth Conference on Innovative Applications of Artificial Intelligence, IAAI 2023, Thirteenth Symposium on Educational Advances in Artificial Intelligence, EAAI 2023, Washington, DC, USA, February 7-14, 2023*, pages 5591–5599. AAAI Press, 2023. doi:10.1609/AAAI.V37I5.25694.
- 12 Paul Erdős and George Szekeres. On some extremum problems in elementary geometry. *Ann. Univ. Sci. Budapest. Eötvös Sect. Math.*, 3(4):53–62, 1960.
- 13 Paul Erdős and György Szekeres. A combinatorial problem in geometry. *Compositio Mathematica*, 2:463–470, 1935. URL: <http://eudml.org/doc/88611>.
- 14 Stefan Felsner and Helmut Weil. Sweeps, arrangements and signotopes. *Discrete Applied Mathematics*, 109(1):67–94, April 2001. doi:10.1016/S0166-218X(00)00232-8.
- 15 Tobias Gerken. Empty Convex Hexagons in Planar Point Sets. *Discrete & Computational Geometry*, 39(1):239–272, March 2008. doi:10.1007/s00454-007-9018-x.
- 16 Sofia Giljegård and Johan Wennerbeck. Puzzle Solving with Proof. Master’s thesis, Chalmers University of Technology, 2021.
- 17 W. T. Gowers, Ben Green, Freddie Manners, and Terence Tao. On a conjecture of Marton, 2023. arXiv:2311.05762.
- 18 Andrew Haberlandt, Harrison Green, and Marijn J. H. Heule. Effective auxiliary variables via structured reencoding. In *26th International Conference on Theory and Applications of Satisfiability Testing (SAT 2023)*, volume 271 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 11:1–11:19, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.SAT.2023.11.
- 19 Heiko Harborth. Konvexe Fünfecke in ebenen Punktmengen. *Elemente der Mathematik*, 33:116–118, 1978. URL: <http://eudml.org/doc/141217>.
- 20 Marijn J. H. Heule, Oliver Kullmann, and Victor W. Marek. *Solving and Verifying the Boolean Pythagorean Triples Problem via Cube-and-Conquer*, pages 228–245. Springer International Publishing, 2016. doi:10.1007/978-3-319-40970-2_15.
- 21 Marijn J. H. Heule and Manfred Scheucher. Happy ending: An empty hexagon in every set of 30 points, 2024. arXiv:2403.00737.
- 22 Andreas F Holmsen, Hossein Nassajian Mojarrad, János Pach, and Gábor Tardos. Two extensions of the erdős-szekeres problem. *arXiv preprint arXiv:1710.11415*, 2017.
- 23 J. D. Horton. Sets with No Empty Convex 7-Gons. *Canadian Mathematical Bulletin*, 26(4):482–484, 1983. doi:10.4153/CMB-1983-077-8.

- 24 Donald E. Knuth. Axioms and Hulls. In Donald E. Knuth, editor, *Axioms and Hulls*, Lecture Notes in Computer Science, pages 1–98. Springer, Berlin, Heidelberg, 1992. doi:10.1007/3-540-55611-7_1.
- 25 Boris Konev and Alexei Lisitsa. A SAT Attack on the Erdos Discrepancy Conjecture, 2014. arXiv:1402.2184.
- 26 Peter Lammich. Efficient Verified (UN)SAT Certificate Checking. *Journal of Automated Reasoning*, 64(3):513–532, March 2020. doi:10.1007/s10817-019-09525-z.
- 27 Filip Marić. Formal verification of a modern SAT solver by shallow embedding into Isabelle/HOL. *Theor. Comput. Sci.*, 411(50):4333–4356, 2010. doi:10.1016/J.TCS.2010.09.014.
- 28 Filip Marić. Fast formal proof of the Erdős-Szekeres conjecture for convex polygons with at most 6 points. *J. Autom. Reason.*, 62(3):301–329, 2019. doi:10.1007/S10817-017-9423-7.
- 29 The mathlib Community. The Lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, POPL '20. ACM, January 2020. doi:10.1145/3372885.3373824.
- 30 Carlos M. Nicolas. The Empty Hexagon Theorem. *Discrete & Computational Geometry*, 38(2):389–397, September 2007. doi:10.1007/s00454-007-1343-6.
- 31 Duckki Oe, Aaron Stump, Corey Oliver, and Kevin Clancy. Versat: A Verified Modern SAT Solver. In Viktor Kuncak and Andrey Rybalchenko, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 363–378, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.
- 32 Manfred Scheucher. Two disjoint 5-holes in point sets. *Computational Geometry*, 91:101670, December 2020. doi:10.1016/j.comgeo.2020.101670.
- 33 Sarek Høverstad Skotåm. CreuSAT, Using Rust and Creusot to create the world’s fastest deductively verified SAT solver. Master’s thesis, University of Oslo, 2022. URL: <https://www.duo.uio.no/handle/10852/96757>.
- 34 Leila Sloman. ‘A-Team’ of Math Proves a Critical Link Between Addition and Sets. <https://www.quantamagazine.org/a-team-of-math-proves-a-critical-link-between-addition-and-sets-20231206/>, December 2023.
- 35 Bernardo Subercaseaux and Marijn J. H. Heule. The Packing Chromatic Number of the Infinite Square Grid is 15. In Sriram Sankaranarayanan and Natasha Sharygina, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 29th International Conference, TACAS 2023, Held as Part of ETAPS 2022, Proceedings, Part I*, volume 13993 of *Lecture Notes in Computer Science*, pages 389–406. Springer, 2023. doi:10.1007/978-3-031-30823-9_20.
- 36 Bernardo Subercaseaux, John Mackey, Marijn J. H. Heule, and Ruben Martins. Minimizing pentagons in the plane through automated reasoning, 2023. arXiv:2311.03645.
- 37 Bernardo Subercaseaux, Wojciech Nawrocki, James Gallicchio, Cayden Codel, Mario Carneiro, and Marijn J. H. Heule. EmptyHexagonLean. Software, swbId: swb:1:dir:29dc0e7145296997bcb1230b4e03cd14c8d75617 (visited on 2024-08-23). URL: <https://github.com/bsubercaseaux/EmptyHexagonLean/tree/itp2024>.
- 38 Andrew Suk. On the erdős-szekeres convex polygon problem. *Journal of the American Mathematical Society*, 30(4):1047–1053, 2017.
- 39 George Szekeres and Lindsay Peters. Computer solution to the 17-point Erdős-Szekeres problem. *The ANZIAM Journal*, 48(2):151–164, 2006. doi:10.1017/S144618110000300X.
- 40 George Szekeres and Lindsay Peters. Computer solution to the 17-point erdős-szekeres problem. *The ANZIAM Journal*, 48(2):151–164, 2006.
- 41 Yong Kiam Tan, Marijn J. H. Heule, and Magnus O. Myreen. Verified Propagation Redundancy and Compositional UNSAT Checking in CakeML. *International Journal on Software Tools for Technology Transfer*, 25(2):167–184, April 2023. doi:10.1007/s10009-022-00690-y.
- 42 Mark Walters. It Appears That Four Colors Suffice : A Historical Overview of the Four-Color Theorem, 2004. URL: <https://api.semanticscholar.org/CorpusID:14382286>.
- 43 Nathan Wetzler, Marijn J. H. Heule, and Warren A. Hunt. DRAT-trim: Efficient checking and trimming using expressive clausal proofs. In Carsten Sinz and Uwe Egly, editors, *Theory and Applications of Satisfiability Testing – SAT 2014*, pages 422–429, Cham, 2014. Springer International Publishing.

Defining and Preserving More C Behaviors: Verified Compilation Using a Concrete Memory Model

Andrew Tolmach ✉ 

Portland State University, OR, USA

Chris Chhak ✉

Portland State University, OR, USA

Sean Anderson ✉ 

Portland State University, OR, USA

Abstract

We propose a concrete (“pointer as integer”) memory semantics for C that supports verified compilation to a target environment having simple “public vs. private” data protection based on tagging or sandboxing (such as the WebAssembly virtual machine). Our semantics gives definition to a range of legacy programming idioms that cause undefined behavior in standard C, and are not covered by existing verified compilers, but that often work in practice. Compiler correctness in this context implies that target programs are secure against all control-flow attacks (although not against data-only attacks). To avoid tying our semantics too closely to particular compiler implementation choices, it is parameterized by a novel form of oracle that non-deterministically chooses the addresses of stack and heap allocations. As a proof-of-concept, we formalize a small RTL-like language and verify two-way refinement for a compiler from this language to a low-level machine and runtime system with hardware tagging. Our Coq formalization and proofs are provided as supplementary material.

2012 ACM Subject Classification Security and privacy → Logic and verification; Software and its engineering → Compilers; Software and its engineering → Semantics

Keywords and phrases Compiler verification, C language semantics, Coq proof assistant

Digital Object Identifier 10.4230/LIPIcs.ITP.2024.36

Supplementary Material *Software*: <https://github.com/hope-pdx/MoreC.git>

Funding Work supported by the National Science Foundation under Grant No. 2048499.

1 Introduction

Undefined memory behaviors (UBs) in C programs, notably buffer overflows, are a major source of bugs and security exploits in real world systems. One approach to this problem is to detect or prevent memory UBs at runtime, turning silent violations into observable failures. A wide variety of hardware and software mechanisms have been proposed to achieve this [11, 29, 30, 12, 18, 42, 33, 13]. These mechanisms offer complex and non-obvious trade-offs between execution overhead and implementation complexity on one hand, and precision and reliability of memory safety enforcement on the other. For example, we might naively wish to trap all violations of spatial and temporal safety [37] at the granularity of individual C memory objects, but this may cause unacceptable execution overhead; moreover, it may break running systems, because lots of legacy C code relies on UB idioms that (usually) work in practice. Thus, we may be driven to employ more coarse-grained protection.

One simple idea is to classify all in-memory data as either public or private [41]. *Public data* consists of stack-allocated arrays and variables whose addresses are taken, heap objects allocated by `malloc`, and globals. *Private data* includes everything else the compiler puts in memory: control information such as saved return addresses or heap metadata, unaddressable



© Andrew Tolmach, Chris Chhak, and Sean Anderson;
licensed under Creative Commons License CC-BY 4.0

15th International Conference on Interactive Theorem Proving (ITP 2024).

Editors: Yves Bertot, Temur Kutsia, and Michael Norrish; Article No. 36; pp. 36:1–36:20

Leibniz International Proceedings in Informatics



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

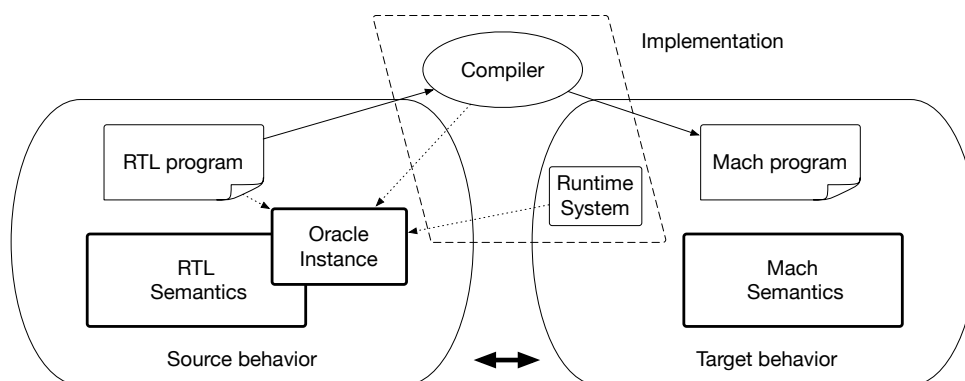
variables, function parameters and return values, spilled temporaries, etc. *Public vs. Private (PvP) protection* enforces that program loads and stores via C data pointers can touch only public data; private data is protected and accessed only by (trustworthy) compiler-generated or runtime system code. However, no protection boundaries are enforced between different pieces of public data; e.g., an out-of-bounds store to one array might overwrite the contents of another array. PvP can be implemented by a variety of techniques, including sandboxing or tagging (see §2). It is essentially the memory model provided by the WebAssembly virtual machine [14], and it has been proposed as the minimal “backstop” protection model for the Tagged C system of security policy enforcement [9, 2].

When properly used by a compiler, PvP (combined with a mechanism for preventing corruption or forging of function pointers) should suffice to prevent all *control-flow attacks* [37] i.e., it should be impossible for a compiled program to corrupt its own control flow. It would be natural to formalize this intuition as part of the specification and verification of compiler correctness. But conventional C compilers, including the CompCert verified compiler [20], make no guarantees at all about programs that exhibit UB. Indeed, compilers like gcc and Clang notoriously take advantage of the assumption that “UB cannot happen” to perform aggressive optimizations that often surprise programmers and can lead to serious security vulnerabilities [43, 36]. Hence, if we want to use compiler semantic preservation to characterize the security guarantees provided by PvP, we need to start from a source semantics that defines more memory behaviors than standard C.

To this end, we describe the design of a *concrete* memory semantics for C and a strategy for verified compilation from C with this semantics to a target machine and runtime system that enforce PvP protections. The concrete semantics treats (data) pointers as word-size machine integers. Load and store are well-defined at every address (i.e., every integer), possibly as an explicit *out-of-bounds (OOB)* failstop. Memory is finite, so *out-of-memory (OOM)* failstops are also possible. Since pointers are integers, the concrete semantics supports arbitrary arithmetic operations over them, including many useful low-level programming idioms that are UB in ISO standard C (see §3).

Using concrete pointers distinguishes us from most verified C compiler efforts, which follow the pioneering example of CompCert [22] by modeling pointers as a pair of abstract block identifier and offset. CompCert’s use of a single, abstract, infinite memory model across all compiler phases considerably eases verification; making do without this is part of the challenge we take up in this work. More fundamentally, changing to a concrete pointer semantics inhibits some useful compiler optimizations (see §3), which may make compiled code run more slowly. The payoff is that we significantly widen the set of source programs for which a verified compiler guarantees to preserve correct control-flow behavior.

At the same time, we don’t want our C semantics to be *too* concrete: it obviously should not depend on the choice of PvP enforcement mechanism or the details of the compiler or runtime system. This goal motivates the most novel aspect of our approach, which is the treatment of memory allocation. The concrete source semantics must assign a well-defined integer address to each stack and heap object. These integers must not change during compilation – otherwise, semantics preservation would break. But stack frame offsets are not determined until late in the compilation pipeline, and heap locations not until the `malloc` implementation executes at run time. So the source semantics must somehow predict where each object will actually live by using the details of exactly how the compiler and runtime work. Yet putting these details in the semantics itself would pollute it horribly, making the source language definition implementation-dependent. Our solution is to parameterize the source semantics with an *oracle* – essentially an external source of non-determinism – which



■ **Figure 1** Proof Architecture. Dotted lines represent information dependencies; the thick double-headed arrow is bi-directional refinement.

it consults to obtain the addresses of allocated objects. The oracle is packaged into a memory model with an abstract interface equipped with a small set of axioms that characterize its behavior, which is carefully designed to be independent of the PvP enforcement mechanism. The C semantics can be understood (and used to reason about C program behavior) based just on the memory model interface and axioms; it is entirely independent of how the oracle is instantiated. For any given *implementation* (compiler and runtime system for a particular PvP target), we will be able to construct a corresponding memory model *instantiation* that validates the model’s axioms. A proof of semantic preservation for a compiler connects the behavior of the source semantics equipped with a particular instantiation to the behavior of the corresponding implementation (see Figure 1). The **design and axiomatization of the oracular memory model** is our first technical contribution (§4).

Our second technical contribution is a **proof-of-concept verified compiler**, fully formalized in Coq [10], that prototypes key aspects of our approach (§5). We start by formalizing the oracular memory model’s interface and axiomatic properties. We then define a small toy source language RTL with functions and concrete memory, and give it a semantics parameterized by the memory model. We define a simple RISC-like target machine Mach and runtime system with tag-based memory protection, and a compiler from source to target. Then we instantiate the memory model so that its allocation behavior matches the target implementation and verify that the instantiation obeys the axioms. Finally, we prove bi-directional refinement between the source semantics, operating under that instantiation, and the target implementation. Much of our formal framework is borrowed from CompCert, but our memories use concrete addresses rather than abstract blocks and offsets, and, unlike in CompCert, the source and target model memory quite differently.

Determinizing the memory behavior of RTL with an oracle introduces some technical complexity in our proofs, but having a deterministic source language lets us use a forward simulation to prove semantic equivalence, which is much easier to construct than a backward one. As usual in CompCert [21], we rely on determinism of the target language Mach to derive a backward simulation automatically from the forward one. In a full compiler, RTL might itself be the target language for an earlier pass to be verified by forward simulation, giving another reason for wanting it to be deterministic. Moreover, defining an explicit oracle lets us prove that the memory model axiomatization is consistent.

We view this work as just an initial step in a larger project of building a “boring” [4] but fully secure C compiler. In particular, this paper considers only data pointers; function pointer protection is also essential to guarantee correct control flow (see §7). More broadly, to

get formal guarantees for the full range of programs accepted by standard compilers, we need a source semantics from which *all* UB has been removed (including non-memory UB such as integer overflow). Ultimately, we would like to show a secure compilation property [32] for systems that link C code against arbitrary machine code restricted to access only public data.

2 Background: PvP Enforcement

Our memory oracle can be instantiated using a wide range of PvP mechanisms, both hardware and software. These fall into two main categories: tagging and sandboxing. Our formal proof-of-concept compiler development assumes a tagged target, but could readily be retargeted for sandboxing.

Tagging mechanisms adjoin a metadata tag to each byte (or chunk of adjacent bytes) in memory, and to each pointer, and compare them at load and store operations. A one-bit tag suffices to distinguish public from private addresses. Using tagging lets the compiler keep the ordinary unified single-stack layout, with public and private data interleaved freely within each stack frame. Similarly, the runtime system can use tags to protect private metadata (e.g. block lengths, free list pointers, etc.) appearing in heap block headers or within unused blocks, as is common in conventional heap allocator implementations. To be secure and reasonably efficient, tagging requires hardware support, which is becoming increasingly common, e.g. via ARM MTE [34, 3, 24] or the PIPE ISA extension [13, 1].

Sandboxing places all public data in a contiguous region of memory and then forces all public loads and stores to lie within that region. It has been widely used for browser protection [44] and has recently been popularized as part of the WebAssembly virtual machine [14]. To use sandboxing, the compiler must adopt a non-standard memory layout that places just the public parts of the stack and heap in the sandbox. While direct hardware support for sandboxing is possible (e.g. via x86 segment-based addressing [44]), it is often implemented using software instrumentation. The simplest approach is just to add explicit bounds checks around each public access; this is expensive, but gives a hard failstop in the event of an error. Software Fault Isolation (SFI) [40, 17] is a cheaper alternative based on the assumption that the sandbox size and base address alignment have the form 2^n ; then any arbitrary integer can be “warped” to an address within the sandbox by zeroing all but its low-order n bits and or-ing in the base address. OOB accesses do not failstop, but memory outside the sandbox is never corrupted.

3 Concrete Memory Semantics for C

In this section, we sketch the form of concrete memory semantics we envisage for C, and examine some of its consequences. Public memory is structured into *regions*, which are used to store scalars, `structs`, or `unions` whose address is (potentially) taken, arrays, and `malloced` heap objects. Region allocation and deallocation can occur either implicitly, e.g. for locals during function entry/exit and for globals at program start, or explicitly, for heap data managed by `malloc` and `free` calls in program code. The semantics assigns a concrete integer address to the base of each allocated region by consulting an *oracular memory model*, which is a parameter of the semantics.

The semantics makes no distinction between data pointers and word-sized machine integers. Thus pointers support the same arithmetic and bitwise operations as ordinary integers, and casts between pointers and integers are semantic no-ops. Loads and stores

```

void *memmove(void *s1,
              const void *s2,
              size_t n) {
  char *dest = (char *) s1;
  const char *src = (const char *) s2;
  if (dest <= src) // UB in standard C
    while (n--)
      *dest++ = *src++;
  else {
    src += n;
    dest += n;
    while (n--)
      *--dest = *--src;
  }
  return s1;
}

extern char hash(void *p);
int main() {
  int *p = (int *) malloc(sizeof(int));
  *p = 0;
  int *q = // UB in standard C
    (int *) ((uintptr_t) p | (hash(p) & 0xF));
  int *r = (int *) (((uintptr_t) q >> 4) << 4);
  return *r;
}

void bad() {
  char p[4], q[4]; // may failstop with OOM
  q[2] = 0;
  p[6] = 1;        // may failstop with OOB
  print(q[2]);    // if reached, prints number
}

```

■ **Figure 2** Concrete pointer examples (adapted from [6, 19]).

through *any* integer that points into a public region succeed and obey the usual “good variables” properties (i.e., a load from a given location returns the value most recently stored there). Loads or stores through an integer that points at an *unallocated* location might not obey the good variables properties, and might also cause execution to failstop, again based on a decision by the oracle. Accesses that would make the implementation halt due to a PvP violation or an unmapped page fault should correspond to source OOB failstops; those that the implementation deems harmless or “warps” into allocated locations can be allowed. At a minimum, for the semantics preservation theorems to hold, the oracle must refuse to overwrite any location that contains private compiler-generated data. But the source semantics doesn’t know anything about private data, just that accesses outside allocated regions are unreliable.

Figure 2 illustrates some code that has memory-related UB in ISO standard C and in CompCert, but is well defined in our concrete semantics. Function `main` performs bit-level operations on the representation of `p`, “stealing” the low order 4 bits to hold a hash code; it relies on the result of `malloc` being 16-byte aligned, so the low order bits can be safely zeroed again before the pointer is used. Function `memmove` works even when source and target buffers overlap; it relies on making a comparison between `dest` and `src`. Function `bad` performs an out-of-bounds write; in our semantics, depending on where `p` and `q` live in the stack frame, the code will either failstop with OOB on the assignment to `p[6]`, or continue and print some number (0 or 1 in any reasonable instantiation of our memory model) – but it won’t behave *arbitrarily*.

Out of Memory. Since actual machine memory is finite and pointers are represented as machine integers with limited range [28], any oracle instance will need to refuse some allocation requests due to stack overflow or heap exhaustion. Running out of memory causes execution to enter a *failstop* state. This is different from *getting stuck*, which corresponds to a UB for which the compiled code can do anything; failstopping is a well-defined behavior that must be preserved by compiled code. Thus a target will always run out of memory when the source does; this is essential for the preservation of safety properties (since otherwise the target could perform a bad behavior even though the source failstops).

To avoid polluting its axiomatization with implementation-specific details about memory consumption, we allow the oracle to fail with OOM at any time. To gain confidence that our compiler correctness results are not vacuous, we have also proven reverse refinement

```

extern void f(int *p);
void g(int c) {
    int x[10], y[10], u, v, w;
    x[0] = 42; y[0] = 99; u = x[0]; // optimize to u = 42 ?
    y[c] = 99; v = x[0]; // optimize to v = 42 ?
    f(y); w = x[0]; // optimize to w = 42 ?
}

```

■ **Figure 3** Potential Redundant Load Optimizations.

from target to source – showing that the source runs out of memory *only* when the target does. We can confirm that the resulting allocation behavior is reasonable by inspecting the (completely concrete) target semantics and the generated target code.

Optimization. Using concrete semantics inevitably limits a compiler’s optimization opportunities. Most significantly, any store to an unknown location and any function call potentially overwrites arbitrary public locations. (Private data, including ordinary scalars whose addresses are not taken, *are* preserved across unknown stores and function calls just as in ordinary C semantics.) This invalidates non-aliasing analyses, and so can prevent removal of some redundant public data loads and stores and also reduce the applicability of register promotion optimizations [25].

For example, in the code in Figure 3, all the suggested redundant load optimizations are valid in standard C (and performed by `gcc` and Clang): the first two because the stores to `y` can be assumed to be in bounds and the last because the address of `x` does not escape to `f`. But only the first is valid in our semantics, because in a concrete world, an out-of-bounds store to `y[c]` might indeed overwrite `x[0]`, and the unknown function `f` might overwrite *any* public location. The desire to maintain these optimizations while supporting more liberal pointer arithmetic has led to considerable work on hybrid models that allow some form of interoperation between block-based and concrete views [16, 15, 27]. Despite this, we are unaware of any empirical studies that assess how important alias-based optimizations actually are for real C workloads. We note that CompCert [23] doesn’t perform the second or third optimizations in Figure 3 either, although they are valid under its memory model.

A second limit on optimization is a subtle consequence of working with finite memory while preserving refinement [19]. In our oracular approach, the source semantics locates each in-memory object at exactly the same address as in the target implementation. If the compiler were allowed to remove an allocation operation as part of dead code elimination, the oracle would have no idea where to put the object! Nor could it just announce OOM, since then the source would failstop whereas the target might continue, violating forward refinement. The upshot is that the compiler cannot optimize away dead allocations; this is unfortunate, but perhaps not too important in practice (e.g., CompCert doesn’t currently perform such optimizations either).

Fortunately, many other optimizations involving allocations are still valid; in particular, CompCert-style function inlining and tail-call elimination should both be possible in our framework because they only adjust the locations of public stack allocations, not their total size. (Tail call elimination can also change the order of allocations and deallocations, but only for zero-sized blocks, which does not matter.)

4 Axiomatic Memory Model

The C memory model used by our concrete semantics gives an abstract characterization of *memories* that are modified and inspected using a set of *operator* functions. We now describe the model in detail, giving the signatures of the operators and a set of axioms specifying their behavior. The design goal for the axiomatization is to capture just those properties that are necessary for a C semantics and program logic to employ the memory model, while constraining possible instantiations as little as possible.

The model interface consists of an abstract type \mathcal{M} of memories, with the following constants and operators:

<code>initm</code>	::	\mathcal{M}	<code>hpFree</code>	::	$\mathcal{M} \rightarrow \mathcal{A} \rightarrow [\mathcal{M}]$
<code>stkAlloc</code>	::	$\mathcal{M} \rightarrow \mathcal{L} \rightarrow \mathcal{S} \rightarrow \mathcal{G} \rightarrow [(\mathcal{R}, \mathcal{M})]$	<code>perturb</code>	::	$\mathcal{M} \rightarrow \mathcal{L} \rightarrow [\mathcal{M}]$
<code>stkFree</code>	::	$\mathcal{M} \rightarrow [\mathcal{M}]$	<code>load</code>	::	$\mathcal{M} \rightarrow \mathcal{A} \rightarrow [\mathcal{V}]$
<code>hpAlloc</code>	::	$\mathcal{M} \rightarrow \mathcal{L} \rightarrow \mathcal{S} \rightarrow \mathcal{G} \rightarrow [(\mathcal{R}, \mathcal{M})]$	<code>store</code>	::	$\mathcal{M} \rightarrow \mathcal{A} \rightarrow \mathcal{V} \rightarrow [\mathcal{M}]$

Notation: These functions are all partial, as indicated by the fact that they return option types, where we write $[T]$ for $(\text{Option } T)$, $[t]$ for $(\text{Some } t)$ and \emptyset for None . Memories $M \in \mathcal{M}$ are byte-indexed. Addresses \mathcal{A} and sizes \mathcal{S} are non-negative integers. Values \mathcal{V} are machine bytes. Labels \mathcal{L} are an arbitrary type, explained further below. Alignments \mathcal{G} are positive integral powers of two. Regions $r \in \mathcal{R} = \mathcal{A} \times \mathcal{S}$ are half-open intervals with base $bs\ r$ and size $sz\ r$; we say a is in the *footprint* of r , written $a \in r$, iff $bs\ r \leq a < bs\ r + sz\ r$.

The basic meaning of most of these operations should be clear from their names and type signatures. Memory is organized into (*allocated*) *regions*, which are intended to hold public data. Stack regions are intended for public data associated with function activations, allocated and freed as part of compiler-generated function call/entry and exit/return code. Heap regions are intended for explicitly allocated storage, allocated and freed by the C library `malloc` and `free` calls; they are also used for globals allocated at program start-up time (and never freed). Formally, the only difference between them is how a region to be freed is identified: for the stack, it is implicitly the most recently allocated region; for the heap, it is explicitly specified by a region base address.

Load and store are single-byte operations taking a concrete address. This byte-based interface can be used as a foundation on which a semantics can build a higher-level interface supporting reads and writes of multi-byte types, assuming suitable functions for encoding and decoding these in terms of bytes, and additional alignment checks where needed [22].

Figure 4 gives the full axiomatization of the memory operators. Although the precise layout of memory is deliberately kept abstract, the axioms use two observation functions that expose information about the allocated regions. S_M is the stack of allocated stack regions in M , represented as a list in stack order (with most recently pushed region at the head). H_M is the set of the allocated heap regions in M , represented as an (unordered) list with no duplicates. We write A_M for $S_M ++ H_M$, the full (multi)set of allocated regions in M .

Further notation: We write $::$ for list cons, $[]$ for the empty list, \in for list membership, $++$ for list concatenation, and $L_1 \cong L_2$ if L_1 and L_2 are equal considered as multisets (i.e., are permutations of each other). M' is *value-stable on* M , written $M \lesssim M'$, iff $\forall a \in r \in A_M, \text{load } M\ a = \text{load } M'\ a$. We assume that machine addresses have w bits.

We start with the basic invariants on memories that are maintained by all operations. RWF specifies basic well-formedness conditions on regions. It implies that:

- (i) all addresses in a region (and the address immediately following) are representable in an unsigned machine word, allowing them to be computed using machine arithmetic;

$$\begin{array}{c}
 \frac{r \in A_M}{0 < bs \ r \leq bs \ r + sz \ r < 2^w - 1} \text{ (RWF)} \qquad \frac{A_M = rs_1 ++ r_1 :: rs_2 ++ r_2 :: rs_3}{\forall a. a \in r_1 \Rightarrow a \notin r_2} \text{ (RDISJ)} \\
 \\
 \frac{H_M = rs_1 ++ r_1 :: rs_2 ++ r_2 :: rs_3}{bs \ r_1 \neq bs \ r_2} \text{ (HPRDIST)} \qquad A_{\text{initm}} = [] \text{ (INIT)} \\
 \\
 \frac{\text{stkAlloc } M \text{ lbl } s \ g = \lfloor (r, M') \rfloor}{S_{M'} = r :: S_M \wedge H_{M'} \cong H_M} \text{ (STKA)} \qquad \frac{\text{hpAlloc } M \text{ lbl } s \ g = \lfloor (r, M') \rfloor}{H_{M'} \cong r :: H_M \wedge S_{M'} = S_M} \text{ (HPA)} \\
 \\
 \frac{\text{stkAlloc } M \text{ lbl } s \ g = \lfloor (r, M') \rfloor}{sz \ r \geq s \wedge (bs \ r) \bmod g = 0} \text{ (STKAR)} \qquad \frac{\text{hpAlloc } M \text{ lbl } s \ g = \lfloor (r, M') \rfloor}{sz \ r \geq s \wedge (bs \ r) \bmod g = 0} \text{ (HPAR)} \\
 \\
 \frac{\text{stkAlloc } M \text{ lbl } s \ g = \lfloor (r, M') \rfloor}{M \lesssim M'} \text{ (STKAV)} \qquad \frac{\text{hpAlloc } M \text{ lbl } s \ g = \lfloor (r, M') \rfloor}{M \lesssim M'} \text{ (HPAV)} \\
 \\
 \frac{S_M \neq []}{\text{stkFree } M \neq \emptyset} \text{ (STKFOK)} \qquad \frac{\text{hpFree } M \ a = \lfloor M' \rfloor}{\exists r \in H_M. bs \ r = a \wedge H_M \cong r :: H_{M'} \wedge S_{M'} = S_M} \text{ (HPF)} \\
 \\
 \frac{\text{stkFree } M = \lfloor M' \rfloor}{\exists r. S_M = r :: S_{M'} \wedge H_{M'} \cong H_M} \text{ (STKF)} \qquad \frac{r \in H_M}{\text{hpFree } M \ (bs \ r) \neq \emptyset} \text{ (HPFOK)} \\
 \\
 \frac{\text{stkFree } M = \lfloor M' \rfloor}{M' \lesssim M} \text{ (STK FV)} \qquad \frac{\text{hpFree } M \ a = \lfloor M' \rfloor}{M' \lesssim M} \text{ (HPFV)} \\
 \\
 \frac{\text{perturb } M \text{ lbl} = \lfloor M' \rfloor}{S_M = S'_M \wedge H_M \cong H'_M \wedge M \lesssim M' \wedge M' \lesssim M} \text{ (PERT)} \\
 \\
 \frac{a \in r \in A_M}{\text{store } M \ a \ v \neq \emptyset} \text{ (STOK)} \qquad \frac{a \in r \in A_M}{\text{load } M \ a \neq \emptyset} \text{ (LDOK)} \\
 \\
 \frac{\text{store } M \ a \ v = \lfloor M' \rfloor}{\text{load } M' \ a = \lfloor v \rfloor} \text{ (LDSTEQ)} \qquad \frac{\text{store } M \ a \ v = \lfloor M' \rfloor}{S_M = S'_M \wedge H_M \cong H'_M} \text{ (STR)} \\
 \\
 \frac{a \in r \in A_M \quad \text{store } M \ a \ v = \lfloor M' \rfloor \quad a' \neq a}{\text{load } M' \ a' = \text{load } M \ a'} \text{ (LDSTNEQ)}
 \end{array}$$

■ **Figure 4** Axiomatization of memory constants and operators.

(ii) the C NULL pointer, which we assume to be 0 (as is natural on almost all current machines), does not point into any region;

(iii) regions may be empty (have size 0), which can help avoid special cases in a semantics. RDISJ states the basic property that no address can be within the footprint of two allocated regions, which is essential for reasoning about separation. Combined with RWF it further implies that the total amount of allocated memory is bounded. Since the address passed to `hpFree` should unambiguously identify a single region, we impose a further invariant HPRDIST to guarantee this (and hence H_M is indeed a set).

The heap and stack are initially empty (INIT). A successful allocation pushes a new region onto the stack regions (STKA) or adds a new region to the heap (HPA), preserving the values in existing allocated regions (STKAV,HPAV). The new region is of at least the requested size and alignment (STKAR, HPAR). Standard `malloc` behavior can be obtained by using `hpAlloc` with the maximum required alignment for any primitive type (typically $w/4$, allowing the lower-order $\log_2 w - 2$ bits to be “stolen” for other uses).

Each allocation operation carries a *label* from some type \mathcal{L} , which is a parameter of the entire model. Labels are contextual clues that an instantiation can use to place allocations at the same locations as an actual target implementation. For example, if public stack allocation

occurs as part of function entry, the labels on `stkAlloc` might carry the name or definition of the function, giving an instantiation access to the size of the function’s private data and hence enabling it to calculate the location of the public region. Crucially, labels have no impact at all on the axiomatized behavior of the interface. In this sense, they do not affect the source semantics; they just make it possible to prove compiler semantic preservation for a particular choice of labeling scheme, instantiation, and target implementation.

The axioms make no guarantees about when `stkAlloc` or `HpAlloc` will succeed; as discussed in §3, they may return \emptyset at any time, to indicate OOM. On the other hand, `StkFree` is guaranteed to succeed iff the stack is non-empty (`STKFOK`, `STKF`) and `HpFree` succeeds iff the heap contains an allocated region at the specified address (`HPFOK`, `HPOK`). Again, these operations do not affect values in other allocated regions (`STKFV`, `HPFV`).

The `load` and `store` operations always succeed in allocated regions (`STOK`, `LDOK`). They *might* also succeed at unallocated addresses. It is important not to require failure on these accesses, because trapping them may be expensive or impossible in some implementations. The usual “good variables” properties hold for stores into an allocated region (`LDSTEQ`, `LDSTNEQ`). In fact, `LDSTEQ` holds even for addresses *outside* allocated regions, although this is not likely to be useful in practice. The restriction to allocated addresses for `LDSTNEQ` is essential to allow implementations like SFI masking that “warp” out-of-bounds stores into essentially unpredictable (but in-bounds) stores. Note that the axioms say nothing about initial values; that is, freshly allocated regions (and all unallocated addresses) start with defined but unpredictable contents.

Since all the operations (except `load`) return a potentially changed memory, they destroy any guarantees about the contents of *unallocated* locations, reflecting the fact that an implementation may change the layout or contents of private data at these points. But an implementation might also do this at other points that do not correspond to a source-level memory operation. For example, the stack pointer might change during function call/entry or exit/return sequences, even if no public data is allocated (or even accessed) at these points. To account for this, a source semantics can use the `perturb` pseudo-operation to signal places where such changes may occur in the target, without altering the status and contents of allocated data (`PERT`). A semantics should sprinkle `perturb` calls generously, as this allows a wider range of target implementations to support an instantiation of the model.

5 Verified Proof-of-Concept Compiler

We now describe the Coq implementation and verification of a proof-of-concept compiler and runtime based on the memory model. The implementation is coded in Coq’s internal functional language, Gallina. Our source and target languages are highly simplified, while still containing enough features to exercise the memory model and to illustrate the interesting and challenging parts of the semantics preservation proof.

The formalisation of the memory model closely follows the design given in §4, except that memory is 64bit-word-indexed rather than byte-indexed, so values \mathcal{V} are 64-bit words and pointers occupy a single location, and there are no alignments.

5.1 Source language

The source language (Figure 5, left) is a simple Register Transfer Language (RTL). A program is a list of named functions, each with an unbounded number of local pseudo-registers, jointly operating on a word-addressed memory. Registers and memory contain 64-bit machine words; there is absolutely no distinction between integers and pointers. Function call is an atomic operation passing all arguments at once, and the call stack is implicit. Since intrafunction control flow is unimportant for the compilation issues we wish to explore, we dispense with it altogether: function bodies are just straight-line instruction sequences.

36:10 Defining and Preserving More C Behaviors

r	$:=$ SP RV RA GP1 GP2 GP3	
\hat{i}	$:=$ mov $r_s r_d$	move register
	movi $n r_d$	move immediate
	mov& r_d	move array base
	op \oplus $r_{s_1} r_{s_2} r_d$	binary operation
	ld $r_a r_d$	load
	st $r_s r_a$	store
	call $f \vec{r} r_d$	call
\oplus	$:=$ + -	binary operator
f	$:= f(\vec{r}_a, n) = \vec{i} \text{ ret } r_r$	function
pr	$:= \vec{f}$	RTL program
r	$:=$ SP RV RA GP1 GP2 GP3	
\hat{i}	$:=$ mov $r_s r_d$	move register
	movi $n r_d$	move immediate
	op \oplus $r_{s_1} r_{s_2} r_d$	binary operation
	ld $p ofs(r_a) r_d$	load
	st $p t r_s ofs(r_a)$	store
	jal id	jump-and-link
	builtin b	builtin
p	$:=$ hi lo	privilege
t	$:=$ pro unpro	protection tag
b	$:=$ malloc free	builtin
f	$:= id : \vec{i}$	function
pr	$:= \vec{f}$	Mach program

■ **Figure 5** Syntax for RTL (left) and Mach (right).

There are two ways of allocating memory. For the **stack**, each function activation implicitly allocates a local array of statically fixed size. Just as with C local arrays or address-taken scalars, this storage can be used by the function itself and by callees who are given a pointer to it. For the **heap**, there are C-like built-in functions `malloc: words \rightarrow ptr` and `free: ptr \rightarrow void` that do explicit allocation and deallocation. The RTL semantics implements these features using calls to the corresponding memory model operations.

Function $f(\vec{r}_a, n) = \vec{i} \text{ ret } r_r$ takes its arguments in \vec{r}_a , allocates a local array of constant size n , executes the straight-line sequence of instructions \vec{i} , deallocates its local array, and returns the value in register r_r . Functions can use any number of additional local registers, which are implicitly initialized to 0. The `mov& r_d` instruction moves the base address of the local array into r_d . Other instructions should be self-explanatory.

Programs can have one of four behaviors:

- Terminate with a result value. The first function in the program is taken to be the program's entry point and its final return value is the program's overall result.
- Failstop with a memory error. This can be either a failed allocation (OOM) or an unsuccessful access to an unallocated memory address (OOB).
- Diverge. Despite the lack of control-flow instructions, this can happen via an infinite recursion, provided the functions involved have zero-size local arrays (otherwise the program will eventually run out of memory).
- Get stuck. This can only happen if the program calls a function that does not exist or has the wrong number of arguments. It is easy to define a static typing judgement that rules out such programs.

The formal semantics of RTL is given by a straightforward instruction stepping relation over machine states. We include an explicit `Failstop` state. The state maintains an implicit call stack of pending activations, including return addresses and local registers. A single memory is threaded throughout, with memory model operations invoked at the following places in the semantics:

- Function entry: `stkAlloc` is invoked to allocate the local array, passing the current code location (function id and remaining instructions) as the context label. If the allocation succeeds, the array base is remembered in the state for future use by the `mov&` instruction. If it fails, the machine enters the `Failstop OOM` state.

- Function exit: `stkFree` is invoked to deallocate the local array. We can use the `STKFOK` axiom to prove that this can never fail in this semantics.
- Function call and return: `perturb` is invoked (again with the code location as label) to reflect the fact that the target implementation may change the stack to pass arguments or clean up after a call; if either operation fails, the machine enters `Failstop OOM`.
- Execution of `malloc` built-in: `hpAlloc` is called with the requested number of words. If this fails, the machine enters `Failstop OOM`.
- Execution of `free` built-in: `hpFree` is called with the specified address; if this fails, the address must be bogus, and the machine enters `Failstop OOB`.
- Load and store: these are done directly by the memory model's `load` and `store` operations; if they fail, the machine enters `Failstop OOB`. Recall that accesses to allocated locations are guaranteed to succeed, but out-of-bounds accesses will not necessarily fail.

Although the memory model abstracts over the behavior of the allocation oracle, for any given instantiation of the memory model, RTL itself is deterministic.

5.2 Target machine

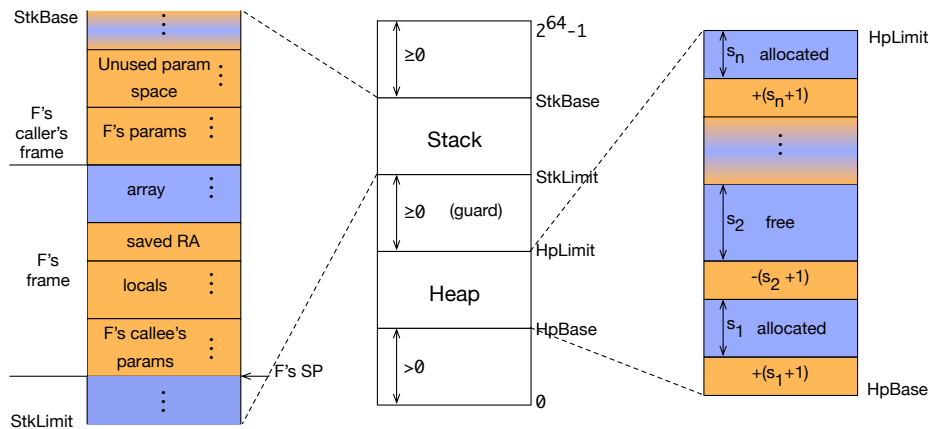
The compiler's target language (Figure 5, right) is a simple RISC-like assembly code (Mach). The instruction set is very similar to RTL, but operates over a small fixed set of registers, including a stack pointer (`SP`), return address register (`RA`), return value register (`RV`), and general-purpose registers (`GP1-3`). Registers and memory can contain either 64-bit integers or abstract code pointers (actually just sequences of instructions). Machine code lives separately from data memory, and is divided into named functions, the first of which is the program's entry point `main`. Again, for simplicity, each function is a straight-line sequence of instructions, and there are no intrafunction branch instructions. Functions are invoked by `jal`, which stores the "return address" (the remaining instructions in the caller) in `RA`. After the last instruction in a function, the machine "returns" by continuing execution at the "address" in `RA`. `RV` is intended to hold the result value of a function when it returns.

The machine's memory is modeled as a single 64bit-word-addressed array. In addition to a containing a value (the *payload*), each address has an associated single-bit *protection* tag, either *protected* or *unprotected*. Store operations set both the payload and the tag. Load and store operations are parameterized by a corresponding *privilege* flag: `hi` or `lo`. Low-privilege instructions can only access unprotected locations; high-privilege instructions can access all locations. Protection violations cause a failstop. This corresponds to a simplified version of ARM MTE [34, 3] or PIPE [13]. The initial values in memory are defined but arbitrary and are all tagged as `unpro`. The intention is that private data such as return addresses, arguments, and variables will be tagged `pro` and accessed with `hi` instructions, whereas public data such as the per-function arrays will be tagged `unpro` and accessed with `lo` instructions.

The overall structure of memory is specified by a small set of parameters defining the bases and limits of a *stack area* and a *heap area*, organized as shown in Figure 6. All loads and stores are constrained to lie within one of these areas, which correspond to the pages mapped by a process in a real machine. Accesses outside these areas cause failstops. The machine initializes `SP` to `StkBase`; thereafter, the stack is managed entirely by code generated by the compiler. The heap is managed by the `malloc` and `free` functions that, in a real system, would be part of the runtime system code. For simplicity (and because our target language is so impoverished) here we write these functions in Gallina instead and invoke them via special built-in machine instructions; see §5.4.

Mach programs are deterministic, and can have the same four kinds of behaviors as RTL programs. Termination is signaled by returning from `main`; the overall program result value is in `RV`. Failstops can be due either to memory errors (protection violation or out-of-bounds

36:12 Defining and Preserving More C Behaviors



■ **Figure 6** Layout of memory (center), stack frames (left), and heap (right); orange regions are **pro** and blue regions are **unpro**.

access) or a failure by a built-in function (out of memory in `malloc` or bad argument to `free`). Although real machines don't get stuck, this one can, by trying to jump to a function that doesn't exist or return when `RA` does not contain a code pointer; these are artifacts of our abstract view of code that would disappear if we used concrete code addresses.

5.3 Compilation scheme and stack layout

The compiler from RTL to Mach has a very simple structure: each function is translated independently, and each source instruction is translated to a fixed sequence of target instructions. The target code maintains a conventional stack of function activation frames, growing towards lower addresses. A function's frame (Figure 6, left) holds its parameters and local variables, its return address, and its local array; thus private and public data are interleaved. There is no attempt to perform register allocation; instead, each RTL parameter and pseudo-register is mapped to a fixed frame offset and loaded/stored each time it is used. We adopt a calling convention where the entire responsibility for constructing and freeing stack frames is given to the callee. The size of the frame is computed statically and the `SP` is adjusted just once at function entry and again at function exit. We view parameters to any callees as part of the *caller's* frame, which must include enough space to hold the maximum number of parameters needed by any call the function makes, computed by a simple pass over the function body. Since the sizes of all frame components are known statically, they can be accessed as offsets from the `SP`; there is no need for a separate frame pointer.

Initially, the entire memory is tagged **unpro**, but all components of stack frames except the local array should be tagged **pro**. The compiler generates function entry code to do this one word at a time, and matching exit code that retags everything **unpro**. Setting a tag also requires writing a payload value; both these operations write zeros. The dual approach of making **pro** be the default and unprotecting/reprotecting the array component would also work, but could be much less efficient, since arrays can be arbitrarily large. To protect non-stack memory, target code must failstop if the stack overflows its predefined area, i.e. if `SP` goes below `StkLimit`. We achieve this via the usual trick of placing a "guard" area of invalid addresses between the stack and the heap.

5.4 Runtime allocator and heap layout

As described in §5.2, the target machine is equipped with a runtime heap memory allocator packaged as C-like `malloc` and `free` functions. For simplicity and convenience, we build this code into the machine’s semantics, but in a real implementation it would be part of the runtime system; our Gallina code is “honest” in the sense that it uses only memory operations already provided by the machine’s instruction set. We use a very simple heap organization (Figure 6, right). The heap region is divided into objects, each consisting of a one word header followed by zero or more words of data. The header is an integer whose absolute value is the size of the object (including the header itself), and whose sign indicates whether the object is currently allocated (sign +1) or free (sign -1). Headers are tagged `pro`; all other head addresses are tagged `unpro`. Initially, the heap consists of a single free object.

To allocate a new heap region of size $s \geq 0$, `malloc` loops over the objects in increasing address order, starting at `HpBase`, until it finds a free object of size $n \geq s$. It returns that object’s address (the first word above the header), after using a privileged write to flip the sign bit in the header to mark the object as allocated. If $n > s$, the object is first split into two objects, which involves changing the size in the header of the existing object and transforming a data word into a header for the second object, again using privileged writes. If no sufficiently large object is found before the loop reaches `HpLimit`, `malloc` failstops.

To deallocate a heap region at a specified address, `free` must operate similarly: it loops over the objects in order starting at `HpBase` until it finds an allocated object at that address, and then uses a privileged write to flip the sign bit in that object’s header, marking it as free. The newly freed object is then coalesced with its neighbors on each side if they were already free. If no matching address is found, the specified address was invalid, so `free` failstops.

Obviously this scheme (especially for `free`) has poor efficiency properties compared to real C memory manager implementations. But it does illustrate a realistic interleaving of (public) data and (private) allocator metadata in memory, and shows how a tag-based machine can protect the metadata, while keeping proofs fairly simple. Extending the algorithms to manage multiple free lists, etc. would be straightforward but tedious. Implementing a more realistic $O(1)$ time `free` safely could be done by storing and protecting metadata separately.

5.5 Memory model instantiation

We now describe how the memory model (§4) can be instantiated so that the addresses it assigns to stack and heap allocations requested by the source semantics (§5.1) match those generated by compiled code (§5.3) and the runtime heap allocator (§5.4). This is the most novel aspect of our approach. In essence, we work backwards from the compiler and runtime design to identify (just) those aspects of target state that affect public memory layout, and use these to define instantiations of the \mathcal{M} type and the model’s operations and observation functions that validate the model’s axioms. The semantics preservation proof (§5.6) confirms that we have done so correctly for this source and target.

The construction of this instantiation is parameterized by an RTL source program, which is consulted to obtain the definitions of functions passed as labels to `StkAlloc`, from which the oracle can calculate private stack data sizes. The instantiation itself is then passed as a parameter to the definition of the RTL semantics. For the proof-of-concept system, we define $\mathcal{M} = \{m : (vals : Vals) \times (st : St) \times (hp : Hp) \mid MInv\ m\}$, i.e. a Coq subset type containing a triple of value map `vals`, stack `st`, and heap `hp`, which obeys an invariant `MInv`. We discuss each component in turn.

36:14 Defining and Preserving More C Behaviors

Values. To support the model’s `load` and `store` operations, the instantiation simply maintains a map *vals* of type $Vals = \mathcal{A} \rightarrow \mathcal{V}$ that contains the correct values for all public (`unpro`) locations, and is arbitrary at other locations.

Stack. The source semantics uses stack allocation operations only for the local array created as part of each function entry sequence; the model instantiation must predict the runtime location of this array. To do this, the instantiation maintains an abstract stack $st : St$ of function activations kept in one-to-one correspondence with the frames of the concrete implementation stack. Here St is lists of abstract *frames*; each invocation of `stkAlloc` with size s and label f pushes a new frame (f, s) onto st . The instantiation uses the function definition for f and knowledge of how the compiler lays out concrete frames (Figure 6) to compute the array address to return (or \emptyset if this address is below `StkLimit`).

Each `stkFree` pops the top frame from st . Since the implementation retags private frame locations as `unpro` during function exit, resetting their payload values to 0, the instantiation must also zero the corresponding locations in *vals*.

In this particular implementation all the work associated with building and tearing down frames is done in the callee. Although the semantics invokes `perturb` operations on the caller side before and after each call, nothing happens to the concrete stack at these points, so `perturb M l` always returns $[M]$.

The observation function S_M is obtained simply by mapping over st and extracting the array base and size from each frame. The instantiation also defines a predicate identifying accessible (i.e. public) addresses in the stack area, which includes both addresses in allocated regions (i.e. local arrays) and those beyond the end of the stack pointer; `load` and `store` on inaccessible stack area addresses return \emptyset .

Heap. RTL’s invocations of `hpAlloc` and `hpFree` are in one-to-one correspondence with calls to the “runtime system” primitives `malloc` and `free`. Hence, the model instantiation simply maintains a slightly abstracted version $hp : Hp$ of the runtime’s heap data structure and executes abstracted versions of the runtime algorithms over it. Hp is lists of *objects* maintained in increasing address order, with the first object being understood to start at `HpBase`. Each object is described as a pair (s, af) where s is the object size and af is a boolean is-allocated flag. The heap accessibility predicate holds for addresses within both allocated and unallocated objects, but not the private header words. When the implementation of `free` performs coalescing, it will retag some private headers as `unpro` and reset their payload values to 0, so the model instantiation must also zero the corresponding locations in *vals*.

Invariant. By design, the type of \mathcal{M} allows very little “junk,” but we do need to carry two simple technical invariants in $MInv$: for st , the calculated SP must always lie within the stack area; for hp , the total size of the objects (including headers) always equals `HpLimit` - `HpBase`. Among other things, these let us prove that all accessible addresses are representable in a machine word (axiom RWF).

5.6 Semantic preservation

We say a pair of RTL and Mach behaviors are *equivalent* (written \approx) if they both Terminate with the same value, both Diverge, both get Stuck, or both Failstop (for *any* reason). Our notion of semantic preservation is two-way refinement: if an RTL program does not get Stuck, then each of its behaviors is equivalent to a behavior of the corresponding Mach program, and vice-versa. Since both Mach and RTL (with a specific memory model instantiation) are deterministic, this simplifies to the following strong result. For any RTL program P , let $\mathcal{I}(P)$ be the memory model instantiation defined as in §5.5, applied to P . Then we have:

► **Theorem 1 (Equivalent Behavior).** *Let S be an RTL source program and T be the corresponding compiled Mach target program. Let $\mathcal{B}_{RTL}[I](S)$ be the behavior of S under the RTL semantics with memory model instantiation I , and $\mathcal{B}_{Mach}(T)$ be the behavior of T under the Mach semantics. Finally, let $B_{RTL} = \mathcal{B}_{RTL}[I](S)$ and $B_{Mach} = \mathcal{B}_{Mach}(T)$. Then, if B_{RTL} is not Stuck, $B_{Mach} \approx B_{RTL}$.*

As noted in §5.1, we can easily define a static typing judgement on RTL programs that rules out Stuck behavior. Then we have a corollary stating that if S is well-typed, $B_{Mach} \approx B_{RTL}$. The proof of this theorem is based on a standard stepwise forward simulation lemma showing that each RTL step corresponds to one or more Mach steps while preserving a matching relation between states. Iterating this lemma directly gives forward refinement; coupled with determinacy of the target semantics it also implies reverse refinement [21].

As usual, the main challenge of the proof lies in defining the matching relation by cases over the possible RTL states, which all include the memory model’s internal state M as one component. To describe Mach memory, we use a version of CompCert’s separation logic library, modified to work for tagged concrete memory. A typical separation logic assertion is the following characterization of the Mach memory corresponding to a single stack frame, as the separated conjunction of three simpler assertions:

```
Definition frame_contents (f:RTL.function) (sp:Z) (retaddr:mval)
  (sB:RTL.regbank) (vm:MemInst.valmem) : massert :=
  match_env f sB sp ** hasvalue (sp + ra_ofs f) retaddr t_pro
  ** match_pub vm (sp + arr_ofs f) (sp + arr_ofs f + sz_a f).
```

Roughly, this says that a frame based at Mach memory address sp contains separate slots for the private values of parameter and local registers of the RTL function (`match_env`), the private saved return address (`hasvalue`), and the public array contents stored in M ’s *vals* component (`match_pub`). More complex assertions are used to characterize the memory of the stack as a whole. In addition to matching memory contents, we must maintain a *three-way* relation among the RTL call stack, the *st* component of M , and the Mach SP, in order to guarantee that source and target share the same notion of accessible stack memory.

Heap matching is easier, because RTL delegates all knowledge of heap structure to M . The following definition characterizes the heap segment h starting at Mach memory base address b ; note the encoding of the abstract allocation flag af into the sign of the size in the header.

```
Fixpoint heap_contents (h:MemInst.heap) (b:Z) (vm:MemInst.valmem) : massert :=
  match h with
  | [] => pure True
  | (sz,af)::rest => hasvalue b (if af then sz+1 else -(sz+1)) t_pro
    ** match_pub vm (b+1) (b+1+sz)
    ** heap_contents rest (b+1+sz) vm
  end.
```

The overall matching relation combines stack matching, heap matching, and a rather large number of purely technical invariants. The simulation proof itself is quite lengthy, but fairly straightforward. We develop a series of lemmas to show that whenever source and target are in matching states, each publicly accessible address in M points to an **unpro** location in Mach memory with the same value, and each inaccessible address points to a **pro** Mach address. These are then used to prove that private (resp. public) stores in RTL can be simulated by privileged (resp. unprivileged) stores in Mach, preserving state matching; furthermore, failing stores in RTL can be simulated by failing stores in Mach. We prove

auxiliary lemmas about fixed sequences of code generated during compilation for function call, entry and exit. Similarly, we prove that the low-level Mach implementations of `malloc` and `free` correctly simulate those in the M implementation.

The overall structure of our development is inspired by CompCert, and we make direct use of the following CompCert modules: **Separation** (modified as noted above), **Integers** (for machine integers), **Smallstep** and **Behaviors** (both modified to remove traces and add Failstop states), and various low-level libraries.

6 Related Work

Memory models. Norrish [31] gives a mechanized C semantics with a concrete byte-level memory, which was later refined by Tuch, et al. [39, 38] to incorporate type-based non-aliasing. The main focus of this line of work is to support verification of C programs, rather than of C compilers. Memarian’s Cerberus system [26] includes an (unaxiomatized) C memory model interface that abstracts over various alternatives for pointer semantics, some fairly concrete.

Several variants of the original CompCert memory model [22] treat memory and pointer representations more concretely. CompCertTSO [35], which extends CompCert v1.5 to target a low-level machine with a TSO relaxed memory model, switches from abstract to concrete pointers early in the compilation pipeline. Since CompCert v1.5 lacked alias analysis, the impact of concrete addressing on optimization does not seem to have been considered. Mullen et al. [28] append a new peephole optimization pass to the CompCert pipeline that uses a low-level version of assembly code in which pointers have been mapped to concrete integers. Our concrete semantics should validate all of their transformations. CompCertMC [41] extends the CompCert proof chain to a machine language with a flat, concrete memory model, similar to our Mach language. None of these systems expose concrete pointers at source level, or attempt to give meaning to OOB behaviors.

Oracles. Carbonneaux et al. [8] introduce the idea of a memory oracle that collects information from the compiler about the size of each target stack frame, and reflects that back to the source. CompCertMC [41] and CompCertS [6, 5] employ such an oracle to derive a bound on total stack size from inspection of the source program. Our oracle is much more detailed, as it says *where* each allocation goes, not just how big it is; also, unlike these systems, we describe heap allocation in detail. On the other hand, we deliberately avoid exposing private data size information in the source semantics itself, so we cannot support bounds calculations there. Another difference is that these systems treat source level OOM as a stuck behavior for which the compiler makes no guarantees, whereas we treat OOM as a distinct behavior that is verifiably preserved by the compiler.

Low-level pointer arithmetic. Kang et al. [15] propose a hybrid *quasi-concrete* memory model in which abstract pointers acquire concrete addresses only when they are cast to integers. The intent is to support both arithmetic operations on cast values and aggressive optimization when casting has not occurred. As in our semantics, OOM becomes an observable behavior of the source program, which is guaranteed to be preserved by the target; one unintuitive feature is that these OOMs occur at cast time rather than at allocation time. OOB behaviors remain UB, hence not preserved.

Besson et al. [6, 5] propose a memory model that keeps CompCert’s abstract representation of pointers, but also supports certain bit-level arithmetic computations. Their CompCertS semantics builds a symbolic representation of each computed pointer value in terms of

abstract block addresses; this has well-defined semantics iff its value is invariant under all possible legal block placements. For example, CompCertS gives the expected semantics to function `main` of Figure 2, since the value of `r` does not depend on where `p` lives, but not to `memmove`, since the result of comparing `dest` and `src` depends on the relative placements of the corresponding blocks. Our semantics gives defined semantics to both functions.

In a separate line of work, Besson et al. [7] show how to modify CompCert to turn the bit-level pointer operations needed for SFI memory address “warping” into well-defined behaviors that are preserved by compilation, by adding a compiler pass that “arithmetises” all pointers into integer offsets within a single sandbox block, referenced by a shadow stack pointer [17]; the behavior of globals and `malloc` is axiomatized to use the sandbox as well. They prove in Coq that transformed programs are well-defined and safe. They do not propose a concrete pointer semantics at source level; indeed, they do not attempt to prove that their new pass preserves source behavior (which would require adding a notion of OOM at source level). Their transformation inhibits later optimizations much like our semantics; their benchmarks suggest that this can have noticeable, though inconsistent, effects on performance.

7 Conclusions and Future Work

We have proposed a concrete memory semantics for C, with no memory UBs, suitable for compiling to a target having public vs. private memory enforcement, and equipped with a novel oracle that predicts the concrete placement of memory allocations based on the actual behavior of the compiler and runtime system. Our proof-of-concept verified compiler demonstrates the feasibility of this approach for a very simple, but characteristic, subset of C and a tag-based enforcement mechanism.

Our next work is to extend this subset by incorporating function pointers, which can be protected in the target environment using a range of hardware and software tagging or trampolining techniques [7] analogous to those used for data pointers. Once again, it is desirable to abstract away the details of enforcement when specifying the source semantics; we believe this can be done in a style similar to the oracular memory model we use here.

A significant question is whether our oracular approach supports vertical compositionality across a multi-phase compiler. It is particularly desirable that oracles can be constructed modularly, i.e. that an oracle instantiation for each stage can be constructed from an *arbitrary* instantiation of the successor stage’s oracle. We have some preliminary work on an inlining phase suggesting that this is indeed possible, but much more experience is needed. Ultimately, we hope to use this approach to build a complete *Concrete C* compiler with a UB-free source semantics and the full scope and depth of CompCert.

References

- 1 Arthur Azevedo de Amorim, Maxime Dénès, Nick Giannarakis, Catalin Hritcu, Benjamin C. Pierce, Antal Spector-Zabusky, and Andrew Tolmach. Micro-policies: Formally verified, tag-based security monitors. In *2015 IEEE Symposium on Security and Privacy*, pages 813–830, 2015. doi:10.1109/SP.2015.55.
- 2 Sean Anderson, Allison Naaktgeboren, and Andrew Tolmach. Flexible runtime security enforcement with tagged C. In Panagiotis Katsaros and Laura Nenzi, editors, *Runtime Verification - 23rd International Conference, RV 2023, Thessaloniki, Greece, October 3-6, 2023, Proceedings*, volume 14245 of *Lecture Notes in Computer Science*, pages 231–250. Springer, 2023. doi:10.1007/978-3-031-44267-4_12.

- 3 ARM. Armv8.5-a memory tagging extension white paper. URL: https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Arm_Memory_Tagging_Extension_Whitepaper.pdf.
- 4 Daniel J. Bernstein. boringcc. Boring crypto google group post, 2015. URL: <https://groups.google.com/g/boring-crypto/c/48qa1kWignU/m/o8GGp2K1DAAJ>.
- 5 Frédéric Besson, Sandrine Blazy, and Pierre Wilke. CompCertS: A Memory-Aware Verified C Compiler using a Pointer as Integer Semantics. *Journal of Automated Reasoning*, 63(2):369–392, August 2019. doi:10.1007/s10817-018-9496-y.
- 6 Frédéric Besson, Sandrine Blazy, and Pierre Wilke. A verified compcert front-end for a memory model supporting pointer arithmetic and uninitialised data. *J. Autom. Reasoning*, 62(4):433–480, 2019. doi:10.1007/s10817-017-9439-z.
- 7 Frédéric Besson, Sandrine Blazy, Alexandre Dang, Thomas P. Jensen, and Pierre Wilke. Compiling sandboxes: Formally verified software fault isolation. In Luís Caires, editor, *Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings*, volume 11423 of *Lecture Notes in Computer Science*, pages 499–524. Springer, 2019. doi:10.1007/978-3-030-17184-1_18.
- 8 Quentin Carbonneaux, Jan Hoffmann, Tahina Ramananandro, and Zhong Shao. End-to-end verification of stack-space bounds for C programs. In Michael F. P. O’Boyle and Keshav Pingali, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI ’14, Edinburgh, United Kingdom - June 09 - 11, 2014*, pages 270–281. ACM, 2014. doi:10.1145/2594291.2594301.
- 9 CHR Chhak, Andrew Tolmach, and Sean Anderson. Towards formally verified compilation of tag-based policy enforcement. In *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2021*, pages 137–151, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3437992.3439929.
- 10 Coq Development Team. *The Coq Reference Manual, version 8.18.0*, September 2023. URL: <https://coq.inria.fr/doc/V8.18.0/refman/>.
- 11 Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7, SSYM’98*, page 5, USA, 1998. USENIX Association. URL: https://static.usenix.org/publications/library/proceedings/sec98/full_papers/cowan/cowan.pdf.
- 12 Joe Devietti, Colin Blundell, Milo M. K. Martin, and Steve Zdancewic. HardBound: Architectural support for spatial safety of the C programming language. In *13th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 103–114, 2008. URL: http://acg.cis.upenn.edu/papers/asplos08_hardbound.pdf.
- 13 Udit Dhawan, Catalin Hritcu, Raphael Rubin, Nikos Vasilakis, Silviu Chiricescu, Jonathan M. Smith, Thomas F. Knight, Jr., Benjamin C. Pierce, and Andre DeHon. Architectural support for software-defined metadata processing. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS ’15*, pages 487–502, New York, NY, USA, 2015. ACM. doi:10.1145/2694344.2694383.
- 14 Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and J. F. 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, 2017. doi:10.1145/3062341.3062363.
- 15 Jeehoon Kang, Chung-Kil Hur, William Mansky, Dmitri Garbuzov, Steve Zdancewic, and Viktor Vafeiadis. A formal C memory model supporting integer-pointer casts. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI ’15*, pages 326–335, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2737924.2738005.

- 16 Robbert Krebbers. *The C Standard Formalized in Coq*. PhD thesis, Radboud University Nijmegen, December 2015. URL: <http://robbertkrebbers.nl/research/thesis.pdf>.
- 17 Joshua Kroll, Gordon Stewart, and Andrew Appel. Portable software fault isolation. In *27th IEEE Computer Security Foundations Symposium*. IEEE, 2014. URL: <http://www.cs.princeton.edu/~appel/papers/psfi.pdf>.
- 18 Volodymyr Kuznetsov, Laszlo Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. Code-pointer integrity. 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 147–163. USENIX Association, 2014. URL: <https://www.usenix.org/system/files/conference/osdi14/osdi14-paper-kuznetsov.pdf>.
- 19 Juneyoung Lee, Chung-Kil Hur, Ralf Jung, Zhengyang Liu, John Regehr, and Nuno P. Lopes. Reconciling high-level optimizations and low-level code in LLVM. *Proc. ACM Program. Lang.*, 2(OOPSLA), October 2018. doi:10.1145/3276495.
- 20 Xavier Leroy. A Formally Verified Compiler Back-End. *J. Autom. Reason.*, 43(4):363–446, December 2009. doi:10.1007/s10817-009-9155-4.
- 21 Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009. doi:10.1145/1538788.1538814.
- 22 Xavier Leroy and Sandrine Blazy. Formal verification of a C-like memory model and its uses for verifying program transformations. *Journal of Automated Reasoning*, 41(1):1–31, 2008. URL: <http://xavierleroy.org/publi/memory-model-journal.pdf>.
- 23 Xavier Leroy et al. CompCert 3.10. URL: <https://github.com/AbsInt/CompCert/releases/tag/v3.10>.
- 24 Hans Liljestrand, Carlos Chinae Perez, Rémi Denis-Courmont, Jan-Erik Ekberg, and N. Asokan. Color my world: Deterministic tagging for memory safety. *CoRR*, abs/2204.03781, 2022. doi:10.48550/arXiv.2204.03781.
- 25 John Lu and Keith D. Cooper. Register promotion in C programs. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation, PLDI '97*, pages 308–319, New York, NY, USA, 1997. Association for Computing Machinery. doi:10.1145/258915.258943.
- 26 Kayvan Memarian. *The Cerberus C Semantics*. PhD thesis, University of Cambridge, 2023. URL: <https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-981.pdf>.
- 27 Kayvan Memarian, Victor B. F. Gomes, Brooks Davis, Stephen Kell, Alexander Richardson, Robert N. M. Watson, and Peter Sewell. Exploring C semantics and pointer provenance. *Proceedings of the ACM on Programming Languages*, 3, 2019. URL: <https://dl.acm.org/citation.cfm?id=3290380>.
- 28 Eric Mullen, Daryl Zuniga, Zachary Tatlock, and Dan Grossman. Verified peephole optimizations for CompCert. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016*, pages 448–461, 2016. doi:10.1145/2908080.2908109.
- 29 Santosh Nagarakatte, Jianzhou Zhao, Milo M. K. Martin, and Steve Zdancewic. SoftBound: highly compatible and complete spatial memory safety for C. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 245–258. ACM, 2009. URL: http://repository.upenn.edu/cgi/viewcontent.cgi?article=1941&context=cis_reports.
- 30 Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. CETS: Compiler enforced temporal safety for C. *SIGPLAN Not.*, 45(8):31–40, June 2010. doi:10.1145/1837855.1806657.
- 31 Michael Norrish. *C formalised in HOL*. PhD thesis, University of Cambridge, 1998. URL: <https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-453.pdf>.
- 32 Marco Patrignani, Amal Ahmed, and Dave Clarke. Formal approaches to secure compilation: A survey of fully abstract compilation and related work. *ACM Comput. Surv.*, 51(6), February 2019. doi:10.1145/3280984.

- 33 Alexander L. Richardson. *Complete Spatial Safety for C and C++ using CHERI capabilities*. PhD thesis, University of Cambridge, October 2019. URL: <https://www.repository.cam.ac.uk/bitstream/handle/1810/307454/thesis-hardbound.pdf>.
- 34 Kostya Serebryany. ARM memory tagging extension and how it improves C/C++ memory safety. *login Usenix Mag.*, 44(2), 2019. URL: <https://www.usenix.org/publications/login/summer2019/serebryany>.
- 35 Jaroslav Sevcík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. CompCertTSO: A verified compiler for relaxed-memory concurrency. *Journal of the ACM*, 60(3):22, 2013. doi:10.1145/2487241.2487248.
- 36 Laurent Simon, David Chisnall, and Ross J. Anderson. What you get is what you C: Controlling side effects in mainstream C compilers. In *IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 1–15. IEEE, 2018. doi:10.1109/EuroSP.2018.00009.
- 37 Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. SoK: Eternal war in memory. In *IEEE Symposium on Security and Privacy*, pages 48–62. IEEE Computer Society, 2013. doi:10.1109/SP.2013.13.
- 38 Harvey Tuch. Formal verification of C systems code. *J. Autom. Reasoning*, 42:125–187, April 2009. doi:10.1007/s10817-009-9120-2.
- 39 Harvey Tuch, Gerwin Klein, and Michael Norrish. Types, bytes, and separation logic. *SIGPLAN Not.*, 42(1):97–108, January 2007. doi:10.1145/1190215.1190234.
- 40 Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proceedings of the Symposium on Operating Systems Principles, SOSP*, pages 203–216, 1993. URL: <http://www.eecs.harvard.edu/~greg/cs255sp2004/wahbe93efficient.pdf>.
- 41 Yuting Wang, Pierre Wilke, and Zhong Shao. An abstract stack based approach to verified compositional compilation to machine code. *Proc. ACM Program. Lang.*, 3(POPL), January 2019. doi:10.1145/3290375.
- 42 Robert N.M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, Steven J. Murdoch, Robert Norton, Michael Roe, Stacey Son, and Munraj Vadera. CHERI: A hybrid capability-system architecture for scalable software compartmentalization. In *Proceedings - IEEE Symposium on Security and Privacy*, 2015. doi:10.1109/SP.2015.9.
- 43 Jianhao Xu, Kangjie Lu, Zhengjie Du, Zhu Ding, Linke Li, Qiushi Wu, Mathias Payer, and Bing Mao. Silent bugs matter: A study of compiler-introduced security bugs. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 3655–3672, Anaheim, CA, August 2023. USENIX Association. URL: <https://www.usenix.org/conference/usenixsecurity23/presentation/xu-jianhao>.
- 44 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. *Communications of the ACM*, 53(1):91–99, 2010. doi:10.1145/1629175.1629203.

Integrals Within Integrals: A Formalization of the Gagliardo-Nirenberg-Sobolev Inequality

Floris van Doorn   

Mathematical Institute, University of Bonn, Germany

Heather Macbeth   

Department of Mathematics, Fordham University, New York, NY, USA

Abstract

We introduce an abstraction which allows arguments involving iterated integrals to be formalized conveniently in type-theory-based proof assistants. We call this abstraction the *marginal construction*, since it is connected to the marginal distribution in probability theory. The marginal construction gracefully handles permutations to the order of integration (Tonelli’s theorem in several variables), as well as arguments involving an induction over dimension.

We implement the marginal construction and several applications in the language Lean. The most difficult of these applications, the Gagliardo-Nirenberg-Sobolev inequality, is a foundational result in the theory of elliptic partial differential equations and has not previously been formalized.

2012 ACM Subject Classification Mathematics of computing → Continuous mathematics; Theory of computation → Logic and verification

Keywords and phrases Sobolev inequality, measure theory, Lean, formalized mathematics

Digital Object Identifier 10.4230/LIPIcs.ITP.2024.37

Supplementary Material

Software (Formalization code): <https://github.com/leanprover-community/mathlib4>

archived at `swh:1:dir:650d45022cc1853d3a91744387e43ff17631c638`

Acknowledgements We thank Patrick Massot for useful comments on a draft of this article.

1 Introduction

There are two major challenges which appear in the formalization of iterated integration. First, according to Tonelli’s theorem, the order of integration does not matter on well-behaved integrands. A formalism for iterated integration should make this convenient to state and apply. Secondly, iterated integration often turns up in the wild in the context of analytic arguments involving induction on dimension. Experience suggests such arguments are intrinsically hard to formalize. A good formalism for iterated integration should provide auxiliary constructions which enable users to mimic such induction arguments.

In this article we introduce a framework for iterated integration in the `Mathlib` library of the interactive proof assistant Lean. We test this framework in several applications, most notably in a proof of the Gagliardo-Nirenberg-Sobolev inequality, a foundational result from the theory of elliptic partial differential equations. The proof of the inequality is a tricky argument whose details are often elided in the literature. It involves both the reordering of iterated integrals and (something akin to) induction on dimension.

The structure of the article is as follows. As a foundation for this project we construct the finitary product measure in Lean (Section 3). This is the most general context for which iterated integration can be considered. This setting includes \mathbb{R}^n , whose standard measure is built as the product of n copies of the Lebesgue measure on \mathbb{R} . This work builds on earlier work of van Doorn [19] defining the binary product measure and a pre-existing measure theory library developed over the previous several years [3].



© Floris van Doorn and Heather Macbeth;

licensed under Creative Commons License CC-BY 4.0

15th International Conference on Interactive Theorem Proving (ITP 2024).

Editors: Yves Bertot, Temur Kutsia, and Michael Norrish; Article No. 37; pp. 37:1–37:18

Leibniz International Proceedings in Informatics



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

We then develop (Section 4) a notion of iterated integration suitable for type theory, and implement this in Lean. We refer to our notion of iterated integration as the *marginal construction* because it is inspired by the marginal distribution in probability theory. Our construction permits expression of iterated integrals in a form which is closer to the on-paper notion, bringing the formalized math into better correspondence with the written form. Our framework, as mentioned above, allows for arguments involving induction on dimension to be expressed intuitively. Furthermore, our framework handles Tonelli’s theorem silently, reducing it to a statement on equalities of sets.

We give several demonstrations (Section 5) of our iterated integration framework. First, as an example we compute the volume of a ball in \mathbb{R}^n (Subsection 5.1), as well as a matrix change of coordinates argument previously formalized in Lean by Gouézel [9] as a key step to prove the change of variables formula for integration. In these examples, we will show how our framework can handle these arguments elegantly.

Next we provide a more elaborate example (Subsection 5.3), which we name the “grid-lines lemma.” This is an argument requiring both induction over dimension and Tonelli’s theorem, and it would be extremely cumbersome to express without an explicit notion of iterated integration.

The grid-lines lemma is an abstraction of the key argument in the Gagliardo-Nirenberg-Sobolev inequality [17, 8, 7]. The final component of our project (Section 6) is the deduction of this inequality from our grid-lines lemma. The Gagliardo-Nirenberg-Sobolev inequality has not been previously formalized.

In Section 7 we give a sketch of the importance of the Gagliardo-Nirenberg-Sobolev inequality to the theory of elliptic partial differential equations and suggest future formalization work in this direction. Related work is discussed in Section 8.

2 Preliminaries

2.1 Lean and Mathlib

Lean [4] is a theorem proving language; its logical foundation is dependent type theory. **Mathlib** [3] is its standard mathematical library, currently totalling 1.4 million lines of code. The design of **Mathlib** prioritizes convenience and mathematical generality; a tradeoff is that no effort is made to work constructively. The development of **Mathlib** is a distributed project, with some 300 contributors over the seven years of its existence.

Recently a new version, Lean 4 [15], was introduced and the **Mathlib** library was ported from Lean 3 to Lean 4. We refer to the two versions of the library as **Mathlib3** and **Mathlib4** when there is a possibility of confusion.

The finitary product measure construction described in this article (Section 3) was written in 2021 in Lean 3 and contributed to the **Mathlib3**, and was ported to Lean 4 as part of the broader **Mathlib** porting effort.

The rest of the work described in the article was written in Lean 4. The marginal construction (Section 4) and its prerequisites were contributed to **Mathlib4** in 2023, and the Sobolev inequality was contributed in 2024. We will use clickable links to link our paper to specific results in the library (to the version of **Mathlib** of June 29, 2024).¹ The application described in Section 5.1 is not part of **Mathlib**, but on a branch², since Xavier Roblot had already contributed the computation of the volume of a ball to **Mathlib4**, using different techniques.

We estimate that the whole project comprises some 3,500 lines of code.

2.2 Basic Measure Theory

In this section we briefly describe the most important parts of the measure theory library in `Mathlib` prior to our work. We refer to [19] for a fuller description.

In `Mathlib` we have the notion of *measurable space*, which is just a type equipped with a chosen σ -algebra of sets which we call the measurable sets. On a measurable space we can consider a measure μ which sends any measurable set A to a number in $[0, \infty]$ which is a monotone and countably additive. Here $[0, \infty]$ is the type of nonnegative real numbers extended by a single element ∞ , and denoted $\mathbb{R}_{\geq 0\infty}$ in Lean. In Lean we allow μ to be applied to any set A (even nonmeasurable ones), in which case it is defined as the infimum of the measures of measurable sets containing A . This makes μ an outer measure on all sets (i.e. monotone and countably subadditive function that sends \emptyset to 0).

`Mathlib` contains two notions of integration. The *Lebesgue integral* is for functions $X \rightarrow [0, \infty]$ where X is a measurable space equipped with a measure μ . The *Bochner integral* is for integrable functions $X \rightarrow E$ where E is a Banach space. We will denote both of these integrals using any of the following notations:

$$\int_X f \, d\mu = \int_X f(x) \, d\mu(x) = \int_X f = \int_X f(x) \, dx.$$

In this paper we will be almost exclusively working with the Lebesgue integral.

Given two measurable spaces X and Y , and two σ -finite measures μ and ν , we can construct a measure $\mu \times \nu$ on the measurable space $X \times Y$, which satisfies

$$(\mu \times \nu)(A \times B) = \mu(A)\nu(B)$$

for all sets A and B (we do not need to require that A and B are measurable).

Tonelli's theorem is an important theorem that states how to compute Lebesgue integrals with respect to the product measure.

► **Theorem 1** (Tonelli's theorem). *Let $f : X \times Y \rightarrow [0, \infty]$ be a measurable function. Then* \square

$$\int_{X \times Y} f \, d(\mu \times \nu) = \int_X \int_Y f(x, y) \, d\nu(y) \, d\mu(x) = \int_Y \int_X f(x, y) \, d\mu(x) \, d\nu(y)$$

and all the functions in the integrals above are measurable. \square

There is also an analogous theorem for the Bochner integral in `Mathlib`, called Fubini's theorem, but we will not be using that in this paper.

2.3 Notation

In this paper, we will use set-theoretic notation for type-theoretic concepts, writing $i \in \iota$ or $A \subseteq \iota$ even when ι is a type.

Let ι be a type. If \mathbf{x} is a vector of type $\prod_{i \in \iota} X_i$, and t is of type X_i , then $X_i(\mathbf{x}, t)$ denotes the vector whose i -th coordinate is t and whose j -th coordinate is \mathbf{x}_j for $j \neq i$. In Lean this vector is denoted `Function.update x i t`. \square

Similarly, if $A \subseteq \iota$ and $y : \prod_{i \in \iota} X_i$, we use the same notation $X_A(\mathbf{x}, y)$ for the operation that updates \mathbf{x} on A . \square

$$X_A(x, y)_i := \begin{cases} y_i & \text{if } i \in A \\ x_i & \text{otherwise.} \end{cases}$$

3 Finite Product Measures

There are a few ways to define the product measure on finite product spaces. Conceptually this can be done by iterating the binary product measure construction. However, some care is required, since the spaces $X \times (Y \times Z)$ and $(X \times Y) \times Z$ are not the same space, they are merely equivalent spaces.

Given a finite family of measurable spaces $(X_i)_{i \in \iota}$ with a σ -finite measure μ_i on X_i , we want to define the product measure on $\prod_{i \in \iota} X_i$. We could define the measure by choosing an arbitrary enumeration of ι as $\iota = \{i_1, \dots, i_k\}$, and then by using the measurable equivalence

$$\left(\prod_{i \in \iota} X_i \right) \simeq X_{i_1} \times X_{i_2} \times \dots \times X_{i_k}, \tag{1}$$

to transport the iterated binary product measure from the right-hand side to the left-hand side. We decide not to do this in order to avoid arbitrary choices in the definition. Instead, we don't care too much how we define the measure, as long as it satisfies the property that if $A_i \subseteq X_i$ for all $i \in \iota$, then

$$(\prod_i \mu_i)(\prod_i A_i) = \prod_i \mu_i(A_i). \tag{2}$$

We will define the measure as the maximal measure satisfying (2). To do this, we first define the projection $\pi_i(A)$ of a subset $A \subseteq \prod_{i \in \iota} X_i$ as the image of A under the evaluation function $\pi_i : (\prod_{i \in \iota} X_i) \rightarrow X_i$. Then we define an auxiliary function n which sends a set $A \subseteq \prod_{i \in \iota} X_i$ to

$$n(A) := \prod_{i \in \iota} \mu_i(\pi_i(A)) \in [0, \infty].$$

Note that n will be equal to the measure of the smallest box containing A . Now there is a unique maximal outer measure m such that $m(A) \leq n(A)$ for all sets A .

Next, we want to turn this outer measure m into a measure on the product space. We say that a subset $A \subseteq \prod_{i \in \iota} X_i$ is *Carathéodory-measurable w.r.t. m* if for all $B \subseteq \prod_{i \in \iota} X_i$ the following equality holds:

$$m(B) = m(B \cap A) + m(B \setminus A).$$

We then show that all measurable subsets of $\prod_{i \in \iota} X_i$ are actually Carathéodory-measurable w.r.t. m , and this shows that m allows us to get a measure $\prod_i \mu_i$ on $\prod_{i \in \iota} X_i$, such that for each measurable set A we have $(\prod_i \mu_i)(A) = m(A)$.

Finally, we need to show that this measure satisfies (2). We first do this in the case that each A_i is measurable. In this case, it is easy to show that

$$(\prod_i \mu_i)(\prod_i A_i) = m(\prod_i A_i) \leq n(\prod_i A_i) = \prod_i \mu_i(A_i).$$

We can show the reverse inequality by giving a specific instance of a measure that is bounded by n and satisfies (2). To do this, we use the idea at the start of this section, by enumerating $\iota = \{i_1, \dots, i_k\}$ and using equivalence (1) to construct some specific measure ν . It is not too hard to show that $\nu \leq n$ and that $\nu(\prod_i A_i) = \prod_i \mu_i(A_i)$. Since $\prod_i \mu_i$ is the maximal measure bounded by n , we have $\nu \leq \prod_i \mu_i$, hence reverse inequality follows.

There are some interesting observations in implementing ν , since it is naively defined by recursion on the cardinality of ι . However, it is not so easy to perform recursion on finite types, especially if you define something that depends on the ordering on the type. Furthermore, it is convenient if we don't transport along too many equivalences, during our construction.

Our solution was to define a new way to define product types as an auxiliary construction. [↗](#)

```
def TProd { $\iota$  : Type*} ( $\alpha$  :  $\iota \rightarrow$  Type*) ( $l$  : List  $\iota$ ) : Type* :=
  l.foldr (fun i  $\beta \mapsto \alpha$  i  $\times \beta$ ) PUnit
```

So e.g. $\text{TProd } \alpha$ $[i, j, k]$ is by definition α $i \times \alpha$ $j \times \alpha$ $k \times \text{PUnit}$, where PUnit is the (universe polymorphic) unit type. This definition is convenient, since $\text{TProd } \alpha$ $(i::l)$ is by definition the same as α $i \times \text{TProd } \alpha$ l . This makes it very easy to define the product measure on $\text{TProd } \alpha$ l by induction on l , given measures on each α i . [↗](#) We prove that if l contains each element of ι exactly once, then $\text{TProd } \alpha$ l is equivalent to the usual product $\prod_i \alpha_i$. [↗](#) Finally, we construct ν by transporting the measure on $\text{TProd } \alpha$ l along this equivalence. [↗](#) Finally, this definition makes it very easy to show that $\nu(\prod_i A_i) = \prod_i \mu_i(A_i)$. [↗](#)

In `Mathlib` we generally try to avoid these auxiliary constructions, because it's yet another way to talk about the same mathematical object. The thing to be worried about is that we would want all the properties for product of types stated both for the usual Π -type and for TProd , leading to a large duplication of lemmas. However, in this case we explicitly mark TProd as an implementation detail, and avoid its usage unless you specifically want its precise definitional behavior. For our construction, this definitional behavior made the construction particularly easy, since we didn't have to transport anything along an equivalence in the recursion argument, only once at the end.

This definition of finitary product measures was completed in 2021 and has since been used in various analysis formalizations in Lean. It is used to define the Lebesgue measure on \mathbb{R}^n (or, more precisely, $\iota \rightarrow \mathbb{R}$). There is another definition of this measure using the Haar measure, but we show that these give rise to the same measure. [↗](#) And this definition makes it a lot easier to show some simple properties about this measure, such as (2).

It is also used by Kudryashov [13] as the setting for the Bochner-integrability version of his divergence theorem, and by Gouézel [9] in order to use \mathbb{R}^n as a setting for certain measure-theoretic results which are subsequently transferred to a general finite-dimensional normed space by a choice of basis (see Subsection 5.2).

4 The Marginal Construction

4.1 Approaches to Formalization

We note that the fundamental issues in formalizing iterated integration are the same for systems based on simple type theory and dependent type theory. In dependent type theory it is possible directly to express a dependent finitary product $\prod_{j:\iota} A_j$ of measure spaces, but the issues we address appear already in the setting of a function type $\iota \rightarrow A$, which can be expressed in simple type theory.

When working with products of finitely many spaces, one also wants to use the Tonelli and Fubini theorems. For example, if $f : \mathbb{R}^{n+m} \rightarrow \mathbb{R}$, then one might want to write

$$\int_{\mathbb{R}^{n+m}} f(z) dz = \int_{\mathbb{R}^n} \int_{\mathbb{R}^m} f(x, y) dy dx.$$

People familiar with formalization will note that this is not simply an application of Fubini's theorem, since \mathbb{R}^{n+m} is not the same entity as $\mathbb{R}^n \times \mathbb{R}^m$. One option is to show that there

37:6 The Gagliardo-Nirenberg-Sobolev Inequality

is a “canonical” measure-preserving equivalence $\mathbb{R}^{n+m} \simeq \mathbb{R}^n \times \mathbb{R}^m$. However, this is still a bit inconvenient to work with, since one has to work explicitly with this equivalence. Below we will develop a framework that does not require working with any of these measure-preserving equivalences (of course, to prove that the framework is correct, we will use such measure-preserving equivalences).

Another expression that one might want to deal with is for $f : \mathbb{R}^n \rightarrow \mathbb{R}$ an iterated integral of the form

$$\int \cdots \int f(x_1, \dots, x_n) dx_1 \cdots dx_k$$

where $k \leq n$. Here only some of the arguments of f are integration variables, and the remaining expression is still a function of the remaining variables. Manipulating iterated integrals like this is a key part of the proof of the grid-lines lemma discussed in Section 5.3.

The solution we implement was suggested in the concluding section of [19], which we will do in the next section.

4.2 Definition and Properties

We encapsulate this notion in the following definition. In this definition we will generalize \mathbb{R}^n to an arbitrary product space $\prod_{i \in \iota} X_i$.

► **Definition 2.** *Let ι be a indexing set (not necessarily finite), $A \subseteq \iota$ a finite subset and E be a Banach space. For $i \in \iota$ suppose we are given a measure space (X_i, μ_i) and let $f : (\prod_{i \in \iota} X_i) \rightarrow [0, \infty]$ be a function. Then the marginal of f w.r.t. A*

$$\int \cdots \int_{i \in A} f d\mu_i$$

is by definition another function $(\prod_{i \in \iota} X_i) \rightarrow [0, \infty]$ that is defined as (the notation is explained in Subsection 2.3). ◻

$$x \mapsto \int_{\prod_{i \in A} X_i} f(X_A(x, y)) d\prod_{i \in A} \mu_i(y).$$

Note that $\int \cdots \int_{i \in A} f d\mu_i$ is a function that does not depend on the arguments in A . We could also view this as a function on $\prod_{i \in \iota \setminus A} X_i$ instead of $\prod_{i \in \iota} X_i$. However, it is much more convenient to view it as a function on the whole product space $\prod_{i \in \iota} X_i$, since the alternative makes the statements of the lemmas below much more complicated.

We call this operation the marginal of f because of our intuition from probability theory. If all the μ_i are probability measures and f is a random variable, then $\int \cdots \int_{i \in A} f d\mu_i$ is the marginal variable on $\prod_{i \in \iota \setminus A} X_i$.

Note that we do not assume that ι is finite: this construction works in an infinite product, as long as we only have finitely many integration variables.

► **Lemma 3.** *The following basic properties hold for any function $f : (\prod_{i \in \iota} X_i) \rightarrow [0, \infty]$.*

1. $\int \cdots \int_{i \in \emptyset} f d\mu_i = f$. ◻
2. If $x, x' \in \prod_{i \in \iota} X_i$ and $x_i = x'_i$ for all $i \in \iota \setminus A$ then $\int \cdots \int_{i \in A} f d\mu_i$ will have the same value on x and x' . ◻
3. $\int \cdots \int_{i \in A} f d\mu_i$ is monotone in f . ◻
4. If ι is finite then $\int \cdots \int_{i \in \iota} f d\mu_i$ is the constant function with value $\int f d\prod_i \mu_i$. ◻
5. If f is measurable, then so is $\int \cdots \int_{i \in A} f d\mu_i$. ◻

Proof. Parts 2, 3 and 4 follow immediately from the definition.¹

For Part 1 note that the marginal of f w.r.t. \emptyset is an integral over an empty product space. Since the empty product of measures is the Dirac measure on the unique point in the space, this equality follows easily.

For Part 5, to show that $\int \cdots \int_{i \in A} f \, d\mu_i$ is measurable, by Tonelli's theorem it suffices to show measurability for $(x, y) \mapsto f(X_A(x, y))$, which is an easy exercise. ◀

Using the definition of marginal, we get a very nice formulation of Tonelli's theorem for finitary products.

► **Lemma 4.** *If f is measurable and A and B are disjoint finite subsets of ι , then* ◻

$$\int \cdots \int_{i \in A \cup B} f \, d\mu_i = \int \cdots \int_{i \in A} \int \cdots \int_{j \in B} f \, d\mu_j \, d\mu_i$$

Proof. Since A and B are disjoint, we have a measurable equivalence

$$e : \left(\prod_{i \in A} X_i \right) \times \left(\prod_{i \in B} X_i \right) \simeq \left(\prod_{i \in A \cup B} X_i \right).$$

Note that e maps the measure $(\prod_{i \in A} \mu_i) \times (\prod_{i \in B} \mu_i)$ to the measure $\prod_{i \in A \cup B} \mu_i$. Therefore we compute

$$\begin{aligned} \int \cdots \int_{i \in A \cup B} f \, d\mu_i &= \int_{\prod_{i \in A \cup B} X_i} f(X_{A \cup B}(x, y)) \, d\prod_{i \in A \cup B} \mu_i(y) \\ &= \int_{(\prod_{i \in A} X_i) \times (\prod_{i \in B} X_i)} f(X_{A \cup B}(x, e(y))) \, d(\prod_{i \in A} \mu_i) \times (\prod_{i \in B} \mu_i)(y) \\ &= \int_{\prod_{i \in A} X_i} \int_{\prod_{i \in B} X_i} f(X_{A \cup B}(x, e(y, z))) \, d\prod_{i \in B} \mu_i(z) \, d\prod_{i \in A} \mu_i(y) \\ &= \int_{\prod_{i \in A} X_i} \int_{\prod_{i \in B} X_i} f(X_B(X_A(x, y), z)) \, d\prod_{i \in B} \mu_i(z) \, d\prod_{i \in A} \mu_i(y) \\ &= \int \cdots \int_{i \in A} \int \cdots \int_{j \in B} f \, d\mu_j \, d\mu_i. \end{aligned}$$

where the second step uses the properties of e and the third step uses Tonelli's theorem. ◀

► **Lemma 5.** *For $i_0 \in \iota$,* ◻

$$\int \cdots \int_{i \in \{i_0\}} f \, d\mu_i = \int_{X_{i_0}} f(X_{i_0}(x, y)) \, d\mu_{i_0}(y)$$

Proof. We have a measurable equivalence $(\prod_{i \in \{i_0\}} X_i) \simeq X_{i_0}$ that maps the measure $\prod_{i \in \{i_0\}} \mu_i$ to μ_{i_0} . Therefore,

$$\begin{aligned} \int \cdots \int_{i \in \{i_0\}} f \, d\mu_i(x) &= \int_{\prod_{i \in \{i_0\}} X_i} f(X_{\{i_0\}}(x, y)) \, d\prod_{i \in \{i_0\}} \mu_i(y) \\ &= \int_{X_{i_0}} f(X_{i_0}(x, y)) \, d\mu_{i_0}(y). \end{aligned}$$
 ◀

¹ In Part 4 there is a slight complication in the formalization, because the type ι is not the same type as the universal subtype of ι (in the definition of marginal, A is used as a subtype of ι). This is not a problem: the proof is still only a few lines long in the formalization.

5 Applications of the Marginal Construction

5.1 Volume of an n -ball

Our first application is a loose port of Manuel Eberl's Isabelle formalization² (2017) of the formula for the volume of a ball in Euclidean n -space.

Let ι be a type of finite cardinality n . In this section \mathbf{x} will denote a point in \mathbb{R}^ι and (x_j) the individual co-ordinates of such a point. Fix a real number $R \geq 0$. We will study the Euclidean ball in \mathbb{R}^ι ,

$$\{\mathbf{x} : \|\mathbf{x}\| \leq R\} = \left\{ \mathbf{x} : \sum_{j:\iota} x_j^2 \leq R^2 \right\}.$$

Define a constant $B_n := \frac{\pi^{n/2}}{\Gamma(\frac{n}{2}+1)}$, where Γ denotes the gamma function (available in `Mathlib` due to work of David Loeffler³). In this section we prove:

► **Proposition 6.** volume $\{\mathbf{x} : \|\mathbf{x}\| \leq R\} = B_n R^n$.[☞]

We introduce the notation [☞]

$$I_k(t) := \begin{cases} 0, & t < 0 \\ t^{k/2}, & 0 \leq t, \end{cases}$$

for $k : \mathbb{N}$ and $t : \mathbb{R}$, and [☞]

$$A_s(\mathbf{x}) := B_{|s|} I_{|s|} \left(R^2 - \sum_{j \in s^c} x_j^2 \right),$$

for a set s in ι and a vector $\mathbf{x} : \mathbb{R}^\iota$.

Observe that (denoting by χ_U the characteristic function of a set U)

$$\begin{aligned} \int \cdots \int_{\emptyset^c} A_\emptyset &= \int_{\mathbf{x}:\mathbb{R}^\iota} B_{|\emptyset|} \chi_{\{\mathbf{x}:0 \leq R^2 - \|\mathbf{x}\|^2\}} = \text{volume } \{\mathbf{x} : \|\mathbf{x}\| \leq R\}, \\ \int \cdots \int_{\text{univ}^c} A_{\text{univ}} &= B_n R^n. \end{aligned}$$

(A priori the left-hand sides are functions on \mathbb{R}^ι . The statements are to be understood as saying that these functions are constant and equal to the expressions on the right-hand sides.)

So Proposition 6 follows by induction from the following fact:

► **Proposition 7.** For all sets s in ι and all $i \notin s$,[☞]

$$\int \cdots \int_{s^c} A_s = \int \cdots \int_{(\{i\} \cup s)^c} A_{\{i\} \cup s}.$$

Proof. Given $i : \iota$, $\mathbf{x} : \mathbb{R}^\iota$ and $t : \mathbb{R}$,

A computation in single-variable calculus establishes that for all natural numbers k and all real numbers c ,

$$\int B_k I_k(c - t^2) dt = B_{k+1} I_{k+1}(c).$$

² <https://isabelle-dev.sketis.net/rISABELLEc60e3d615b8>

³ <https://github.com/leanprover-community/mathlib/pull/12917>

Therefore for any $\mathbf{x} : \mathbb{R}^l$,

$$\begin{aligned} \int A_s(X_i(\mathbf{x}, t)) dt &= \int_{B_{|s|} I_{|s|}} \left(\left[R^2 - \sum_{j \in (\{i\} \cup s)^c} x_j^2 \right] - t^2 \right) dt \\ &= B_{\{\{i\} \cup s\} I_{\{\{i\} \cup s\}}} \left(R^2 - \sum_{j \in (\{i\} \cup s)^c} x_j^2 \right) = A_{\{i\} \cup s}(\mathbf{x}). \end{aligned}$$

Integrating this fact over the variables in $(\{i\} \cup s)^c$,

$$\int \cdots \int_{s^c} A_s = \int \cdots \int_{(\{i\} \cup s)^c} \left(\mathbf{x} \mapsto \int A_s(X_i(\mathbf{x}, t)) dt \right) = \int \cdots \int_{(\{i\} \cup s)^c} A_{\{i\} \cup s}. \quad \blacktriangleleft$$

5.2 Transvections Preserve the Lebesgue Measure

A *transvection* is a matrix of the form $1 + A$, where 1 is the identity matrix and A is a matrix that has one, off-diagonal, non-zero entry. `Mathlib` contains the result that the linear transformation of \mathbb{R}^n induced by a transvection preserves the Lebesgue measure.

This is one step in the proof of the corresponding result for a general matrix M (where a factor $|\det(M)|$ occurs). This result, the infinitesimal version of the change of variables formula, was contributed to `Mathlib` by Gouëzel [9, Section 5], and had previously been formalized in other systems. For example, Harrison [10, Section 7], working in `HOL Light`, calls it out as “quite hard work to formalize.” In both cases this result is proved along the way to a (non-infinitesimal) version of the change of variables formula.

The existing `Mathlib` proof was pretty long (34 lines) and required reasoning about explicit equivalences on the indexing set. Using the marginal construction, we gave a proof in 15 lines with the same mathematical argument. The main mathematical argument lies in proving equation (3) below. This remains roughly the same in both versions of the formalization, but the marginal construction allowed us to remove a lot of work for the remaining part of the argument.

► **Proposition 8.** *If ι is a finite indexing set, and T is a transvection on \mathbb{R}^ι , then T preserves the Lebesgue measure.* ◻

Proof. We have to show that $T_*\lambda = \lambda$ where λ is the Lebesgue measure. Since boxes form a basis of the σ -algebra on \mathbb{R}^n , it is sufficient to show that the measures agree on a box $A = \Pi_i A_i$, so we have to show that $\lambda(T^{-1}(A)) = \lambda(A)$. Suppose that $T = 1 + M$ where M has entry $c \neq 0$ in position (i, j) for $i \neq j$. For a given $\mathbf{x} \in \mathbb{R}^n$ we will first show that the following equality holds:

$$\lambda(\{y \mid X_i(\mathbf{x}, y) \in T^{-1}(A)\}) = \lambda(\{y \mid X_i(\mathbf{x}, y) \in A\}). \quad (3)$$

The intuition of this equality is that we’re fixing all but one of the coordinates, and looking at the length A when varying only coordinate i . We calculate:

$$\begin{aligned} \lambda(\{y \mid X_i(\mathbf{x}, y) \in T^{-1}(A)\}) &= \lambda(\{y \mid T(X_i(\mathbf{x}, y)) \in A\}) \\ &= \lambda(\{y \mid X_i(\mathbf{x}, y) + c\mathbf{x}_j \mathbf{e}_i \in A\}) = \lambda(\{y \mid X_i(\mathbf{x}, y + c\mathbf{x}_j) \in A\}) = \lambda(\{y \mid X_i(\mathbf{x}, y) \in A\}), \end{aligned}$$

where \mathbf{e}_i is the i -th standard vector and where in the last inequality we use the translation-invariance of λ , showing (3).

37:10 The Gagliardo-Nirenberg-Sobolev Inequality

To finish the proof, notice that we want to prove that

$$\int \cdots \int_{\{1, \dots, n\}} \chi_A = \int \cdots \int_{\{1, \dots, n\}} \chi_{T^{-1}(A)},$$

where χ_X is the characteristic function of X . Equation (3) can be rewritten as

$$\int \cdots \int_{\{i\}} \chi_A = \int \cdots \int_{\{i\}} \chi_{T^{-1}(A)},$$

and the claim follows from Proposition 4, in the following form:

$$\int \cdots \int_{\{1, \dots, n\}} f = \int \cdots \int_{\{1, \dots, n\} \setminus \{i\}} \int \cdots \int_{\{i\}} f. \quad \blacktriangleleft$$

5.3 Grid-lines Lemma

In this section we present our most intricate application of the marginal construction: the key argument in the Gagliardo-Nirenberg-Sobolev inequality (see Section 6), which for clarity we have abstracted as a separate proposition and baptized the *grid-lines lemma*.

This key argument is quite an illuminating test case for the difference between informal and formal mathematical practice. So before discussing our approach, we describe the presentations available in the mathematical literature. The argument involves a succession of uses of Hölder's inequality with respect to different variables of integration. In the literature, the argument is either presented in a particular low dimension and left for the reader to extrapolate, or described as an implicit induction with the actual structure of the induction being left unstated.

- **Nirenberg, 1959 [17]**: “We shall prove (2.4)’ here for $n = 3 \dots$. For general n the inequality is proved in the same way.”
- **Gilbarg-Trudinger, 1977 [8]**: “The inequality (7.27) is now integrated successively over each variable x_i , $i = 1, \dots, n$, the generalized Hölder inequality (7.11) for $m = p_1 = \dots = p_m = n - 1$ then being applied after each integration. Accordingly we obtain ...”
- **Evans, 1998 [7]**: “We continue by integrating with respect to x_3, \dots, x_n , eventually to find ...”
- **Tsui, 2008 [18]**: “To illustrate the main ideas, we discuss the case when $n = 3 \dots$. For the general case, we start with ... Repeating this process, we get ...”
- **Liu, 2023 [14]**: “[T]he inequality (1) for $p = 1$ is proved by integrating ... with respect to x_1 and applying the extended Hölder inequality, then repeating this procedure with respect to x_2, x_3, \dots, x_n successively ... This tedious procedure is not very transparent, and is not easy to follow.”

To formalize this argument, we need an explicit statement in general dimension. Given the appeals to the extrapolation in the presentations quoted above, it is perhaps not surprising that we did not find this explicit statement in the literature!

We need an ι -indexed family of sigma-finite measure spaces $(A_i)_{i:\iota}$, where ι is a type of finite cardinality n . The reader may wish to imagine for concreteness that each factor A_i is \mathbb{R} , so that the product type $\prod_{i:\iota} A_i$ is the function type $\iota \rightarrow \mathbb{R}$ (or \mathbb{R}^ι for short). The essential points of the argument remain unchanged in this special case, which is in fact the case needed for the Gagliardo-Nirenberg-Sobolev inequality.

We furthermore need a nonnegative real parameter p , which at different times will have different upper bounds (specified explicitly).

The statement of the lemma in general dimension is as follows.

► **Proposition 9** (Grid-lines lemma). *Suppose that $(n - 1)p \leq 1$. If $f : \prod_{j:\iota} A_j \rightarrow [0, \infty]$ is a measurable function, then (the notation is explained in Subsection 2.3) \square*

$$\int_{\mathbf{x}:\prod_{j:\iota} A_j} f(\mathbf{x})^{1-(n-1)p} \prod_i \left(\int_{t:A_i} f(X_i(\mathbf{x}, t)) \right)^p \leq \left(\int_{\mathbf{x}:\prod_{j:\iota} A_j} f(\mathbf{x}) \right)^{1+p}.$$

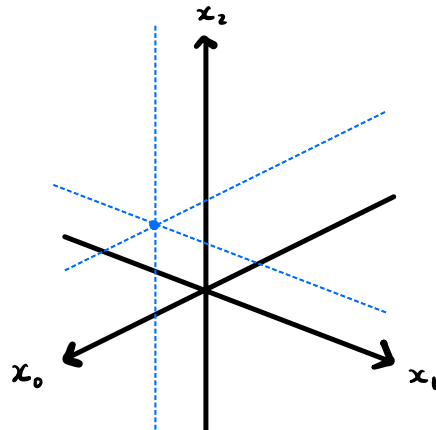
```

theorem lintegral_mul_prod_lintegral_pow_le (hp : (#ℓ - 1 : ℝ) * p ≤ 1)
  {f : (∀ i : ℓ, A i) → ℝ≥0∞} (hf : Measurable f) :
  ∫- x, f x ^ (1 - (#ℓ - 1 : ℝ) * p)
  * ∏ i, (∫- x_i, f (update x i x_i) ∂μ i) ^ p ∂.pi μ
  ≤ (∫- x, f x ∂.pi μ) ^ (1 + p)
    
```

Our name for this lemma comes from the integrand on the left-hand side. Note that at $\mathbf{x} : \prod_{j:\iota} A_j$ the integrand is a weighted product of $f(\mathbf{x})$ and expressions of the form

$$\int_{t:A_i} f(X_i(\mathbf{x}, t));$$

this expression is the integral of f in the single co-ordinate i , and in \mathbb{R}^ι such an expression represents the integral of f over the “grid line” through \mathbf{x} obtained by varying the i -th co-ordinate while fixing the others. See Figure 1.



■ **Figure 1** The left-hand integrand of the grid-lines lemma at a fixed point (blue point) is a weighted product of the value there with integrals over the lines through it parallel to the axes (blue dotted lines).

We introduce the notation

$$I_j f := \int \cdots \int_{\{j\}} f$$

for the marginal integral over the singleton set $\{j\}$ in ι , and \square

$$T_{p,s}(f) := \int \cdots \int_s f^{1-(|s|-1)p} \prod_{j \in s} (I_j f)^p.$$

37:12 The Gagliardo-Nirenberg-Sobolev Inequality

Observe that

$$T_{p,\emptyset} \left(\int_{\prod_{j:\iota} A_j} f \right) = \left(\int_{\prod_{j:\iota} A_j} f \right)^{1+p}$$

$$T_{p,\text{univ}}(f) = \int_{\mathbf{x}:\prod_{j:\iota} A_j} f(\mathbf{x})^{1-(n-1)p} \prod_{j:\iota} \left(\int_{t:A_i} f(X_i(\mathbf{x}, t)) \right)^p.$$

(A priori the left-hand sides are functions on $\prod_{j:\iota} A_j$. The statements are to be understood as saying that these functions are constant and equal to the expressions on the right-hand sides.) So Proposition 9 follows by induction from the following fact:

► **Proposition 10.** *For all sets s in ι and all $i \notin s$, and for all p such that $|s|p \leq 1$,* \square

$$T_{p,\{i\} \cup s}(f) \leq T_{p,s}(I_i f).$$

The proof is a tricky computation that relies on Hölder's inequality at its heart. Note that on the left-hand-side we have an $|s| + 1$ -times iterated integral with $|s| + 2$ factors inside the integral. If x_i denotes the i -th variable, we want to move the integral over x_i inside, and apply Hölder's inequality to the $|s| + 1$ factors that depend on x_i (whose powers sum exactly to 1).

Proof. We have that

$$[1 - |s|p] + |s|p = 1,$$

so for any $\mathbf{x} : \prod_{j:\iota} A_j$, by Hölder's inequality,

$$\int_{t:A_i} f(X_i(\mathbf{x}, t))^{1-|s|p} \prod_{j \in s} I_j f(X_i(\mathbf{x}, t))^p$$

$$\leq \left(\int_{t:A_i} f(X_i(\mathbf{x}, t)) \right)^{1-|s|p} \prod_{j \in s} \left(\int_{t:A_i} I_j f(X_i(\mathbf{x}, t)) \right)^p.$$

Therefore for any $\mathbf{x} : \prod_{j:\iota} A_j$,

$$\int_{t:A_i} f(X_i(\mathbf{x}, t))^{1-|s|p} \prod_{j \in \{i\} \cup s} I_j f(X_i(\mathbf{x}, t))^p$$

$$= \int_{t:A_i} I_i f(\mathbf{x})^p \left(f(X_i(\mathbf{x}, t))^{1-|s|p} \prod_{j \in s} I_j f(X_i(\mathbf{x}, t))^p \right)$$

$$= I_i f(\mathbf{x})^p \int_{t:A_i} f(X_i(\mathbf{x}, t))^{1-|s|p} \prod_{j \in s} I_j f(X_i(\mathbf{x}, t))^p$$

$$\leq I_i f(\mathbf{x})^p \left(\int_{t:A_i} f(X_i(\mathbf{x}, t)) \right)^{1-|s|p} \prod_{j \in s} \left(\int_{t:A_i} I_j f(X_i(\mathbf{x}, t)) \right)^p$$

$$= I_i f(\mathbf{x})^p I_i f(\mathbf{x})^{1-|s|p} \prod_{j \in s} I_j f(\mathbf{x})^p$$

$$= I_i f(\mathbf{x})^{1-(|s|-1)p} \prod_{j \in s} I_j I_i f(\mathbf{x})^p.$$

Integrating this over the variables in s ,

$$\begin{aligned}
T_{p,\{i\}\cup s}(f) &= \int \cdots \int_{\{i\}\cup s} f^{1-|s|p} \prod_{j \in \{i\}\cup s} (I_j f)^p \\
&= \int \cdots \int_s \left(\mathbf{x} \mapsto \int_{t:A_i} f(X_i(\mathbf{x}, t))^{1-|s|p} \prod_{j \in \{i\}\cup s} I_j f(X_i(\mathbf{x}, t))^p \right) \\
&\leq \int \cdots \int_s (I_i f)^{1-(|s|-1)p} \prod_{j \in s} (I_j I_i f)^p \\
&= T_{p,s}(I_i f). \quad \blacktriangleleft
\end{aligned}$$

6 Gagliardo-Nirenberg-Sobolev Inequality

The version of the inequality we prove is due independently to Nirenberg [17, Lecture II] and Gagliardo; a variant result with different exponents was proved earlier by Sobolev, and can be deduced from the Gagliardo-Nirenberg version (although we do not formalize this deduction).

The L^p norm of a function f w.r.t. a measure μ is defined to be

$$\|f\|_{L^p} := \left(\int |f|^p d\mu \right)^{\frac{1}{p}} \in [0, \infty].$$

► **Theorem 11** (Gagliardo-Nirenberg-Sobolev inequality). *Let E be a real normed space of finite dimension $n \geq 2$ with Haar measure μ . Let $1 \leq p < n$ be a real number with Sobolev conjugate $p^* = \frac{np}{n-p}$. Then there exists a nonnegative real number C such that for all compactly supported C^1 functions $u : E \rightarrow \mathbb{R}$,* ◻

$$\|u\|_{L^{p^*}} \leq C \|Du\|_{L^p}. \quad (4)$$

The Lean version, which is displayed below, features a zoo of type classes. The predicate `ContDiff ℝ 1 u` states that u is C^1 , `snorm u p' μ` is the $L^{p'}$ norm of u (w.r.t. μ) and `fderiv ℝ u` is the total derivative of u . The conclusion features a constant `SNormLESNormFderivOfEqConst F μ p : ℝ ≥ 0`. We don't care about the precise value of this constant, but it is important that it only depends on F , μ and p (and E , the space on which μ is a measure).

Note also that in the formalization we generalized the codomain of u to be any finite-dimensional normed vector space.

```

theorem lintegral_pow_le_pow_lintegral_fderiv [NormedAddCommGroup E]
  [NormedSpace ℝ E] [MeasurableSpace E] [BorelSpace E]
  [FiniteDimensional ℝ E] (μ : Measure E) [IsAddHaarMeasure μ]
  [NormedAddCommGroup F] [NormedSpace ℝ F] [FiniteDimensional ℝ F]
  {u : E → F} (hu : ContDiff ℝ 1 u) (h2u : HasCompactSupport u)
  {p p' : ℝ ≥ 0} (hp : 1 ≤ p) (h2p : 0 < finrank ℝ E)
  (hp' : (p' : ℝ)-1 = p-1 - (finrank ℝ E : ℝ)-1) :
  snorm u p' μ ≤
  SNormLESNormFderivOfEqConst F μ p * snorm (fderiv ℝ u) p μ

```

The main difficulty of the proof is for $p = 1$, and we will first prove that in the case that $E = \mathbb{R}^\iota$, where ι is a type of finite cardinality n . In that case, we can prove the following result.

37:14 The Gagliardo-Nirenberg-Sobolev Inequality

► **Proposition 12.** Let ι be a finite type of cardinality $n \geq 2$. For all compactly supported C^1 functions $u : \mathbb{R}^\iota \rightarrow \mathbb{R}$,[□]

$$\int |u|^{n/(n-1)} \leq \left(\int \|Du\| \right)^{n/(n-1)}.$$

Proof. The key observation here is that, by a half-infinite version of the Fundamental Theorem of Calculus, a compactly supported function is bounded pointwise by the integral of the norm of its gradient along any co-ordinate line. To be precise, for a given $\mathbf{x} : \mathbb{R}^\iota$ and $i : \iota$,

$$\begin{aligned} |u(\mathbf{x})| &= \left| \int_{-\infty}^{\mathbf{x}^{(i)}} (u \circ X_i(\mathbf{x}, \cdot))' (t) dt \right| \\ &\leq \int_{-\infty}^{\mathbf{x}^{(i)}} |(u \circ X_i(\mathbf{x}, \cdot))' (t)| dt \\ &\leq \int_{-\infty}^{\infty} \|Du|_{X_i(\mathbf{x}, t)}\| dt. \end{aligned}$$

Here we use $Du|_F$ to denote evaluation of Du at point F .

We obtain the desired bound by taking the product over all $i : \iota$ of these inequalities, for each $\mathbf{x} : \mathbb{R}^\iota$:

$$\begin{aligned} \int |u(\mathbf{x})|^{n/(n-1)} d\mathbf{x} &= \int \prod_i |u(\mathbf{x})|^{1/(n-1)} d\mathbf{x} \\ &\leq \int \prod_i \left(\int \|Du|_{X_i(\mathbf{x}, t)}\| dt \right)^{1/(n-1)} d\mathbf{x}; \end{aligned}$$

the last line has exactly the form of the left-hand side of the grid-lines lemma (Proposition 9), with $f(\mathbf{x}) = \|Du|_{\mathbf{x}}\|$, and so, by that Proposition, is bounded above by

$$\left(\int \|Du|_{\mathbf{x}}\| d\mathbf{x} \right)^{n/(n-1)}. \quad \blacktriangleleft$$

Proof of Theorem 11. For $p = 1$, we can raise Proposition 12 to the power $\frac{n-1}{n}$ to obtain (4) for $E = \mathbb{R}^\iota$. Then we want to transfer this statement to functions u with as domain an arbitrary finite-dimensional vector space.[□] This argument is not hard: we choose a basis on E and then use a continuous linear equivalence $e : \mathbb{R}^n \simeq E$, where n is the dimension of E . Then the measures μ and the pushforward of the Lebesgue measure $e_*(\lambda)$ are both Haar measures, so they must be the same up to some constant factor.[□] So let us assume that $\mu = c \cdot e_*(\lambda)$. Then let

$$C = \frac{\|e\|^p}{c^{p-1}},$$

where $\|e\|$ is the operator norm of e . We then apply 12 to the function $u \circ e : \mathbb{R}^n \rightarrow \mathbb{R}$. A straightforward calculation involving the chain rule then shows (4) for $p = 1$ with the aforementioned value of C .

For $p > 1$, Define $\gamma := \frac{p(n-1)}{n-p}$. A simple calculation shows that $\frac{\gamma n}{n-1} = p^* = \frac{(\gamma-1)p}{p-1}$. We now apply the version for $p = 1$ to the function $v := |u|^\gamma$.

$$\|v\|_{L^{\frac{n}{n-1}}} \leq C \|Dv\|_{L^1} \leq C \gamma \int |u|^{\gamma-1} \|Du\| \leq C \gamma \left(\int |u|^{p^*} \right)^{\frac{p-1}{p}} \left(\int \|Du\|^p \right)^{\frac{1}{p}}$$

where in the second inequality we use the chain rule and in the third inequality we use Hölder's inequality. Hence by using that $\frac{n-1}{n} - \frac{p-1}{p} = \frac{1}{p^*}$ we compute

$$\|u\|_{L^{p^*}} = \|v\|_{L^{\frac{n}{n-1}}}^{\frac{1}{p^*}} \leq C\gamma \left(\int \|Du\|^p \right)^{\frac{1}{p}} = C\gamma \|Du\|_{L^p}.$$

This finishes the proof. \blacktriangleleft

In the formalization, there is one additional step in the proof, since we generalize the codomain of u . In the proof we use the fact that $x \mapsto |x|^\gamma$ is differentiable with derivative bounded by $\gamma|x|^{\gamma-1}$. This is still true in Hilbert spaces, but not generally in normed spaces, since the norm there need not be differentiable at all. To solve this, we first prove it for arbitrary Hilbert spaces, \blacksquare and then use the fact that for every finite-dimensional normed vector space there is a continuous linear equivalence to a Hilbert space (namely \mathbb{R}^n). We can then transfer the result along this equivalence.

Note that in the formalization we transferred the inequality twice along continuous linear equivalences. However, because of the nature of the statement, this transfer is not at all easy: it involves steps like the chain rule, the uniqueness of Haar measures and estimates using the operator norm of a linear map. It would be interesting to see to what extent automated transfer tactics (such as [20]) would be able to transfer a result like this.

The Gagliardo-Nirenberg-Sobolev inequality (Theorem 11) holds uniformly for all functions on a normed space: the supports of the functions considered must be compact, but they can be arbitrarily large. For fixed p and n , the Sobolev conjugate $p^* = \frac{np}{n-p}$ is the unique exponent for which such an inequality could be true; this is easily seen by a scaling argument.

On the other hand, if one restricts consideration to functions supported within a fixed bounded region s , there is more flexibility in the choice of exponent. A variant of Theorem 11 then holds for any q such that $1 \leq q \leq p^*$. \blacksquare This follows immediately from the monotonicity of L^p -membership, which is a consequence of Hölder's inequality:

$$\left(\int_s |f|^q \right)^{1/q} \leq \text{Vol}(s)^{1/q-1/p^*} \left(\int_s |f|^{p^*} \right)^{1/p^*}.$$

The most important special case is the case when q is p itself, which is valid since $p \leq p \frac{n}{n-p} = p^*$.

► Theorem 13. *Let s be a bounded measurable set in \mathbb{R}^d . Let $1 \leq p < |d|$. There exists a constant C , such that for all C^1 functions $u : \mathbb{R}^d \rightarrow \mathbb{R}$ with support in s , \blacksquare*

$$\|u\|_{L^p} \leq C \|Du\|_{L^p}.$$

7 Future Prospects: Sketch of some PDE Theory

Sobolev spaces are a longstanding goal for formalization [1, 2]. They are a standard setting for the solution of elliptic partial differential equations.

We outline a little of this theory to motivate our interest in the Gagliardo-Nirenberg-Sobolev inequality. Sobolev spaces are certain Banach spaces of functions, let us say for “nice” domain $\Omega \subseteq \mathbb{R}^n$ and codomain \mathbb{R} . The simplest example, the Sobolev space $H_0^1(\Omega, \mathbb{R})$, is (to give a nonstandard description) the subspace of the Hilbert space $L^2(\Omega, \mathbb{R}^n)$ consisting of functions U which are equal to Du , in the sense of weak (distributional) derivatives, for some element u in $L^2(\Omega, \mathbb{R})$, and which are L^2 -approximated by C^1 compactly-supported functions

37:16 The Gagliardo-Nirenberg-Sobolev Inequality

in Ω . The Gagliardo-Nirenberg-Sobolev inequality (in the variant Theorem 13) implies that this subspace is a *closed* subspace of $L^2(\Omega, \mathbb{R}^n)$, thus Banach, and the (necessarily linear) operation sending U to a suitable u is a *bounded* linear map, which we notate P_0 .

It follows by the Fréchet-Riesz representation theorem (see formalizations [16, 1, 6]) that for any function f in $L^2(\Omega, \mathbb{R})$, there exists a unique element U of the Sobolev space H_0^1 such that for all V in H_0^1 ,

$$\int_{\Omega} \langle U, V \rangle = \int_{\Omega} f P_0(V). \quad (5)$$

If $U = Du$ for a smooth (not just L^2) function u , this condition implies that for all smooth compactly-supported v ,

$$\int \langle Du, Dv \rangle = \int f v.$$

By integrating by parts, this implies that u solves the *Poisson equation* $-\Delta u = f$. Motivated by this, we define a solution (5) to be a *weak solution* to this Poisson equation, even when U is not smooth, and thus we have proved existence and uniqueness of weak solutions to this Poisson equation.

In fact, the constant coefficients and high degree of symmetry in Poisson's equation make it rather special: it can be solved by a variety of methods and in many cases its solutions can be represented by semi-explicit formulas. See [5] for a formalization in this spirit of some theory of the heat equation, another PDE with constant coefficients and a high degree of symmetry. The notable point of the method described above is that it does not really exploit these constant coefficients or symmetries, so it continues to work for a large class of other *elliptic second-order linear* partial differential equations.

The argument above is representative of the subject as a whole. Most PDEs do not admit explicit solutions. Instead, researchers prove nonconstructive existence, uniqueness and regularity theorems for solutions of such PDEs (and, in the best-case scenario, also prove results about the convergence properties of numerical methods for approximating these solutions). Inequalities such as the Gagliardo-Nirenberg-Sobolev inequality play a crucial role, in establishing the functional-analysis preconditions for the nonconstructive existence theorems which are invoked.

8 Related Work

We refer to [2, Section 1] for a survey of the available formalizations of the binary Tonelli and/or Fubini theorems.

The first work implementing integration on finitary product types in formal theorem provers was carried out by Harrison [11, Section 5], whose work in HOL Light covers the specific case of \mathbb{R}^n as part of a broader theory covering calculus on finite-dimensional vector spaces.

Hölzl and Heller [12] implemented integration on a general finitary product type as part of a full development of measure theory in the language Isabelle. Their framework for measure spaces is flexible: σ -algebras and measures are naturally defined on a subspace of a type. Similarly, when taking product measures, there is a subset of the indexing set that is considered when taking the product measure. This approach is similar to the marginal construction described in Section 4, since both allow for proofs by induction over the dimension. Such an induction is used to calculate the volume of the Euclidean ball in general dimension, formalized by Manuel Eberl. We re-formalize this in our own framework in Section 5.1.

The approaches are not the same; the marginal construction is more expressive than taking integrals with the notion of measure in Isabelle/HOL. Given a finite family of measurable spaces $(X_i)_{i \in \iota}$ and a measure μ_i on each X_i , the framework in Isabelle/HOL allows one to define the integral

$$\int_{\prod_{i \in A} X_i} f(X_A(\bar{x}, y)) \, d\prod_{i \in A} \mu_i(y)$$

(for sets A in ι , and for \bar{x} some fixed default element of $\prod_{i \in \iota} X_i$). In contrast, our marginal construction \bar{x} is a variable, which allows us to define the function

$$x \mapsto \int_{\prod_{i \in A} X_i} f(X_A(x, y)) \, d\prod_{i \in A} \mu_i(y).$$

In this comparison we have translated the concepts from Isabelle/HOL to our framework and notation, but because of differences in foundations, this translation is not exact.

The computation of the volume of a ball is simple enough that this can be conveniently done in Isabelle/HOL. For more complicated cases like the grid-lines lemma, a more expressive notion is needed. It would be interesting to see if the marginal construction can be conveniently adapted to Isabelle/HOL, by defining a variant of the product measure that depends on a point in the product space, used to take the “default values”.

References

- 1 Sylvie Boldo, François Clément, Florian Faissole, Vincent Martin, and Micaela Mayero. A Coq formal proof of the Lax–Milgram theorem. In *6th ACM SIGPLAN Conference on Certified Programs and Proofs*, Paris, France, January 2017. doi:10.1145/3018610.3018625.
- 2 Sylvie Boldo, François Clément, Vincent Martin, Micaela Mayero, and Houda Mouhcine. A Coq formalization of Lebesgue induction principle and Tonelli’s theorem. In *25th International Symposium on Formal Methods (FM 2023)*, volume 14000 of *Lecture Notes in Computer Science*, pages 39–55, Lübeck, Germany, March 2023. doi:10.1007/978-3-031-27481-7_4.
- 3 The mathlib Community. The Lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2020, pages 367–381, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3372885.3373824.
- 4 Leonardo Mendonça de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The Lean Theorem Prover (System Description). In *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*, pages 378–388, Cham, 2015. Springer International Publishing. doi:10.1007/978-3-319-21401-6_26.
- 5 Elif Deniz, Adnan Rashid, Osman Hasan, and Sofiène Tahar. On the formalization of the heat conduction problem in HOL. In *Intelligent Computer Mathematics: 15th International Conference, CICM 2022, Tbilisi, Georgia, September 19-23, 2022, Proceedings*, pages 21–37, Berlin, Heidelberg, 2022. Springer-Verlag. doi:10.1007/978-3-031-16681-5_2.
- 6 Frédéric Dupuis, Robert Y. Lewis, and Heather Macbeth. Formalized functional analysis with semilinear maps. *Journal of Automated Reasoning*, 237:10:1–10:19, 2022. doi:10.4230/LIPIcs.ITP.2022.10.
- 7 Lawrence C. Evans. *Partial differential equations*, volume 19 of *Grad. Stud. Math.* Providence, RI: American Mathematical Society (AMS), 2nd ed. edition, 2010.
- 8 David Gilbarg and Neil S. Trudinger. *Elliptic partial differential equations of second order*, volume 224 of *Grundlehren Math. Wiss.* Springer, Cham, 1977. doi:10.1007/978-3-642-61798-0.

- 9 Sébastien Gouëzel. A formalization of the change of variables formula for integrals in mathlib. In Kevin Buzzard and Temur Kutsia, editors, *Intelligent Computer Mathematics - 15th International Conference, CICM 2022, Tbilisi, Georgia, September 19-23, 2022, Proceedings*, volume 13467 of *Lecture Notes in Computer Science*, pages 3–18. Springer, 2022. doi:10.1007/978-3-031-16681-5_1.
- 10 John Harrison. Formalizing basic complex analysis. In R. Matuszewski and A. Zalewska, editors, *From Insight to Proof: Festschrift in Honour of Andrzej Trybulec*, volume 10(23) of *Studies in Logic, Grammar and Rhetoric*, pages 151–165. University of Białystok, 2007. URL: <http://mizar.org/trybulec65/>.
- 11 John Harrison. The HOL Light theory of Euclidean space. *Journal of Automated Reasoning*, 50(2):173–190, February 2013. doi:10.1007/s10817-012-9250-9.
- 12 Johannes Hölzl and Armin Heller. Three chapters of measure theory in Isabelle/HOL. In *Interactive Theorem Proving*, pages 135–151, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. doi:10.1007/978-3-642-22863-6_12.
- 13 Yury Kudryashov. Formalizing the divergence theorem and the Cauchy integral formula in lean. In June Andronick and Leonardo de Moura, editors, *13th International Conference on Interactive Theorem Proving (ITP 2022)*, volume 237 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 23:1–23:19, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ITP.2022.23.
- 14 Shibo Liu. Gagliardo-Nirenberg-Sobolev inequality: An induction proof. *The American Mathematical Monthly*, 130(9):859–861, 2023. doi:10.1080/00029890.2023.2240683.
- 15 Leonardo de Moura and Sebastian Ullrich. The Lean 4 theorem prover and programming language. In *Automated Deduction – CADE 28: 28th International Conference on Automated Deduction, Virtual Event, July 12–15, 2021, Proceedings*, pages 625–635, Berlin, Heidelberg, 2021. Springer-Verlag. doi:10.1007/978-3-030-79876-5_37.
- 16 Keiko Narita, Noboru Endou, and Yasunari Shidama. The orthogonal projection and the Riesz representation theorem. *Formaliz. Math.*, 23(3):243–252, 2015. doi:10.1515/forma-2015-0020.
- 17 L. Nirenberg. On elliptic partial differential equations. *Annali della Scuola Normale Superiore di Pisa - Scienze Fisiche e Matematiche*, Ser. 3, 13(2):115–162, 1959. URL: http://www.numdam.org/item/ASNSP_1959_3_13_2_115_0/.
- 18 Mao-Pei Tsui. Partial differential equations I. Lecture notes for Math 6540, University of Toledo, 2008. URL: <http://math.utoledo.edu/~mtsui/8540f08/hw/Sobolev-Inequality.pdf>.
- 19 Floris van Doorn. Formalized Haar measure. In Liron Cohen and Cezary Kaliszyk, editors, *12th International Conference on Interactive Theorem Proving (ITP 2021)*, volume 193 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 18:1–18:17, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ITP.2021.18.
- 20 Théo Zimmermann and Hugo Herbelin. Automatic and Transparent Transfer of Theorems along Isomorphisms in the Coq Proof Assistant. In *Conference on Intelligent Computer Mathematics*, Washington, D.C., United States, 2015. URL: <https://hal.science/hal-01152588>.

The Functor of Points Approach to Schemes in Cubical Agda

Max Zeuner  

Department of Mathematics, Stockholm University, Sweden

Matthias Hutzler 

Department of Computer Science and Engineering, Gothenburg University, Sweden

Abstract

We present a formalization of quasi-compact and quasi-separated schemes (qcqs-schemes) in the Cubical Agda proof assistant. We follow Grothendieck’s functor of points approach, which defines schemes, the quintessential notion of modern algebraic geometry, as certain well-behaved functors from commutative rings to sets. This approach is often regarded as conceptually simpler than the standard approach of defining schemes as locally ringed spaces, but to our knowledge it has not yet been adopted in formalizations of algebraic geometry. We build upon a previous formalization of the so-called Zariski lattice associated to a commutative ring in order to define the notion of compact open subfunctor. This allows for a concise definition of qcqs-schemes, streamlining the usual presentation as e.g. given in the standard textbook of Demazure and Gabriel. It also lets us obtain a fully constructive proof that compact open subfunctors of affine schemes are qcqs-schemes.

2012 ACM Subject Classification Theory of computation → Logic and verification; Theory of computation → Constructive mathematics; Theory of computation → Type theory

Keywords and phrases Schemes, Algebraic Geometry, Category Theory, Cubical Agda, Homotopy Type Theory and Univalent Foundations, Constructive Mathematics

Digital Object Identifier 10.4230/LIPIcs.ITP.2024.38

Related Version *Full Version*: <https://arxiv.org/abs/2403.13088>

Supplementary Material *Software (Agda Source Code)*: <https://github.com/agda/cubical/blob/60f18987bb1819a15fccc325343ef7b469bb2eeb/Cubical/Papers/FunctorialQcQsSchemes.agda>
archived at `swh:1:cnt:6c80e0d1208935c92986c8caf689b61365321b8b`

Funding This paper is based upon research supported by the Swedish Research Council (SRC, Vetenskapsrådet) under Grant No. 2019-04545.

Acknowledgements We would like to thank Felix Cherubini, Thierry Coquand and Anders Mörtberg for their support and feedback throughout this project.

1 Introduction

Algebraic geometry developed as the study of solutions to systems of polynomial equations. Objects of interest would e.g. be “affine complex varieties”, subsets of \mathbb{C}^n that can be described as the common roots of a finite system of polynomials $p_1, \dots, p_m \in \mathbb{C}[x_1, \dots, x_n]$. The discipline underwent a fundamental transformation during the latter half of the 20th century with the introduction of *schemes*. This development was spear-headed by Alexandre Grothendieck and led to many incredible achievements in geometry and number theory. Schemes can be seen as a generalization of varieties in several ways, but their standard presentation as “locally ringed spaces with an affine cover” somewhat blurs the connection to classical algebraic geometry, which can make it hard for students learning algebraic geometry to see in what sense schemes are “geometric” objects at all.



© Max Zeuner and Matthias Hutzler;

licensed under Creative Commons License CC-BY 4.0

15th International Conference on Interactive Theorem Proving (ITP 2024).

Editors: Yves Bertot, Temur Kutsia, and Michael Norrish; Article No. 38; pp. 38:1–38:18

Leibniz International Proceedings in Informatics



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

There is, however, a different angle for generalization, where the original motivation of studying solutions to polynomials keeps a more prominent place. Take a polynomial with integer coefficients like $x^n + y^n - z^n \in \mathbb{Z}[x, y, z]$. Fermat's last theorem tells us that this polynomial only has the trivial solution $x = y = z = 0$ for $n > 2$. This does of course only hold for solutions in the integers. We might interpret the same polynomial as living in a polynomial ring $A[x, y, z]$, where A is now any commutative ring (e.g. \mathbb{C}), and ask about solutions in A . The corresponding set of solutions is given by

$$V_{x^n+y^n-z^n}(A) = \{ (a_1, a_2, a_3) \in A^3 \mid a_1^n + a_2^n = a_3^n \}$$

Moreover, given a morphism of rings $\varphi \in \text{Hom}(A, B)$, we can map a solution in A , $(a_1, a_2, a_3) \in V_{x^n+y^n-z^n}(A)$, to a solution $(\varphi(a_1), \varphi(a_2), \varphi(a_3)) \in V_{x^n+y^n-z^n}(B)$ in B . In categorical terms, our polynomial defines a *functor* from the category of commutative rings to the category of sets, mapping a ring A to the set $V_{x^n+y^n-z^n}(A)$ of solutions in A .

This functor $V_{x^n+y^n-z^n}(_) : \text{CommRing} \rightarrow \text{Set}$ turns out to be a very familiar categorical object. For a ring A , homomorphisms $\text{Hom}(\mathbb{Z}[x, y, z], A)$ are in bijection with A^3 (every morphism is determined by its values on x, y and z) and this induces a bijection of morphisms $\text{Hom}(\mathbb{Z}[x, y, z]/\langle x^n+y^n-z^n \rangle, A)$ with $V_{x^n+y^n-z^n}(A)$. Now, a functor from CommRing to Set is nothing but a presheaf on the opposite category CommRing^{op} . In this presheaf category we can look at the Yoneda embedding or representable of the quotient ring $R = \mathbb{Z}[x, y, z]/\langle x^n+y^n-z^n \rangle$, which we will denote by $\text{Sp}(R)$. By the above argument, we get a natural isomorphism of presheaves $\text{Sp}(R) = \text{Hom}(R, _) \cong V_{x^n+y^n-z^n}(_)$.

In the category of functors from commutative rings to sets we can thus study solutions of systems of integer polynomials by looking at representable functors of quotients $\mathbb{Z}[x_1, \dots, x_n]/\langle p_1, \dots, p_m \rangle$. Algebraic geometers call these representables *absolute affine algebraic spaces* [16], which we can generalize to schemes (schemes over \mathbb{Z} or absolute schemes to be more precise). Affine schemes are readily defined as representables of arbitrary commutative rings. From these we can build general schemes as presheaves on CommRing^{op} that are “local” and have an “open cover” by affine schemes in some appropriate sense.

Among the proponents of using the functor of points approach as the primary definition of schemes was Grothendieck himself [16], because, in the terms of Lawvere [22], it does not require “the baggage of prime ideals and the spectral space, sheaves of local rings, coverings and patchings, etc.” Yet, most standard sources [13, 15, 17, 33] for students learning algebraic geometry start with precisely this “baggage”. To our knowledge, the same can be said for existing formalizations of schemes [3, 4, 5, 7, 39]. We want to close this gap and present a first formalization of the functor of points approach.

Admittedly, part of the appeal of schemes as locally ringed spaces as a formalization target for proof assistants is that they are such a layered, involved notion, while at the same time being a point of departure for formalizing a plethora of interesting research level mathematics. The first full formalization of schemes in Lean’s `mathlib` by Buzzard et. al. [4] revealed certain bottlenecks that occur when defining schemes this way. As these bottlenecks might be addressed very differently in different proof assistants, schemes have become somewhat of a benchmark problem, inspiring a formalization in Isabelle/HOL [3], and partial formalizations in Coq’s `UniMath` library [5] and Cubical Agda [39].

It is worth noting that, except for the Cubical Agda-formalization [39], all of the above formalizations are non-constructive as they follow the presentation of Hartshorne’s standard “Algebraic Geometry” [17]. In [39], the authors manage to stay constructive by using “ringed lattices” [8] instead of locally ringed spaces, but the formalization only includes affine schemes. The functor of points approach is often taken to be more amenable for constructive

mathematics.¹ Indeed, to our knowledge we present the first fully constructive formalization of quasi-compact and quasi-separated schemes (qcqs-schemes), an important subclass of schemes that is sufficient for a large portion of modern algebraic geometry.²

Nowadays there exist extensive algebra and category theory libraries for many of the major proof assistants, providing a lot of the necessary tools to formally define schemes using the functor of points approach. The bottlenecks of defining schemes as locally ringed spaces, disappear when following the functor of points approach. One problem that occurs, however, is that the category of functors from rings to sets is not locally small, since arrows between two such functors are natural transformations, i.e. families of functions indexed by the “big” type of all rings in a given universe. As a result, one has to address size issues. Dealing with size issues in a predicative type theory like Cubical Agda’s, one is led to make certain finiteness assumptions, resulting in the aforementioned restriction to qcqs-schemes.

Our work is completely formalized in Cubical Agda and all results are integrated in the `agda/cubical` library.³ We will comment on our usage of Cubical Agda in Section 2.1, but we want to stress that the formalization does not rely on cubical features. It should be possible to more or less directly translate the formalization into a system implementing Homotopy Type Theory and Univalent Foundations of the HoTT book [32] or into UniMath [35]. Our work can be understood as being in line with the goals of Voevodsky’s Foundations library [38]: Developing a library of constructive set-level mathematics based on Univalent Foundations.

As a result of working fully constructive and predicative, our presentation deviates from the standard “Introduction to Algebraic Geometry and Algebraic Groups” by Demazure and Gabriel [12]. Our main contributions and design choices can be summarized as follows:

- In Section 3 we define the category of \mathbb{Z} -functors, differing slightly from Demazure and Gabriel. This is because Agda’s universes are not cumulative and we chose to work with a fully-faithful spectrum functor with the caveat that it only has a relative adjoint.
- In Section 4 we define the notion of coverage and sheaf wrt. a coverage. We define the Zariski coverage on `CommRingop`. Restricting from \mathbb{Z} -functors to Zariski sheaves can be seen as introducing a locality condition, akin to restricting from ringed to locally ringed spaces. We show that affine schemes are local, i.e. that representable presheaves are sheaves wrt. the Zariski coverage. For this one can reuse some key algebraic lemmas, first formalized in [39] to show the sheaf property of the structure sheaf of an affine scheme.
- In Section 5 we define the notions of compact open subfunctor, cover of compact opens and finally qcqs-scheme. It is in this section that we deviate substantially from the standard sources. We argue that the above notions are most conveniently defined by using an appropriate classifier in the topos theoretic sense. Since we have a small Zariski

¹ See e.g. the discussion where the functor of points approach was first suggested as a formalization target for the `agda/cubical-library`: <https://github.com/agda/cubical/issues/657>

² In particular, every noetherian scheme is qcqs. When applying scheme theory to the classic motivating problems of algebraic geometry, Hartshorne notes that “practically all the schemes encountered in this way are noetherian” [17, p. 100]. Deligne’s presentation of étale cohomology [10], a crucial tool for his proof of the Weil conjectures, assumes schemes to be qcqs throughout: “We consider only schemes that are quasi-compact (= finite union of open affines) and quasi-separated (= such that the intersection of two open affines is quasi-compact), and we simply call them schemes.” [11, p. 1].

³ The formalization is summarized in:
<https://github.com/agda/cubical/blob/60f18987bb1819a15fccc325343ef7b469bb2eeb/Cubical/Papers/FunctorialQcQsSchemes.agda>
 This is a permalink to the library at the time of writing, which type-checks with Agda version 2.6.4.1. A clickable rendered version that might be subject to change can be found here:
<https://agda.github.io/cubical/Cubical.Papers.FunctorialQcQsSchemes.html>

lattice but no small type of radical ideals in Cubical Agda, we can only classify *compact opens*. So far, these only appear in the literature on synthetic algebraic geometry ([2, Def. 19.15] and [6, Def. 4.2.1]), but they turn out to be very useful for our purposes as well.

- In Section 6 we prove that compact open subfunctors of affine schemes are qcqs-schemes. We give a point-free proof that the classifier for compact opens is separated, only using the universal property of the Zariski lattice. This gives us that compact opens of affine schemes are sheaves. The fact that compact opens of affines have an affine cover essentially follows from the Yoneda lemma.

2 Background

We begin by giving some helpful background. First, we discuss the Cubical Agda proof assistant and how it is used in the formalization. We then briefly present two algebraic constructions from the `agda/cubical` library, first formalized and described by Zeuner and Mörtberg in [39], that play a key role in this paper as well: localizations of commutative rings and the Zariski lattice.

2.1 Univalent type theory in Cubical Agda

For understanding the details of our formalization, it is worth knowing about certain particularities of the Cubical Agda proof assistant and its library. We will restrict ourselves to the features that are relevant for this paper. Readers familiar with Cubical Agda or Homotopy Type Theory and Univalent Foundations (HoTT/UF) can safely skim this section. Readers interested in more details are referred to [34].

Cubical Agda is a rather recent extension of the Agda proof assistant with fully constructive support of the univalence principle and higher inductive types (HITs). The notation used in this paper is inspired by Agda’s syntax and the conventions of the `agda/cubical` library but we have taken the liberty to simplify the syntax and omit projections whenever possible in order to increase readability. For example we will write `CommRing` to denote both the type and the category of commutative rings and an element $R : \text{CommRing}$ will denote both the ring with its structure and the carrier-type of R , i.e. we write $f : R$ for its elements. For the universe at level ℓ we write `Type ℓ` or `Type $_{\ell}$` , and similarly `CommRing $_{\ell}$` for commutative rings whose carrier type lives in `Type $_{\ell}$` . For a family $B : A \rightarrow \text{Type } \ell$, we denote the dependent pair type over this family as $\Sigma [x \in A] B(x)$.

For definitional equalities we use $=$, while propositional equalities are written using \equiv . Note that Cubical Agda does not use Martin-Löf’s inductive identity type [25] for expressing propositional equalities, but rather so-called *path* types. These path types are defined in terms of a primitive interval type `I`, which allows one to conveniently define *dependent* path types. In this formalization we will not make direct use of the interval or dependent path types. However, path types do entail function extensionality, the right behavior of equalities of dependent pairs and other useful principles, which we will use freely.⁴

Cubical Agda does not come with a designated universe of propositions and in fact we cannot generally expect propositional equality types, or rather path types, to be propositions in any sensible way. This is because Cubical Agda proves univalence and thus disproves *Uniqueness of Identity Proofs* (UIP), also known as Streicher’s axiom K [31]. We can, however,

⁴ These principles also follow from univalence albeit with a slightly different computational behavior.

internally define (proof-relevant) propositions as subsingleton types and sets as types whose equalities are propositions, i.e. as types satisfying UIP:

```
isProp : Type ℓ → Type ℓ           isSet : Type ℓ → Type ℓ
isProp A = (x y : A) → x ≡ y      isSet A = (x y : A) → isProp (x ≡ y)
```

The type (universe) of propositions at level ℓ is defined as $\mathbf{hProp} \ell = \Sigma [A \in \mathbf{Type} \ell] (\mathbf{isProp} A)$, a *subset* of A , where $\mathbf{isSet} A$, is a function $S : A \rightarrow \mathbf{hProp} \ell$. With some abuse of notation we will identify a subset S with the corresponding Σ -type $\Sigma [a \in A] (a \in S)$, where $a \in S$ is the proposition (type of proofs) that a is actually in S . We thus write $a : S$ for elements of S when the proof of $a : A$ belonging to S can be ignored.

Univalence implies that there are types, which are neither propositions nor sets. These types are said to have a higher h-level (homotopy level [36]) than sets. One can use the so-called *structure identity principle* [32, Sec. 9.8] to prove that this holds true for types of algebraic or categorical structures like commutative rings or \mathbb{Z} -functors.⁵ However, we want to stress that this does not affect the formalization presented in this paper.

We do make extensive use higher inductive types (HITs), the other main addition of HoTT/UF to dependent type theory alongside univalence. In particular, we require two HITs: set-quotients and propositional truncations. Set-quotients are needed to define localizations of rings and the Zariski lattice, which we will describe in Section 2.2. We will not go into details on how set-quotients are defined. It suffices to know that as long as we quotient sets by proposition-valued equivalence relations and only consider maps from those quotients into other sets, everything works as one would expect from quotients. The other HIT, propositional truncation, turns any type into a proposition:

```
data ||_|| (A : Type ℓ) : Type ℓ where
  |_|| : A → || A ||
  squash : isProp || A ||
```

This is needed in HoTT/UF to express existential quantification, as using Σ -types is often too strong. We follow the convention and say “there *merely* exists $x : A$ such that $P(x)$ ”, if we have an inhabitant of

$$\exists [x \in A] P(x) = || \Sigma [x \in A] P(x) ||$$

Note that in general this does not let us extract a witness $x : A$, satisfying $P(x)$. We will discuss an example showcasing the proper use of propositional truncation in Remark 18.

2.2 Localizations and the Zariski lattice

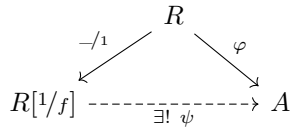
Our formalization builds on a lot of commutative algebra and category theory formalized in the `agda/cubical` library that we will presuppose in this paper. In particular, we will assume familiarity with presheaves, the Yoneda lemma and basic ideal theory of rings and we will not comment on their implementation in the `agda/cubical` library. There are two particular constructions, first described in [39], that are of special importance to this project and we will briefly describe them here.

⁵ The implementation of the structure identity principle in the `agda/cubical-library` is described in [1].

The first, localizations of commutative rings, are a way of making elements invertible by adding fractions. In this paper we only need the special case of inverting a single element. For a ring R and $f : R$ the *localization of R away from f* is the ring $R[1/f]$ of fractions r/f^n where the denominator is a power of f . Equality of two fractions is slightly different than for fractions of integers and can be stated as:⁶

$$r/f^n \equiv r'/f^m \quad \text{iff} \quad \exists [k \in \mathbb{N}] (r f^{k+m} \equiv r' f^{k+n})$$

Localizations satisfy a universal property and in our special case it can be stated as: $R[1/f]$ is the initial R -algebra where f becomes invertible. This means that for any ring A with a homomorphism $\varphi : \mathbf{Hom}(R, A)$ such that $\varphi(f) \in A^\times$ (i.e. $\varphi(f)$ is a unit/invertible), there is a unique $\psi : \mathbf{Hom}(R[1/f], A)$ making the following diagram commute



where $-/1 : \mathbf{Hom}(R, R[1/f])$ is the canonical morphism mapping $r : R$ to the fraction $r/1$. As shown in [39], formalizing localizations with the help of set-quotients is straightforward.

The second construction, the Zariski lattice associated to a ring is slightly more delicate. By a standard argument in classical algebraic geometry there is a one-to-one correspondence between Zariski open sets of $\mathbf{Spec}(R)$ and radical ideals of R . An ideal $I \subseteq R$ is radical if $I = \sqrt{I} = \{x \in R \mid \exists n > 0 : x^n \in I\}$. Furthermore, the *compact open* subsets of $\mathbf{Spec}(R)$ correspond radicals of *finitely generated* ideals. This correspondence is in fact an isomorphism of lattices. Set-theoretic union and intersection of compact opens correspond to addition and multiplication of finitely generated ideals.

This means that we can define this so-called *Zariski lattice* \mathcal{L}_R without having to define $\mathbf{Spec}(R)$ and its topology first: Elements of \mathcal{L}_R are generators $f_1, \dots, f_n : R$ quotiented by the relation that relates another list of generators $g_1, \dots, g_m : R$ if $\sqrt{\langle f_1, \dots, f_n \rangle} \equiv \sqrt{\langle g_1, \dots, g_m \rangle}$. The equivalence class of the generators $f_1, \dots, f_n : R$ is denoted by $D(f_1, \dots, f_n) : \mathcal{L}_R$ and the join on \mathcal{L}_R is given by $D(f_1, \dots, f_n) \vee D(g_1, \dots, g_m) = D(f_1, \dots, f_n, g_1, \dots, g_m)$. The “basic open” $D(f)$ is the equivalence class corresponding to the radical of the principle ideal $\sqrt{\langle f \rangle}$, with $D(1)$ being the top element of \mathcal{L}_R corresponding to the 1-ideal. The basic opens form a basis of \mathcal{L}_R , as $D(f_1, \dots, f_n) = \bigvee_{i=1}^n D(f_i)$.

This definition is due to Español [14], but it has the disadvantage that it uses equality of ideals to define the quotienting relation. In the predicative type theory of Cubical Agda the type of ideals of R lives in the universe above R and so does the equality type between two ideals. This can be avoided by slightly rewriting the equivalence relation, as shown in [39], giving us $\mathcal{L}_R : \mathbf{DistLattice}_\ell$ for $R : \mathbf{CommRing}_\ell$.

Joyal [20] observed that the Zariski lattice has a certain universal property that can be stated in terms of *supports*. A map $d : R \rightarrow L$ from R into a (bounded) distributive lattice L is called a support if it satisfies:

$$d(1) \equiv \top \quad \text{and} \quad d(0) \equiv \perp \tag{1}$$

$$\forall (f \ g : R) \rightarrow d(fg) \equiv d(f) \wedge d(g) \tag{2}$$

$$\forall (f \ g : R) \rightarrow d(f + g) \leq d(f) \vee d(g) \tag{3}$$

⁶ This is to account for zero-divisors and the case where f is nilpotent.

The map $D : R \rightarrow \mathcal{L}_R$ sending $f : R$ to the equivalence class $D(f)$ satisfies conditions (1)-(3) and it is a universal support in the sense for any other support $d : R \rightarrow L$ there is a unique lattice homomorphism $\varphi : \mathcal{L}_R \rightarrow L$ such that the following commutes

$$\begin{array}{ccc}
 & R & \\
 D \swarrow & & \searrow d \\
 \mathcal{L}_R & \overset{\exists! \varphi}{\dashrightarrow} & L
 \end{array}$$

The partial order defined on the Zariski lattice is connected to localizations as for $f, g : R$

$$D(g) \leq D(f) \Leftrightarrow \sqrt{\langle g \rangle} \subseteq \sqrt{\langle f \rangle} \Leftrightarrow g \in \sqrt{\langle f \rangle} \Leftrightarrow f/1 \in R[1/g]^\times$$

In the special case where $g = 1$, this gives us $D(1) \equiv D(f)$ iff $f \in R^\times$. We will utilize this fact in order to interpret the basic opens as affine subschemes in Definition 22. A slight generalization of this fact that we will use in Section 5 to informally justify that Definition 14 is sensible is that for $f_1, \dots, f_n : R$

$$D(1) \equiv D(f_1, \dots, f_n) \Leftrightarrow 1 \in \langle f_1, \dots, f_n \rangle$$

This concludes our discussion of the preliminaries required to formalize qcqs-schemes following the functor of points approach.

3 \mathbb{Z} -Functors

Let us turn to our goal of defining qcqs-schemes as well-behaved functors from rings to sets. As size issues are unavoidable in the functor of points approach, we will be rather explicit about universe levels in this paper. For the remainder we will fix a universe level ℓ and work over commutative rings in the corresponding universe $\mathbf{CommRing}_\ell$.

► **Definition 1.** *The category of \mathbb{Z} -functors, denoted $\mathbb{Z}\mathbf{Functor}_\ell$, is the category of functors from $\mathbf{CommRing}_\ell$ to \mathbf{Set}_ℓ . We write $\mathbf{Sp} : \mathbf{CommRing}_\ell^{op} \rightarrow \mathbb{Z}\mathbf{Functor}_\ell$ for the Yoneda embedding and $\mathbb{A}^1 : \mathbb{Z}\mathbf{Functor}_\ell$ for the forgetful functor from commutative rings to sets. We say that $X : \mathbb{Z}\mathbf{Functor}_\ell$ is an affine scheme if there merely exists $R : \mathbf{CommRing}_\ell$ such that $X \cong \mathbf{Sp}(R)$.*

► **Remark 2.** It is worth noticing that most modern algebraic geometry sources (see e.g. [13, 15, 26, 33]) usually omit any reference to universes when discussing the functor of points approach. The choice of taking functors from rings to sets in the same universe seems perhaps most natural, but actually differs from the standard reference on the functor of points approach by Demazure and Gabriel [12]. They essentially take \mathbb{Z} -functors to be functors from $\mathbf{CommRing}_\ell$ to $\mathbf{Set}_{\ell+1}$.⁷ Their “big spectrum functor” $\mathbf{Sp} : \mathbf{CommRing}_{\ell+1}^{op} \rightarrow (\mathbf{CommRing}_\ell \rightarrow \mathbf{Set}_{\ell+1})$ is defined much like the Yoneda embedding as $\mathbf{Sp}(R) = \mathbf{Hom}(R, _)$, but because of the universe level mismatch it is *not* fully faithful. However, this functor has a left adjoint, namely the functor that we will define in Definition 3. We decided to differ in our definition of \mathbb{Z} -functors since Agda’s non-cumulative universes would otherwise require explicit universe lifts in a lot of places, massively cluttering the code, and it seemed more convenient to use the fully-faithful Yoneda embedding as our \mathbf{Sp} .

⁷ They actually assume two Grothendieck universes $\mathcal{U} \subseteq \mathcal{V}$. As type theoretic universes are usually “lifted” from Grothendieck universes in presheaf models [18], our translation only seems natural.

► **Definition 3.** Let $X : \mathbb{Z}\text{Functor}_\ell$, the ring of functions $\mathcal{O}(X)$ is the type of natural transformations $X \Rightarrow \mathbb{A}^1$ equipped with the canonical point-wise operations, i.e. for $R : \text{CommRing}_\ell$ and $x : X(R)$, addition and multiplication of $\alpha, \beta : X \Rightarrow \mathbb{A}^1$ are given by

$$(\alpha + \beta)_R(x) = \alpha_R(x) + \beta_R(x) \quad (\alpha \cdot \beta)_R(x) = \alpha_R(x) \cdot \beta_R(x)$$

This defines a functor $\mathcal{O} : \mathbb{Z}\text{Functor}_\ell \rightarrow \text{CommRing}_{\ell+1}^{\text{op}}$, whose action on morphisms (natural transformations) is given by precomposition.

The universal property of schemes is often stated to be: The global sections functor Γ is left adjoint to Spec and the counit of this adjunction is an isomorphism. However, this is already true for locally ringed spaces. In a similar fashion we would like to have an adjunction $\mathcal{O} \dashv \text{Sp}$, but unfortunately we run into a universe level mismatch. We still get something that looks a lot like an adjunction. The proof of the following proposition is straightforward.

► **Proposition 4.** For $R : \text{CommRing}_\ell$ and $X : \mathbb{Z}\text{Functor}_\ell$ there is an isomorphism of types

$$\text{Hom}(R, \mathcal{O}(X)) \cong (X \Rightarrow \text{Sp}(R))$$

which is natural in both R and X . Moreover, the induced “counit” $\varepsilon_R : \text{Hom}(R, \mathcal{O}(\text{Sp}(R)))$, which is obtained by applying the inverse of above isomorphism to the identity transformation $\text{Sp}(R) \Rightarrow \text{Sp}(R)$, is an isomorphism of rings for all $R : \text{CommRing}_\ell$.

► **Remark 5.** Proposition 4 type-checks because the type of ring homomorphisms is universe polymorphic, meaning it can take rings living in different universes as arguments. The same holds for the type of isomorphisms/equivalences between two types. From a categorical perspective, we get a so-called *relative coadjunction* [29], written $\mathcal{O} \dashv_i \text{Sp}$, with respect to the inclusion, or lift functor $i : \text{CommRing}_\ell^{\text{op}} \rightarrow \text{CommRing}_{\ell+1}^{\text{op}}$. This is why we only get a counit, but no unit.

4 Local \mathbb{Z} -functors

Functorial (qcqs-) schemes are sheaves with respect to the Zariski coverage. The notion of coverage (also called a Grothendieck pre-topology) generalizes point-set topologies to arbitrary categories. Roughly speaking, a coverage on a category \mathcal{C} associates to each object $U : \mathcal{C}$ a family $\text{Cov}(U)$ of covers. A cover $(U_i \rightarrow U)_{i:I} : \text{Cov}(U)$ is a family of maps into U . These families $\text{Cov}(U)$ should satisfy certain closure properties. If \mathcal{C} has pullbacks then covers should be closed under pullbacks and a presheaf $\mathcal{F} : \mathcal{C}^{\text{op}} \rightarrow \text{Set}$ can be defined to be a sheaf if for any $(U_i \rightarrow U)_{i:I} : \text{Cov}(U)$ we get an equalizer diagram

$$\mathcal{F}(U) \rightarrow \prod_{i:I} \mathcal{F}(U_i) \rightrightarrows \prod_{i,j:I} \mathcal{F}(U_i \times_U U_j)$$

In the case where \mathcal{C} is $\text{Open}(X)$, the poset of open subsets of a topological space X , we get a canonical coverage: A family of opens $(U_i \subseteq U)_{i:I}$ is in $\text{Cov}(U)$ if and only if $\bigcup_{i \in I} U_i = U$. Pullbacks in $\text{Open}(X)$ are given by set-theoretic intersection \cap and we recover the usual definition of when a presheaf $\mathcal{F} : \text{Open}(X)^{\text{op}} \rightarrow \text{Set}$ is a sheaf.

The formalization of coverages and sheaves in the `agda/cubical` library follows the nLab [27] and Johnstone’s classic “Sketches of an elephant” [19, C2]. The advantage of this approach is that it even works for categories without pullbacks. As it turns out, it also lets us conveniently define the Zariski coverage and prove that representables are Zariski sheaves. For now, let us fix an arbitrary category \mathcal{C} .

► **Definition 6.** A cover on an object $c : \mathcal{C}$ consists of an index type I and for each $i : I$ an element in the slice category \mathcal{C}/c , i.e. an arrow $f_i : \mathcal{C}(c_i, c)$. A coverage on \mathcal{C} consist of a family of covers for each $c : \mathcal{C}$ satisfying pullback stability: Given a cover $\{f_i : \mathcal{C}(c_i, c)\}_{i:I}$ of c and arrow $f : \mathcal{C}(d, c)$, there merely exists a cover $\{g_j : \mathcal{C}(d_j, d)\}_{j:J}$ of d such that for each index $j : J$ there merely exists an index $i : I$ and an arrow $h_{ij} : \mathcal{C}(d_j, c_i)$ with $f_i \circ h_{ij} \equiv f \circ g_j$.

Pullback stability can also be stated as: Given an arrow $f : \mathcal{C}(d, c)$ and a cover on c , we can take the sieve generated by this cover and pull it back to a sieve on d . Then there exists a cover on d refining the pulled back sieve on d . Since sieves are not required for the remainder of the paper, we decided to unfold the definition of pullback stability and state it without recourse to sieves. We refer the interested reader to the formalization. We now define what it means to be sheaf with respect to a fixed coverage on \mathcal{C} . For a presheaf P on \mathcal{C} and arrow $f : \mathcal{C}(c, d)$ we write $_ \downarrow_f : P(d) \rightarrow P(c)$ for the restriction map, i.e. the action of P on f .

► **Definition 7.** Let P be a presheaf on \mathcal{C} . Let $c : \mathcal{C}$ and $\{f_i : \mathcal{C}(c_i, c)\}_{i:I}$ be a cover. A compatible family or matching family [28] is a dependent function $x : (i : I) \rightarrow P(c_i)$, i.e. a family of elements $x_i : P(c_i)$, such that for each pair of indices $i, j : I$ and arrows $g_i : \mathcal{C}(d, c_i)$ and $g_j : \mathcal{C}(d, c_j)$ with $f_j g_j \equiv f_i g_i$, we have $x_j \downarrow_{g_j} \equiv x_i \downarrow_{g_i}$ (in $P(d)$). We denote the type of compatible families over a cover $\{f_i : \mathcal{C}(c_i, c)\}_{i:I}$ by $\text{CompatibleFam}^P(\{f_i : \mathcal{C}(c_i, c)\}_{i:I})$.

For an element $x : P(c)$ we get an induced compatible family by taking the restrictions $x_i = x \downarrow_{f_i}$ for $i : I$. The compatibility follows directly from the presheaf property of P . This construction gives us a map $\sigma_P : P(c) \rightarrow \text{CompatibleFam}^P(\{f_i : \mathcal{C}(c_i, c)\}_{i:I})$. We can now conveniently define sheaves in terms of the map σ .

► **Definition 8.** A presheaf P is a sheaf if for all $c : \mathcal{C}$ and covers $\{f_i : \mathcal{C}(c_i, c)\}_{i:I}$, the canonical map σ_P is an isomorphism.

► **Definition 9.** A coverage on \mathcal{C} is called subcanonical if for all $c : \mathcal{C}$ the Yoneda embedding of c is a sheaf with respect to the coverage.

In this paper we are interested in a particular example of a coverage on the opposite category of commutative rings. Covers of a ring R will come from finite lists of generators of the 1-ideal. Classically, this corresponds to the fact that any open cover of an affine scheme is of the form $\text{Spec}(R) = \bigcup_{i=1}^n D(f_i)$ with $1 \in \langle f_1, \dots, f_n \rangle$ (because $\text{Spec}(R)$ is quasi-compact). We call a finite list of elements $f_1, \dots, f_n : R$ such that $1 \in \langle f_1, \dots, f_n \rangle$ a *unimodular vector*.

► **Definition 10.** The Zariski coverage on $\mathcal{C} = \text{CommRing}_\ell^{\text{op}}$ is given by:

- For each $R : \text{CommRing}_\ell$, covers are indexed by the type of unimodular vectors over R .
- For each unimodular vector $f_1, \dots, f_n : R$, the associated cover of R is given by the reversed canonical morphisms $-/1 : R[1/f_i] \rightarrow R$, indexed by $i : \text{Fin } n$, the finite n -element type.
- For a unimodular vector $f_1, \dots, f_n : R$ the pullback along a morphism $\varphi : \text{Hom}(R, A)$ is the vector $\varphi(f_1), \dots, \varphi(f_n) : A$, which is easily shown to be unimodular as well.

A presheaf $X : \mathbb{Z}\text{Functor}_\ell$ is called local if it is a sheaf wrt. the Zariski coverage.

► **Lemma 11.** The Zariski coverage is stable under pullbacks.

Proof. Let $R, A : \text{CommRing}_\ell$, $f_1, \dots, f_n : R$ be a unimodular vector and $\varphi : \text{Hom}(R, A)$. The universal property of localization induces ring morphisms $\psi_i : \text{Hom}(R[1/f_i], A[1/\varphi(f_i)])$ such that the following diagram commutes (in $\text{CommRing}_\ell^{\text{op}}$)

$$\begin{array}{ccc}
 A[1/\varphi(f_i)] & \xrightarrow{\psi_i} & R[1/f_i] \\
 \downarrow -/1 & & \downarrow -/1 \\
 A & \xrightarrow{\varphi} & R
 \end{array}
 \quad \blacktriangleleft$$

38:10 The Functor of Points Approach to Schemes in Cubical Agda

The key result of this section uses an algebraic fact that can be found in many textbooks, such as [23, p. 125], and was already formalized in Cubical Agda to prove [39, Lemma 15].

► **Theorem 12.** *The Zariski coverage is subcanonical, i.e. $\mathbf{Sp}(A)$ is local for $A : \mathbf{CommRing}_\ell$.*

Proof. Let $R : \mathbf{CommRing}_\ell$ and $f_1, \dots, f_n : R$ a unimodular vector be given. For i, j in $1, \dots, n$, we denote by $\chi_{ij}^l : R[1/f_i] \rightarrow R[1/f_i f_j]$ and $\chi_{ij}^r : R[1/f_j] \rightarrow R[1/f_i f_j]$ the canonical morphisms given by the universal property of localization. We use without proof that the map

$$R \rightarrow \Sigma [x \in (i : \mathbf{Fin} \ n) \rightarrow R[1/f_i]] \quad \forall i \ j \rightarrow \chi_{ij}^l(x_i) \equiv \chi_{ij}^r(x_j)$$

sending $g : R$ to $g/1 : R[1/f_i]$ for $i = 1, \dots, n$, is an isomorphism. Using this, one can construct a chain of isomorphisms

$$\begin{aligned} \mathbf{Hom}(A, R) &\cong \Sigma [\varphi \in (i : \mathbf{Fin} \ n) \rightarrow \mathbf{Hom}(A, R[1/f_i])] \quad \forall i \ j \rightarrow \chi_{ij}^l \circ \varphi_i \equiv \chi_{ij}^r \circ \varphi_j \\ &\cong \mathbf{CompatibleFam}^{\mathbf{Sp}(A)}(\{f_i\}_{i=1, \dots, n}) \end{aligned}$$

which factors through the canonical map $\sigma_{\mathbf{Sp}(A)}$. ◀

5 Compact opens and qcqs-schemes

The standard way to define open subfunctors follows a two step process. First, one defines them for representables using (radical) ideals. Then, one defines open subfunctors of general \mathbb{Z} -functors by pulling back to representables. Working predicatively in Cubical Agda, we need to restrict ourselves to *finitely generated* ideals, which gives us compact open subfunctors. Let us sketch the idea behind compact opens informally to see why this restriction is necessary: For a f.g. ideal $I \subseteq A$, we get the *affine compact open subfunctor* $\mathbf{Sp}(A)_I \hookrightarrow \mathbf{Sp}(A)$ given by

$$\mathbf{Sp}(A)_I(B) = \{\varphi \in \mathbf{Hom}(A, B) \mid \varphi^* I = B\} \subseteq \mathbf{Sp}(A)(B)$$

If $I = \langle f_1, \dots, f_n \rangle$, then the “pullback” along $\varphi \in \mathbf{Hom}(A, B)$ is just $\varphi^* I = \langle \varphi(f_1), \dots, \varphi(f_n) \rangle$. With this, we can define a subfunctor $U \hookrightarrow X$ to be *compact open* if pulling back along an A -valued point of X gives an affine compact open subfunctor of $\mathbf{Sp}(A)$, i.e. if for any ring A and $\phi : \mathbf{Sp}(A) \Rightarrow X$ there is a f.g. ideal $I \subseteq A$ such that the following is a pullback square

$$\begin{array}{ccc} \mathbf{Sp}(A)_I & \longrightarrow & U \\ \downarrow & \lrcorner & \downarrow \\ \mathbf{Sp}(A) & \xrightarrow{\phi} & X \end{array}$$

Note that the ideal I is not uniquely determined in this case. Indeed, if $I = \langle f_1, \dots, f_n \rangle$ and $J = \langle g_1, \dots, g_m \rangle$ are such that $\sqrt{I} = \sqrt{J}$, then for any $\varphi \in \mathbf{Hom}(A, B)$ we have

$$1 \in \langle \varphi(f_1), \dots, \varphi(f_n) \rangle \quad \text{iff} \quad 1 \in \langle \varphi(g_1), \dots, \varphi(g_m) \rangle$$

and thus $\mathbf{Sp}(A)_I \cong \mathbf{Sp}(A)_J$. In fact, one can prove that the converse also holds. This means that the compact open U and the A -valued point $\phi : \mathbf{Sp}(A) \Rightarrow X$ determine a finitely generated ideal $I = \langle f_1, \dots, f_n \rangle$ up to equality of radical ideals, i.e. an element $D(f_1, \dots, f_n)$ of the Zariski lattice \mathcal{L}_A . Note that we can describe $\mathbf{Sp}(A)_I$ purely in terms of $D(f_1, \dots, f_n)$, as the B -valued points are given by

$$\begin{aligned} \mathbf{Sp}(A)_I(B) &= \{\varphi \in \mathbf{Hom}(A, B) \mid 1 \in \langle \varphi(f_1), \dots, \varphi(f_n) \rangle\} \\ &= \{\varphi \in \mathbf{Hom}(A, B) \mid D(\varphi(f_1), \dots, \varphi(f_n)) = D(1)\} \end{aligned}$$

The pullback condition ensures that this mapping is natural in A . In other words, the compact open subfunctors of X are in one-to-one correspondence with natural transformations from X to the \mathbb{Z} -functor \mathcal{L} that sends a ring to its Zariski lattice. Note that we can define this \mathbb{Z} -functor \mathcal{L} because of the “small” definition of Zariski lattice. If we drop the finiteness assumption on ideals to get open subfunctors we cannot hope to define the classifier in Cubical Agda.⁸ We will discuss possibilities to do so in other systems in Section 7.1.

For a topos theorist it might not constitute a particularly deep insight that the compact open subfunctors (sub-objects) of a \mathbb{Z} -functor are *classified* by the “internal Zariski lattice” \mathcal{L} . This means that the compact opens are precisely given by pullbacks of the form

$$\begin{array}{ccc} U & \longrightarrow & \mathbf{1} \\ \downarrow & \lrcorner & \downarrow D(1) \\ X & \longrightarrow & \mathcal{L} \end{array}$$

where $D(1) : \mathbf{1} \Rightarrow \mathcal{L}$ is the “constant” natural transformation, sending the point of the terminal \mathbb{Z} -functor $\mathbf{1}$ to the top element of the Zariski lattice. From a formal perspective however, we found it significantly more convenient to work with natural transformations into \mathcal{L} and the induced subfunctors, as opposed to following the text-book strategy of defining compact-openness as a property of subfunctors through the two step process outlined above.⁹ We will thus proceed to describe how compact opens can be formally defined as natural transformations and how this gives a concise definition of qcqs-schemes.

► **Definition 13.** Let $\mathcal{L} : \mathbb{Z}\mathbf{Func}_\ell$ be the \mathbb{Z} -functor mapping a ring $R : \mathbf{CommRing}_\ell$ to the underlying set of the Zariski lattice \mathcal{L}_R . The action on morphisms is induced by the universal property of the Zariski lattice, i.e. for $\varphi : \mathbf{Hom}(A, B)$ we take the morphism $\varphi^\mathcal{L}$ induced by the support $D \circ \varphi$:

$$\begin{array}{ccc} & A & \\ D \swarrow & & \searrow D \circ \varphi \\ \mathcal{L}_A & \overset{\exists! \varphi^\mathcal{L}}{\dashrightarrow} & \mathcal{L}_B \end{array}$$

► **Definition 14.** Let $X : \mathbb{Z}\mathbf{Func}_\ell$, a compact open of X is a natural transformation $U : X \Rightarrow \mathcal{L}$. The realization $\llbracket U \rrbracket^\mathbf{co} : \mathbb{Z}\mathbf{Func}_\ell$ of a compact open U of X , is given by

$$\llbracket U \rrbracket^\mathbf{co}(R) = \Sigma [x \in X(R)] U(x) \equiv D(1)$$

A compact open U is called affine, if its realization is affine, i.e. if there merely exists $R : \mathbf{CommRing}_\ell$ such that $\llbracket U \rrbracket^\mathbf{co} \cong \mathbf{Sp}(R)$.

The reader may verify that for $U : X \Rightarrow \mathcal{L}$, $R : \mathbf{CommRing}_\ell$ and $x : X(R)$ such that $U(x) = D(f_1, \dots, f_n)$, we have

$$\begin{array}{ccccc} \mathbf{Sp}(R)_{\langle f_1, \dots, f_n \rangle} & \longrightarrow & \llbracket U \rrbracket^\mathbf{co} & \longrightarrow & \mathbf{1} \\ \downarrow & \lrcorner & \downarrow & \lrcorner & \downarrow D(1) \\ \mathbf{Sp}(R) & \xrightarrow{\phi_x} & X & \xrightarrow{U} & \mathcal{L} \end{array}$$

⁸ In fact, having such a classifier would imply propositional resizing. By a result of De Jong and Escardó [9, Cor. 28], the existence of a single ring $R : \mathbf{CommRing}_\ell$, such that the frame of radical ideals of R (i.e. Zariski opens) is ℓ -small, would suffice to prove resizing for \mathbf{hProp}_ℓ .

⁹ A rare exception to following the standard definition is a blog-post by Madore [24].

38:12 The Functor of Points Approach to Schemes in Cubical Agda

where ϕ_x corresponds to the R -valued point x by the Yoneda lemma.

Since \mathcal{L} is a presheaf that takes values in distributive lattices and its restriction maps are lattice morphisms, it is an *internal* lattice in the presheaf topos of \mathbb{Z} -functors.¹⁰ As such, it endows the compact opens with a distributive lattice structure.

► **Definition 15.** Let $X : \mathbb{Z}\text{Functor}_\ell$, the lattice of compact opens of X , $\text{CompOpen}(X)$, is the type $X \Rightarrow \mathcal{L}$ equipped with the canonical point-wise operations, i.e. for $R : \text{CommRing}_\ell$ and $x : X(R)$, top, bottom, join and meet are given by

$$\begin{aligned} \top_R(x) &= D(1), & \perp_R(x) &= D(0) \\ (U \wedge V)_R(x) &= U_R(x) \wedge V_R(x) \\ (U \vee V)_R(x) &= U_R(x) \vee V_R(x) \end{aligned}$$

This defines a functor $\text{CompOpen} : \mathbb{Z}\text{Functor}_\ell \rightarrow \text{DistLattice}_{\ell+1}^{\text{op}}$. With the action on morphisms given by pre-composition.

► **Definition 16.** $X : \mathbb{Z}\text{Functor}_\ell$ is a qcqs-scheme if it is a local \mathbb{Z} -functor and has an affine cover by compact opens. That is, there merely exist compact opens $U_1, \dots, U_n : X \Rightarrow \mathcal{L}$ such that each U_i is affine and $\top \equiv \bigvee_{i=1}^n U_i$ in the lattice $\text{CompOpen}(X)$.

As an immediate sanity check we get that affine schemes are qcqs-schemes:

► **Proposition 17.** $\text{Sp}(R)$ is a qcqs-scheme, for $R : \text{CommRing}_\ell$.

Proof. $\text{Sp}(R)$ is local by Theorem 12. The top element $\top : \text{CompOpen}(\text{Sp}(R))$ is the “constant” natural transformation, sending everything to $D(1)$, which by the Yoneda lemma corresponds to $D(1) : \mathcal{L}_R$. It thus constitutes a trivial affine cover with $\llbracket \top \rrbracket^{\text{co}} \cong \text{Sp}(R)$. ◀

► **Remark 18.** Of course, a qcqs-scheme X can and will have multiple different covers. Definition 16 suggests that “having an affine cover” should be expressed using nested mere existential quantification. In practice, it is more convenient to define the record-type `AffineCover`, of all affine covers of X , consisting of a finite list or vector of compact opens and proofs that these compact opens are affine and cover X . The property of having an affine cover is then defined as the truncation of this record type:

```
record AffineCover (X :  $\mathbb{Z}\text{Functor } \ell$ )
  : Type ( $\ell\text{-suc } \ell$ ) where
  hasAffineCover :  $\mathbb{Z}\text{Functor } \ell \rightarrow \text{Type } (\ell\text{-suc } \ell)$ 
  hasAffineCover X =  $\llbracket \text{AffineCover } X \rrbracket$ 

  field
  n :  $\mathbb{N}$ 
  U :  $\text{FinVec } (\text{CompactOpen } X) \ n$ 
  covers :  $\text{isCompactOpenCover } X \ U$ 
  isAffineU :  $\forall i \rightarrow \text{isAffineCompactOpen } (U \ i)$ 
```

If we want to prove a proposition about a qcqs-scheme X , we can assume we have a witness of type `AffineCover(X)`. If we want to map from X into a set by using that X has an affine cover, we have to show that the mapping is independent of the choice of cover.¹¹ This is very much in line with informal mathematical practice.

¹⁰ It is even an internal lattice the big Zariski topos, i.e. in local \mathbb{Z} -functors. However for our purposes, we do not need that \mathcal{L} is a Zariski sheaf.

¹¹ This holds by the general *elimination* principle of the propositional truncation due to Kraus [21]. When mapping into types that are not sets, things get complicated very quickly.

► Remark 19. One big advantage of using the internal lattice \mathcal{L} to classify compact opens is that we get the notion of cover for free from the induced lattice operations in Definition 15. In textbooks, a cover by open subfunctors is usually defined directly using addition of ideals [12, 26] or by taking the set-theoretic union at field-valued points [13]. The latter is not an option for our purposes, as the notion of field is not well-behaved constructively. In the Cubical Agda library, the join $_ \vee _ : \mathcal{L}_A \rightarrow \mathcal{L}_A \rightarrow \mathcal{L}_A$ is also defined in terms of ideal addition, but we can upstream the necessary constructions and do not have to concern ourselves with pullbacks of \mathbb{Z} -functors.

6 Open subschemes

The benchmark for a workable formal definition of schemes as locally ringed spaces, as in [3, 4], usually consists of a proof of the “universal property”, i.e. an adjunction $\Gamma \dashv \text{Spec}$ where the counit is an isomorphism. Proposition 4, the functorial analogue is rather straightforward to prove. Instead, we give a proof that compact opens of affine schemes are qcqs-schemes. This can be seen as a constructive special case of the standard classical result that “open subfunctors of schemes are themselves schemes” [12, Ch. I, §1, 3.11]. We start by showing that compact opens of Zariski sheaves are Zariski sheaves. Essentially, this holds because compact opens are classified by \mathcal{L} , which is itself a Zariski sheaf. As it turns out, however, it is sufficient to prove something weaker.

For the remainder of the paper we adopt the following notation: For a ring R and elements $f : R$ and $u : \mathcal{L}_R$, we write $u \upharpoonright_{R[1/f]} : \mathcal{L}_{R[1/f]}$ for the result of applying $\mathcal{L}_{(-/1)}$, the \mathcal{L} -action on the canonical morphism. In particular we have $D(g_1, \dots, g_m) \upharpoonright_{R[1/f]} = D(g_1/1, \dots, g_m/1)$.

► Lemma 20. \mathcal{L} is Zariski-separated, i.e. for $R : \text{CommRing}_\ell$ and $f_1, \dots, f_n : R$ unimodular the following holds: given $u, v : \mathcal{L}_R$, if $u \upharpoonright_{R[1/f_i]} \equiv v \upharpoonright_{R[1/f_i]}$ for all $i = 1, \dots, n$, then $u \equiv v$.

Proof. Let $R : \text{CommRing}_\ell$ and $f_1, \dots, f_n : R$ unimodular be given together with $u, v : \mathcal{L}_R$ satisfying $u \upharpoonright_{R[1/f_i]} \equiv v \upharpoonright_{R[1/f_i]}$ for all $i = 1, \dots, n$. Recall that for $i = 1, \dots, n$, the restriction $_ \upharpoonright_{R[1/f_i]} : \mathcal{L}_R \rightarrow \mathcal{L}_{R[1/f_i]}$ is induced by the support $D(-/1) : R \rightarrow \mathcal{L}_{R[1/f_i]}$. Now let us fix an $i = 1, \dots, n$. Much like in classical algebraic geometry, we can identify $\mathcal{L}_{R[1/f_i]}$ with $\downarrow D(f_i)$, the lattice of elements of \mathcal{L}_R smaller than $D(f_i)$.¹² The map $d : R[1/f_i] \rightarrow \downarrow D(f_i)$ given by $d(r/f_i^n) = D(r) \wedge D(f_i)$ defines a support and thus induces a morphism $\varphi : \mathcal{L}_{R[1/f_i]} \rightarrow \downarrow D(f_i)$.

Now, consider the map $_ \wedge D(f_i) : \mathcal{L}_R \rightarrow \downarrow D(f_i)$. We claim that $_ \wedge D(f_i)$ factors through φ . By the universal property of \mathcal{L}_R , there is a unique $\psi : \mathcal{L}_R \rightarrow \downarrow D(f_i)$, such that $\psi \circ D \equiv d(-/1)$. Both $_ \wedge D(f_i)$ and $\varphi(_ \upharpoonright_{R[1/f_i]})$ satisfy the same commutativity condition as ψ , which implies $_ \wedge D(f_i) \equiv \psi \equiv \varphi(_ \upharpoonright_{R[1/f_i]})$. Pictorially, this amounts to observing that the following diagram commutes

$$\begin{array}{ccccc}
 R & \xrightarrow{-/1} & R[1/f_i] & & \\
 D \downarrow & & D \downarrow & \searrow d & \\
 \mathcal{L}_R & \xrightarrow{- \upharpoonright_{R[1/f_i]}} & \mathcal{L}_{R[1/f_i]} & \xrightarrow{\varphi} & \downarrow D(f_i) \\
 & \searrow _ \wedge D(f_i) & & &
 \end{array}$$

From our assumption it thus follows that

$$u \wedge D(f_i) \equiv \varphi(u \upharpoonright_{R[1/f_i]}) \equiv \varphi(v \upharpoonright_{R[1/f_i]}) \equiv v \wedge D(f_i)$$

¹²Showing that $\text{Spec}(R[1/f])$ is homeomorphic to $D(f)$ is a standard exercise in algebraic geometry.

38:14 The Functor of Points Approach to Schemes in Cubical Agda

for all $i = 1, \dots, n$. Since the f_i 's are unimodular, we get $D(1) \equiv \bigvee_i D(f_i)$ and hence

$$u \equiv u \wedge D(1) \equiv \bigvee_{i=1}^n (u \wedge D(f_i)) \equiv \bigvee_{i=1}^n (v \wedge D(f_i)) \equiv v \wedge D(1) \equiv v \quad \blacktriangleleft$$

► **Lemma 21.** *If $X : \mathbb{Z}\text{Func}_\ell$ is local, then for any compact open $U : X \Rightarrow \mathcal{L}$ its realization $\llbracket U \rrbracket^{\text{co}} : \mathbb{Z}\text{Func}_\ell$ is local.*

Proof. Let $R : \text{CommRing}_\ell$ and $f_1, \dots, f_n : R$ be unimodular. We need to construct an inverse to the map

$$\sigma_U : \Sigma [x \in X(R)] U(x) \equiv D(1) \rightarrow \text{CompatibleFam}^{\llbracket U \rrbracket^{\text{co}}}(\{f_i\}_{i=1, \dots, n})$$

For $x : X(R)$ with $U(x) \equiv D(1)$, $\sigma_U(x)$ is the family of elements $x \upharpoonright_{R[1/f_i]}$. It is essentially the same map as the corresponding

$$\sigma_X : X(R) \rightarrow \text{CompatibleFam}^X(\{f_i\}_{i=1, \dots, n})$$

but it keeps track of the fact that for each $i = 1, \dots, n$ one has $U(x \upharpoonright_{R[1/f_i]}) \equiv D(1)$.

Now, any compatible family of elements $x_i : X(R[1/f_i])$ with $U(x_i) \equiv D(1)$ can be seen as a compatible family on X by forgetting that $U(x_i) \equiv D(1)$. To this family we apply the inverse map

$$\sigma_X^{-1} : \text{CompatibleFam}^X(\{f_i\}_{i=1, \dots, n}) \rightarrow X(R)$$

that exists since X was assumed local. We claim that $U(\sigma_X^{-1}(\{x_i\}_{i=1, \dots, n})) \equiv D(1)$, thus allowing us to set $\sigma_U^{-1}(\{x_i\}_{i=1, \dots, n}) = \sigma_X^{-1}(\{x_i\}_{i=1, \dots, n})$. From this it also follows immediately that σ_U and σ_U^{-1} are mutually inverse. To prove the claim we use Lemma 20 and the fact that for each $i = 1, \dots, n$:

$$\begin{aligned} U(\sigma_X^{-1}(\{x_i\}_{i=1, \dots, n})) \upharpoonright_{R[1/f_i]} &\equiv U(\sigma_X^{-1}(\{x_i\}_{i=1, \dots, n}) \upharpoonright_{R[1/f_i]}) \\ &\equiv U(\sigma_X(\sigma_X^{-1}(\{x_i\}_{i=1, \dots, n}))_i) \equiv U(x_i) \equiv D(1) \end{aligned} \quad \blacktriangleleft$$

It remains to prove that compact opens of affine schemes (merely) have an affine cover. Before treating arbitrary compact opens, we introduce the standard or basic opens of a representable \mathbb{Z} -functor with fair bit of abuse of notation.

► **Definition 22.** *Let $R : \text{CommRing}_\ell$ and $f : R$, the standard open $D(f) : \text{Sp}(R) \Rightarrow \mathcal{L}$ is given by applying the Yoneda lemma to the basic open $D(f) : \mathcal{L}_R$.*

► **Proposition 23.** *For $R : \text{CommRing}_\ell$ and $f : R$, the standard open $D(f)$ is affine. In particular one has a natural isomorphism $\llbracket D(f) \rrbracket^{\text{co}} \cong \text{Sp}(R[1/f])$.*

Proof. The universal properties of localization and Zariski lattice give us for A -valued points

$$\begin{aligned} \text{Sp}(R[1/f])(A) &= \text{Hom}(R[1/f], A) \\ &\cong \Sigma [\varphi \in \text{Hom}(R, A)] \varphi(f) \in A^\times \\ &\cong \Sigma [\varphi \in \text{Hom}(R, A)] D(\varphi(f)) \equiv D(1) \\ &= \llbracket D(f) \rrbracket^{\text{co}}(A) \end{aligned}$$

We omit the proof that this is natural in A . ◀

► **Theorem 24.** *The realization $\llbracket U \rrbracket^{\text{co}}$ of a compact open $U : \text{Sp}(R) \Rightarrow \mathcal{L}$ is a qcqs-scheme.*

Proof. We get that $\llbracket U \rrbracket^{\text{co}}$ is local from Lemma 21 and Theorem 12, the subcanonicity of the Zariski coverage. It remains to show that $\llbracket U \rrbracket^{\text{co}}$ (merely) has an affine cover. By the Yoneda lemma, the compact open U corresponds to an element $u : \mathcal{L}_R$. Every element of \mathcal{L}_R can (merely) be expressed as a join of basic opens, i.e. we can assume $u \equiv \bigvee_i D(f_i)$ for some $f_1, \dots, f_n : R$. Since the Yoneda lemma actually gives us an isomorphism of lattices between \mathcal{L}_R and $\text{CompOpen}(\text{Sp}(R))$, we get a cover of compact opens $U \equiv \bigvee_i D(f_i)$ which is affine by Proposition 23. Note that this is an equality in the lattice $\text{CompactOpen}(X)$. But since $D(f_i) \leq U$ in $\text{CompactOpen}(X)$ for $i = 1, \dots, n$, we may regard the $D(f_i)$ as affine compact opens of $\llbracket U \rrbracket^{\text{co}}$ covering of the top element of $\text{CompactOpen}(\llbracket U \rrbracket^{\text{co}})$. ◀

7 Conclusion

In this paper we presented a formalization of qcqs-schemes as a full subcategory of the category of \mathbb{Z} -functors. We defined the Zariski coverage on $\text{CommRing}_\ell^{\text{op}}$ and proved it subcanonical. This let us define locality of \mathbb{Z} -functors and conclude that affine schemes, i.e. representable \mathbb{Z} -functors, are local. When formalizing the notion of an open covering, we introduced compact open subfunctors. We argued that compact opens can conveniently be classified by the \mathbb{Z} -functor that maps a ring to its Zariski lattice. We leveraged this fact to automatically obtain a notion of covering by compact opens and thus a formal definition of qcqs-schemes. Finally, we gave a fully constructive proof that compact opens of affine schemes are qcqs-schemes using only point-free methods.

As mentioned before, our formalization should be regarded as a univalent rather than a cubical formalization. We do not depend on cubical features of Cubical Agda such as the interval. However, we are adopting the univalent approach of distinguishing propositions, sets etc. internally and we do require the propositional truncation and the set-quotient HITs. Univalence is only used in the guise of its useful consequences like function extensionality.

7.1 Going classical

Cubical Agda's type theory is fully constructive and predicative. Using set-quotients, we can define the Zariski lattice over a ring living in the same universe as the base ring, as shown in [39]. This predicative definition is essential for defining the classifier $\mathcal{L} : \mathbb{Z}\text{Functor}_\ell$ of compact opens and thus plays a key role in our definition of functorial qcqs-schemes. This makes our approach easily extensible with the using additional logical assumptions. If one would want to formalize not only qcqs- but general schemes using the functor of points approach, this should be directly possible by using a classifier for opens, not only compact opens, instead.

Assuming impredicativity, e.g. in the form of Voevodsky's resizing axioms [37], one could define the classifier for open subfunctors as the \mathbb{Z} -functor sending a ring R to the *frame* of radical ideals of R .¹³ Alternatively, assuming classical logic, one could use the frame of Zariski-open subsets of $\text{Spec}(R)$ as the classifier. This also induces a notion of cover (not necessarily finite this time) and hence a notion of general functorial schemes. We expect that in this situation one can closely follow the approach of Section 6 to get a corresponding proof that open subfunctors of affine schemes are schemes. The only difference being perhaps the proof of Lemma 20, the fact that the classifier is separated wrt. the Zariski coverage.

¹³ Impredicativity is needed to ensure that the type of ideals of a ring R lives in the same universe as R .

We decided to stick to qcqs-schemes not only because crucial tools like the Zariski lattice were already available in the `agda/cubical` library. We hope that the paper contains valuable insights for constructive mathematicians interested in the foundations of algebraic geometry, while still being usable as a blue-print for formalizing the functor of points approach in other (possibly classical) proof assistants.

7.2 Synthetic algebraic geometry

The functor of points approach allows one to develop algebraic geometry synthetically. Here, the word *synthetic* means “working in the internal language of a suitable topos”. In our case this topos is the *big Zariski topos*, i.e. the sheaf topos of local \mathbb{Z} -functors. From the internal point of view, Zariski sheaves look like simple sets, which can make reasoning about them easier. The PhD thesis of Blechschmidt [2] contains an excellent introduction to synthetic algebraic geometry for interested readers familiar with classical algebraic geometry.

This approach can even be axiomatized. Recently, Cherubini, Coquand and Hutzler [6] have combined the axiomatic approach to synthetic algebraic geometry with HoTT/UF. By adding the axioms of synthetic algebraic geometry to a dependent type theory with univalence and HITs one can even study the cohomology of schemes synthetically. They give a model construction in a “higher” Zariski topos, where they restrict themselves to functors from *finitely presented algebras* to sets in order to avoid size issues. Finitely presented algebras over a ring R are of the form $R[x_1, \dots, x_n]/\langle p_1, \dots, p_m \rangle$. For a fixed R , the category of f.p. R -algebras is small and one can thus use it to develop functorial algebraic geometry without having to worry about universe levels. Repeating the steps outlined in this paper for f.p. algebras should give rise to a truly predicative formalization of schemes of finite presentation over R .

7.3 A constructive comparison theorem

For algebraic geometers, using the functor of points approach can sometimes be advantageous, but ultimately one wants to switch seamlessly between schemes as \mathbb{Z} -functors and schemes as locally ringed spaces. This is made possible by the so-called *comparison theorem* [12, p. 23], giving an adjunction between \mathbb{Z} -functors and locally ringed spaces, which becomes an equivalence of categories when restricted to the respective full subcategories of schemes.

Coquand, Lombardi and Schuster [8] give a point-free reconstruction of geometric qcqs-schemes that is suitable for constructive study. Instead of using locally ringed spaces, their “spectral schemes” are given as distributive lattices with a sheaf of rings. The affine scheme associated to a ring R is just the Zariski lattice \mathcal{L}_R equipped with the usual structure sheaf. Classically, these spectral schemes are equivalent to conventional qcqs-schemes because the topology of a qcqs-scheme is *coherent* or *spectral*.¹⁴

Our definition of qcqs-scheme can be seen as the functorial counterpart to the lattice-based definition of spectral scheme due to Coquand, Lombardi and Schuster. One would hope that these turn out to be equivalent by a *constructive* comparison theorem à la Demazure and Gabriel. Proving such a theorem requires a decent amount of novel constructive mathematics to be developed first. One needs to introduce a point-free notion of *locally* ringed distributive lattices that contain spectral schemes as a full subcategory and construct a suitable adjunction with \mathbb{Z} -functors. In our setting, one would only get a relative coadjunction as in Remark 5. This could be a particularly interesting problem in a univalent setting.

¹⁴Stone’s representation theorem for distributive lattices [30], tells us that all topological information of a coherent space is encoded in its lattice of compact open subsets.

References

- 1 Carlo Angiuli, Evan Cavallo, Anders Mörtberg, and Max Zeuner. Internalizing Representation Independence with Univalence. *Proc. ACM Program. Lang.*, 5(POPL), January 2021. doi: 10.1145/3434293.
- 2 Ingo Blechschmidt. Using the internal language of toposes in algebraic geometry, 2021. arXiv:2111.03685.
- 3 Anthony Bordg, Lawrence C. Paulson, and Wenda Li. Simple type theory is not too simple: Grothendieck’s schemes without dependent types. *Exp. Math.*, 31(2):364–382, 2022. doi: 10.1080/10586458.2022.2062073.
- 4 Kevin Buzzard, Chris Hughes, Kenny Lau, Amelia Livingston, Ramon Fernández Mir, and Scott Morrison. Schemes in lean. *Exp. Math.*, 31(2):355–363, 2022. doi:10.1080/10586458.2021.1983489.
- 5 Tim Cherganov. Sheaf of rings on Spec R, 2022. URL: <https://github.com/UniMath/UniMath/pull/1385>.
- 6 Felix Cherubini, Thierry Coquand, and Matthias Hutzler. A Foundation for Synthetic Algebraic Geometry, 2023. arXiv:2307.00073.
- 7 Laurent Chicli. Une formalisation des faisceaux et des schémas affines en théorie des types avec coq. In Pierre Castéran, editor, *Journées francophones des langages applicatifs (JFLA’01)*, Pontarlier, France, Janvier, 2001, number RR-4216 in Collection Didactique, pages 17–32. INRIA, June 2001. URL: <https://hal.inria.fr/inria-00072403>.
- 8 Thierry Coquand, Henri Lombardi, and Peter Schuster. Spectral Schemes as Ringed Lattices. *Annals of Mathematics and Artificial Intelligence*, 56(3):339–360, 2009. doi: 10.1007/s10472-009-9160-7.
- 9 Tom de Jong and Martín Hötzel Escardó. On Small Types in Univalent Foundations. *Logical Methods in Computer Science*, Volume 19, Issue 2, May 2023. doi:10.46298/lmcs-19(2:8)2023.
- 10 Pierre Deligne and Jean-François Boutot. Cohomologie étale: les points de départ. In *Cohomologie Etale*, pages 4–75, Berlin, Heidelberg, 1977. Springer Berlin Heidelberg.
- 11 Pierre Deligne and Jean-François Boutot. *Étale cohomology: starting points*. Translated by J. S. Milne, 2022. URL: <https://web.archive.org/web/20240511093708/https://www.jmilne.org/math/Documents/DeligneArcata.pdf>.
- 12 Michel Demazure and Peter Gabriel. *Introduction to Algebraic Geometry and Algebraic Groups*. Elsevier, 1980.
- 13 David Eisenbud and Joe Harris. *The Geometry of Schemes*, volume 197. Springer Science & Business Media, 2006.
- 14 Luis Español. Le spectre d’un anneau dans l’algèbre constructive et applications à la dimension. *Cahiers de Topologie et Géométrie Différentielle Catégoriques*, 24(2):133–144, 1983. URL: http://www.numdam.org/item/CTGDC_1983__24_2_133_0/.
- 15 Ulrich Görtz and Torsten Wedhorn. *Algebraic Geometry I: Schemes*. Springer, 2020.
- 16 Alexandre Grothendieck and Federico Gaeta. *Introduction to Functorial Algebraic Geometry: After a Summer Course*. State University of New York at Buffalo, Department of Mathematics, 1974. URL: <https://web.archive.org/web/20221013161204/https://ncatlab.org/nlab/files/GrothendieckIntrodFunctorialGeometryI1973.pdf>.
- 17 Robin Hartshorne. *Algebraic geometry*, volume 52 of *Graduate texts in mathematics*. Springer, 1977. doi:10.1007/978-1-4757-3849-0.
- 18 Martin Hofmann and Thomas Streicher. Lifting Grothendieck Universes. <https://web.archive.org/web/20240405155732/https://www2.mathematik.tu-darmstadt.de/~streicher/NOTES/lift.pdf>, 1997. Unpublished note.
- 19 Peter T Johnstone. *Sketches of an Elephant: A Topos Theory Compendium: Volume 2*, volume 2. Oxford University Press, 2002.
- 20 André Joyal. Les théorèmes de Chevalley-Tarski et remarques sur l’algèbre constructive. *Cahiers Topologie Géom. Différentielle*, 16:256–258, 1976.

- 21 Nicolai Kraus. The General Universal Property of the Propositional Truncation. In Hugo Herbelin, Pierre Letouzey, and Matthieu Sozeau, editors, *20th International Conference on Types for Proofs and Programs (TYPES 2014)*, volume 39 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 111–145, Dagstuhl, Germany, 2015. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.TYPES.2014.111.
- 22 F. William Lawvere. Categories: Grothendieck’s 1973 Buffalo Colloquium, 2003. Category mailing list, March 30. URL: <https://web.archive.org/web/20240418015152/https://www.mta.ca/~cat-dist/archive/2003/03-3>.
- 23 Saunders Mac Lane and Ieke Moerdijk. *Sheaves in Geometry and Logic: A First Introduction to Topos Theory*. Springer Science & Business Media, 2012.
- 24 David Madore. Comment définir efficacement ce qu’est un schéma, 2013. David Madore’s WebLog. URL: <https://web.archive.org/web/20240418040920/http://www.madore.org/~david/weblog/d.2013-09-21.2160.definition-schema.html>.
- 25 Per Martin-Löf. An Intuitionistic Theory of Types: Predicative Part. In H. E. Rose and J. C. Shepherdson, editors, *Logic Colloquium ’73*, volume 80 of *Studies in Logic and the Foundations of Mathematics*, pages 73–118. North-Holland, 1975. doi:10.1016/S0049-237X(08)71945-1.
- 26 Marc Nieper-Wißkirchen. Algebraische Geometrie. unpublished lecture notes, 2008.
- 27 nLab authors. coverage. <https://ncatlab.org/nlab/show/coverage>, May 2024. Revision 42.
- 28 nLab authors. matching family. <https://ncatlab.org/nlab/show/matching+family>, May 2024. Revision 7.
- 29 nLab authors. relative adjoint functor. <https://ncatlab.org/nlab/show/relative+adjoint+functor>, May 2024. Revision 17.
- 30 Marshall Harvey Stone. Topological Representations of Distributive Lattices and Brouwerian Logics. *Časopis pro pěstování matematiky a fyziky*, 67(1):1–25, 1938.
- 31 Thomas Streicher. *Investigations Into Intensional Type Theory*. Habilitation thesis, Ludwig-Maximilians-Universität München, 1993. URL: <https://web.archive.org/web/20240324013847/https://www2.mathematik.tu-darmstadt.de/~streicher/HabilStreicher.pdf>.
- 32 The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. <https://homotopytypetheory.org/book>, Institute for Advanced Study, 2013.
- 33 Ravi Vakil. *The Rising Sea: Foundations of Algebraic Geometry*. Preprint, 2024. URL: <https://web.archive.org/web/20240507025011/http://math.stanford.edu/~vakil/216blog/FOAGfeb2124public.pdf>.
- 34 Andrea Vezzosi, Anders Mörtberg, and Andreas Abel. Cubical Agda: A Dependently Typed Programming Language with Univalence and Higher Inductive Types. *Journal of Functional Programming*, 31:e8, 2021. doi:10.1017/S0956796821000034.
- 35 V. Voevodsky, B. Ahrens, D. Grayson, et al. UNIMATH: Univalent Mathematics. Available at <https://github.com/UniMath>.
- 36 Vladimir Voevodsky. Univalent Foundations, September 2010. Notes from a talk in Bonn. URL: https://web.archive.org/web/20240418032908/https://www.math.ias.edu/vladimir/sites/math.ias.edu.vladimir/files/Bonn_talk.pdf.
- 37 Vladimir Voevodsky. Resizing Rules – their use and semantic justification, 2011. Slides from a talk at TYPES, Bergen, 11 September. URL: https://web.archive.org/web/20240405160805/https://www.math.ias.edu/vladimir/sites/math.ias.edu.vladimir/files/2011_Bergen.pdf.
- 38 Vladimir Voevodsky. An experimental library of formalized mathematics based on the univalent foundations. *Mathematical Structures in Computer Science*, 25(5):1278–1294, 2015. doi:10.1017/S0960129514000577.
- 39 Max Zeuner and Anders Mörtberg. A Univalent Formalization of Constructive Affine Schemes. In Delia Kesner and Pierre-Marie Pédrot, editors, *28th International Conference on Types for Proofs and Programs (TYPES 2022)*, volume 269 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 14:1–14:24, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.TYPES.2022.14.



Robust Mean Estimation by All Means

Reynald Affeldt  



National Institute of Advanced Industrial Science and Technology (AIST), Tokyo, Japan

Clark Barrett  

Stanford University, CA, USA

Alessandro Bruni  

IT University of Copenhagen, Denmark

Ieva Daukantas  

IT University of Copenhagen, Denmark

Harun Khan  

Stanford University, CA, USA

Takafumi Saikawa  

Nagoya University, Japan

Carsten Schürmann  

IT University of Copenhagen, Denmark

Abstract

We report the results of a verification experiment on an algorithm for robust mean estimation, i.e., an algorithm that computes a mean in the presence of outliers. We formalize the algorithm in the Coq proof assistant and devise a pragmatic approach for identifying and solving issues related to the choice of bounds. To keep our formalization succinct and generic, we recast the original argument using an existing library for finite probabilities that we extend with reusable lemmas. To formalize the original algorithm, which relies on a subtle convergence argument, we observe that by adding suitable termination checks, we can turn it into a well-founded recursion without losing its original properties. We also exploit a tactic for solving real-valued inequalities by approximation to heuristically fix inaccurate constant values in the original proof.

2012 ACM Subject Classification Mathematics of computing → Probabilistic inference problems; Theory of computation → Logic and verification

Keywords and phrases robust statistics, probability theory, formal verification

Digital Object Identifier 10.4230/LIPIcs.ITP.2024.39

Category Short Paper

Supplementary Material

Software (Source code): <https://github.com/affeldt-aist/infotheo/> [1]
archived at [swh:1:dir:0fc3cb212f8a5ba2547161db17ea6b87317bad39](https://swh.1:dir:0fc3cb212f8a5ba2547161db17ea6b87317bad39)

Funding The first and sixth authors acknowledge support of the JSPS KAKENHI Grant Number 22H00520. The second and fifth authors were supported in part by the Stanford Center for Automated Reasoning.

Acknowledgements The authors would like to thank the anonymous referees for their thorough reviews.

1 Towards formally-verified robust mean estimators

Our motivation is to produce formally-verified programs that perform robust mean estimation as a first step towards formal robust statistics. The setting for robust mean estimation is as follows. We are given samples from an unknown distribution, but some fraction of them



© Reynald Affeldt, Clark Barrett, Alessandro Bruni, Ieva Daukantas, Harun Khan, Takafumi Saikawa, and Carsten Schürmann;

licensed under Creative Commons License CC-BY 4.0

15th International Conference on Interactive Theorem Proving (ITP 2024).

Editors: Yves Bertot, Temur Kutsia, and Michael Norrish; Article No. 39; pp. 39:1–39:8



Leibniz International Proceedings in Informatics

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

are *outliers* (data points that differ significantly from other observations) that we want to discard. A robust mean estimator is an algorithm that computes the mean of a set of data points while minimizing the effect of outliers. More formally, we say that a mean estimator is *robust* when the difference between the computed mean and the optimal mean, without considering outliers, can be upper-bounded by arbitrarily small positive numbers. There are several algorithms for robust mean estimation, e.g., the median is a robust estimator, and the trimmed mean by Tukey [14] is a robust estimator. Both of these estimators work by excluding samples. The robustness of trimmed mean has already been formalized in COQ [5].

In this paper we verify the robustness of an archetypal algorithm for mean estimation that operates by iteratively re-weighting the influence of each sample. The algorithm is an example of an M-estimator [7, Ch. 3], and it is described in Steinhardt’s Ph.D. thesis [11] under the name `filter1d`. M-estimators assign weights to each sample instead of excluding samples. Not all M-estimators are robust: robustness requires that sufficiently low weights are assigned to outliers.

We identify and resolve two main challenges with the algorithm `filter1d`. First, through formalization, we identify and fix issues with the original proofs which contain erroneous constants and bounds. The second challenge is technical. The original proofs spelled out calculations in terms of “big sums,” which are the intuitive *lingua franca* for probability theory in finite settings. We use and extend existing libraries to keep the formalization succinct and reusable, and as a consequence, proofs became more modular.

These challenges, checking and occasionally fixing constant bounds, and devising techniques for symbolic calculations, are recurring issues when formalizing paper proofs about statistical algorithms. In this paper, we formalize the `filter1d` algorithm and its related theory in COQ [12]. We leverage this experience to inform a broader discussion of these recurring issues and to document a general approach. We represent calculations involving probabilities using an existing library, and we also take advantage of an automated tactic in COQ (namely `interval` [9]) to fix erroneous constants in the original proof. In the end, we are able to formalize `filter1d` without ambiguity in the language of the COQ proof assistant.

The paper is organized as follows. First, we present the `filter1d` algorithm for robust mean estimation in Sect. 2. In Sect. 3, we explain how we formalize the bound on the mean using an existing library and using changes of distributional assumptions. In Sect. 4, we explain how we formalize the bound on the variance by fixing the original proofs, in practice by using the `interval` tactic of COQ. Finally, we formalize the algorithm for robust mean estimation and prove its termination and correctness in Sect. 5. The results presented in this paper are available online as a COQ formalization [1].

2 An archetypal algorithm for robust mean estimation

The algorithm takes as input a discrete random variable X (with sampled values x_1, \dots, x_n), a finite probability distribution P (with probabilities p_1, \dots, p_n), and a positive value v . It either computes a mean $\hat{\mu}$ or fails. The computed mean is expected to be close to the mean of the random variable X for the distribution P with the outliers removed. It is correctly computed in the following sense: if there exists a subset S of $\text{dom}(X)$ such that $v = \sigma_S^2$ and the probability ε of \bar{S} (i.e., the set of outliers) is smaller than $1/12.7$, then $|\hat{\mu} - \mu_S|$ is bounded by $\sqrt{\frac{v \cdot 2\varepsilon}{2 - \varepsilon}} + \sqrt{\frac{16v \cdot 2\varepsilon}{1 - \varepsilon}}$ (where $\sigma_S^2 \stackrel{\text{def}}{=} \mathbb{V}[X|S]$ and $\mu_S \stackrel{\text{def}}{=} \mathbb{E}[X|S]$). This signifies what is meant by robustness: as long as the probability of the set of outliers is smaller than a constant, the result is guaranteed to stay within a reasonable range.

The algorithm maintains a sequence of weights c_i , initialized to 1, which represent the contribution of each point to the computation. These weights are then updated iteratively, such that points that deviate the most are given less weight (Algorithm 1).

■ **Algorithm 1** `filter1d`.

1. Initialize each weight c_1, \dots, c_n to 1
 2. Compute the *empirical mean*: $\hat{\mu}_c \leftarrow (\sum_{i=1}^n p_i x_i c_i) / (\sum_{i=1}^n p_i c_i)$
 3. Compute the *empirical variance*: $\hat{\sigma}_c^2 \leftarrow (\sum_{i=1}^n p_i c_i \tau_i) / (\sum_{i=1}^n p_i c_i)$ where $\tau_i \stackrel{\text{def}}{=} (x_i - \hat{\mu}_c)^2$
 4. If $\hat{\sigma}_c^2 \leq 16v$ then terminate and output $\hat{\mu}_c$
 5. Otherwise, update $c_i \leftarrow c_i(1 - \tau_i/\tau_{max})$ where $\tau_{max} = \max_{i \in \text{supp}(c)} \tau_i$
 6. If all $c_i = 0$ then terminate with error; otherwise, go back to line 2
-

Note that at Step 5, we take the maximum over the support of the function $c : i \mapsto c_i$ whereas the original algorithm [11, Sect. 1.2.3] does not make this explicit. This makes our algorithm slightly different from the original algorithm, but it is a reasonable modification because it does not change the computed values. In Step 6, our algorithm also checks that not all c_i are zero before continuing, because otherwise there is a division by zero in Step 2 (take for example the situation of a positive computed variance and two points of equal weight). These modifications do not change the property originally stated by Steinhardt, namely that when `filter1d` terminates, it results in the desired mean (Step 4). Furthermore, we generalize the algorithm by giving each point x_i a probability p_i instead of assuming a uniform distribution¹ as in [11].

Lastly, the robustness of `filter1d` is a consequence of the following invariant [11, eqn (\mathcal{I}), page 5] being preserved: $\sum_{i \in S} (1 - c_i) p_i \leq \frac{1-\epsilon}{2} \sum_{i \in \bar{S}} (1 - c_i) p_i$ (\mathcal{I}). It shows that the amount of “mass” (the sum of the weights) removed from the points in S is less than $\frac{1-\epsilon}{2}$ times the amount of mass removed from the outliers. The invariant is key to establishing the bound between the empirical mean and the mean on the points in S shown in Sect. 3.3, which delivers the final robustness argument, and the bounding of the empirical variance of Sect. 4, which in turn is necessary to show that the invariant is preserved when the weights are updated.

3 Bounding the empirical mean using resilience and changes of distributional assumptions

In this section, we rework the original proofs so that they can be formalized using INFO_{THEO} [3, 4], a formalization of information theory in COQ.

3.1 Background about formalization of probabilities

INFO_{THEO} introduces a number of definitions and lemmas to deal with finite probabilities. The type of finite distributions is `{fdist U}` where U is a finite type (`finType` as provided by the MATHCOMP library [8]). A finite set in MATHCOMP is an object of type `{set U}` where

¹ This change does not affect the final computation: for finite samples, p_i corresponds to a multiplicity count of the occurrences of x_i . In the case of a uniform distribution, all p_i have the value $1/n$, so the original algorithm is a special case. Note that the computation of the updated c_i at Step 5 does not directly depend on the value of p_i .

39:4 Robust Mean Estimation by All Means

U is a finite type; finite sets are used to represent events. A probability space is a finite type with a function P , which assigns to each point of the type a probability in $[0, 1]$. The domain of P can also be extended to finite sets, taking the sum of the pointwise values, resulting in the corresponding probability measure. A random variable is a function embodying the notion of probabilistically changing values: it goes from a probability space to a type, the input being assumed to be sampled from the probability space according to its probability measure. The type of real-valued random variables is denoted by $\{\text{RV } P \rightarrow \mathbb{R}\}$ where P is the probability measure of the ambient probability space and \mathbb{R} is the type of real numbers [3, Sect. 2]. The conditional expectation of the random variable X w.r.t. an event A is $\mathbb{E}[X \mid A]$ (see [2] for conditional probabilities in `INFOTHEO`). We use a *resilience* lemma that bounds the distance between two conditional expectations (generalizing [5, Sect. 5.1]):

► **Lemma 1 (Resilience).** *Let X be a random variable with probability measure P , and F, G be two events such that $F \subseteq G$. Then for any δ such that $0 < \delta \leq P(F)/P(G)$, we have*

$$|\mathbb{E}[X|F] - \mathbb{E}[X|G]| \leq \sqrt{2\text{V}[X|G] \frac{1-\delta}{\delta}}.$$

3.2 Changing distributional assumptions

Steinhardt argues for the correctness of his algorithm using big sums. We recast big sums in terms of expectation and variance, as provided by `INFOTHEO`, to enable the reuse of existing lemmas. We further extend `INFOTHEO` with lemmas about changes in distributional assumptions of RVs.

A *change of distributional assumption* is the transformation of a RV X on a probability space $T1$ with probability measure P into another RV on another space $T2$ with probability measure Q by precomposing a function $f : T1 \rightarrow T2$:

Definition `change_dist (T1 T2 : finType) (P : {fdist T1}) (Q : {fdist T2}) (f : T2 → T1) (X : {RV P → ℝ}) : {RV Q → ℝ} := X ∘ f.`

We denote with $Q.\text{-RV } X \circ f$ the resulting RV and write $Q.\text{-RV } X$ when f is the identity.

The main application of changing distributional assumptions is to formalize the empirical variance (Step 3 of `filter1d`). Given a probability measure P and (non-negative) weights c_i ($i \in A$), we call the probability measure $i \mapsto c_i p_i / \sum_{j \in A} c_j p_j$ *weighted*. In `COQ` we provide the weighted probability measure of P as a function `wgt` whose argument is a proof of $\sum_{j \in A} c_j p_j \neq 0$. We call *weighted* the change of distributional assumption with a weighted probability measure. In particular, the *empirical mean* of X with weights c_i can be expressed as the expectation of a weighted RV.

Similarly, given a probability measure P over U and a function h with codomain $\subseteq [0, 1]$, we call *split* the probability measure over $U \times \{F, T\}$ (`Split.d P h` in `COQ`):

$$\text{split}(P, h)_{(i,b)} \stackrel{\text{def}}{=} \begin{cases} h(i)p_i & \text{if } b = T \\ (1 - h(i))p_i & \text{if } b = F. \end{cases}$$

We call *first-split* the RV $Q.\text{-RV } X \circ \text{fst}$ (where Q is `Split.d P h`). It is possible to change a conditional expectation by a first-split. (The `COQ` notation `* is for the Cartesian product and `[set: bool]` is for the set of booleans.)

Definition `fst_RV (X : {RV P → ℝ}) : {RV d → ℝ} := (Split.d P h).RV X ∘ fst.`

Lemma `cEx (X : {RV P → ℝ}) A : `E[X | A] = `E[fst_RV X | A `* [set: bool]].`

3.3 Bounding the empirical mean

We prove the following bound between the mean and the empirical mean under the invariant (\mathcal{I}) : $|\mu_S - \hat{\mu}_c| \leq \sigma_S \sqrt{\frac{2\varepsilon}{2-\varepsilon}} + \hat{\sigma}_c \sqrt{\frac{2\varepsilon}{1-\varepsilon}}$ [11, Lemma 1.4].

For that purpose, we introduce an intermediate value $\tilde{\mu}_c$ and first show $(\mu_S - \tilde{\mu}_c)^2 \leq \hat{\sigma}_c^2 \frac{2\varepsilon}{1-\varepsilon}$, which is formalized as follows, using PCO, a proof that `Weighted.total P C != 0`:

`Let WP := Weighted.d PCO.`

`Lemma bound_emean : invariantW -> (* A weaker version of the invariant *)
 (E_[WP.-RV X | S] - E_(WP.-RV X))^+2 <= V_(WP.-RV X) * 2 * eps / (1 - eps).`

The proof is a direct application of Lemma 1 with $\delta = 1 - \varepsilon$ and G being the full set.

The second bound is $(\mu_S - \tilde{\mu}_c)^2 \leq \sigma_S^2 \frac{2\varepsilon}{2-\varepsilon}$, proved using the inequality $1 - \varepsilon/2 \leq \frac{\sum_{i \in S} c_i p_i}{P(S)}$ (which we shall call ‘‘S-mass’’):

`Lemma bound_mean : invariant ->`

`(E_[X | S] - E_[WP.-RV X | S])^+2 <= V_[X | S] * 2 * eps / (2 - eps).`

The proof relies on the observation that one can change the distributional assumption of μ_S as $\mathbb{E}[X_1|S \times \{F, T\}]$ and similarly $\tilde{\mu}_c = \mathbb{E}[X_1|S \times \{T\}]$, where X_1 is the first-split of X . This changing of distributional assumption corresponds to the formalization of the following proof step in [11, page 63]: ‘‘(here we think of $\tilde{\mu}_c$ as the mean of an event occurring with probability $\sum_{i \in S} c_i/|S|$ under the uniform distribution on $|S|$)’’. Using changing of distributional assumption, the proof is actually an application of Lemma 1, using ‘‘S-mass’’ to fulfill its hypothesis:

$$\underbrace{(\mathbb{E}[X_1|S \times \{F, T\}] - \mathbb{E}[X_1|S \times \{T\}])^2}_{\mu} \leq \underbrace{2 \mathbb{V}[X_1|S \times \{F, T\}]}_{\mathbb{V}[X|S]} \frac{1 - (1 - \varepsilon/2)}{1 - \varepsilon/2}.$$

Lemma 1

4 Bounding of variance: using interval to fix proofs

In this section, we explain how we formalize and fix the proofs that bound the empirical variance of `filter1d`. Precisely, we obtain the following bound for the empirical variance.

► **Lemma 2.** *Provided the invariant (\mathcal{I}) , $16\sigma_S^2 \leq \hat{\sigma}_c^2$, and $\varepsilon \leq 1/12.7$, we have:*

(a) $\sum_{i \in S} c_i p_i \tau_i \leq \frac{1-\varepsilon}{3.35} \sigma_c^2$ and (b) $\sum_{i \in \bar{S}} c_i p_i \tau_i \geq \frac{2}{3.35} \sigma_c^2$.

Steinhardt claims an upper-bound of $1/12$ for ε and uses 3 in the denominators in (a) and (b) instead of 3.35 [11, Lemma 1.4 (part 2)/eqn A.6–A.9]. We believe that the lemma cannot be proved with Steinhardt’s bounds because we corrected mistakes in the original proof of (b) [11, eqn A.10–A.11, page 63], which we discovered when we mechanized an argument that ‘‘consist[s] of straightforward but tedious calculation’’ [11, page 5].

The correct bounds can be obtained by using the `interval` tactic [9, 10] of Coq. We parameterize the Coq statements corresponding to Lemma 2 with a variable `eps_max` for the upper-bound to be found and with a variable `denom` for the denominators in (a) and (b):

`Notation eps := Pr P cplt_S. (* cplt_S is the complement of S *)`

`Notation eps_max := 10 / 127. (* values found by try-and-error, see below *)`

`Notation denom := 335 / 100.`

`Hypothesis low_eps : eps <= eps_max.`

`Lemma bound_empirical_variance_S :`

`\sum_(i in S) C i * P i * tau i <= (1 - eps)/denom * V_(WP.-RV X).`

`Lemma bound_empirical_variance_cplt_S :`

`2/denom * V_(WP.-RV X) <= \sum_(i in cplt_S) C i * P i * tau i.`

39:6 Robust Mean Estimation by All Means

Then, we use two proof scripts (inspired by Steinhard’s proofs) to reduce the proof to purely arithmetical subgoals depending on the variables `eps_max` and `denom`. Finally, we use COQ to help find optimal values for `eps_max` and `denom` by adjusting the parameters and replaying the proof scripts, deferring subgoals related to arithmetic to `interval`. In this way, we obtained the values 3.35 and 1/12.7 for `denom` and `eps_max`, respectively, by iterative trial-and-error. Note that with Steinhard’s parameters (3 and 1/12), the proof script `bound_empirical_variance_S` holds, but `bound_empirical_variance_cplt_S` does not. It is possible to use the same process to further refine the constants: e.g., we are able to show the same results for 3.345 and 1/12.65.

5 A formally robust algorithm for mean estimation

5.1 Formalizing `filter1d`

The algorithm `filter1d` was presented in Sect. 2 in the form of a loop. To formalize it in COQ, we turn it into a recursive algorithm by using the `Function` command [12, Functional Induction], which can be used for arbitrary well-founded recursion (not just structural).

```
1 Variables (U : finType) (P : {fdist U}) (X : {RV P -> R}).
2 Function filter1D_rec v (v_ge0 : 0 <= v)
3   (C : nneg_finfun U) (C01 : is01 C) (PC0 : Weighted.total P C != 0)
4   {measure (fun C => #| 0.-support C |) C} :=
5   let WP := wgt PC0 in
6   if `V (WP.-RV X) <=? 16 * v is left _ then
7     Some (`E (WP.-RV X))
8   else
9     let C' := update X PC0 in
10    if Weighted.total P C' !=? 0 is left PC0' then
11      filter1D_rec v_ge0 (is01_update X PC0 C01) PC0'
12    else
13      None.
```

The parameter `U` is a finite type, `P` is a probability measure over `U`, and `X` is a random variable whose probability measure is `P`. The function takes as parameters the variance `v` with a proof that it is non-negative (`v_ge0`), as well as a (non-negative) weight function `C` with proofs that the weights are less than 1 (`C01`) and that their total is not 0 (`PC0`). The measure that controls termination is the size of the support of `C`. Indeed, an execution of Step 5 sets one nonzero weight to zero (the weight C_i such that $\tau_i = \tau_{max}$), rendering the nonzero support of C_i ’s (`0.-support C` at line 4) strictly smaller.

At each iteration, we update the distributional assumption (line 5) and compute the variance (line 6). Then, we update the weights (line 9) and test if we can recurse by checking that the total of the new weights is nonzero. A termination is reached if (1) the empirical variance satisfies the convergence condition $\hat{\sigma}_c^2 \leq 16v$ (line 6), resulting in `Some($\hat{\mu}_c$)`, or (2) the weighted total after the update is zero, resulting in `None` (line 13). When neither is the case, we perform a recursive call at line 11, passing two proof terms that also implicitly carry the updated C'_i ’s. The function `is01_update` computes a term that shows that C'_i ’s do not break the invariant that they are between 0 and 1. The proof `PC0'` directly results from the case analysis at line 10.

We feed a set of constant weights, all equal to 1, as the base case of C_i to complete the definition of `filter1d` (again, in this definition, the constant weights are implicitly passed by two proof terms `C1_is01 U` and `PC1_neq0 P`):

Definition `filter1D v (v_ge0 : 0 <= v) := filter1D_rec v_ge0 (C1_is01 U) (PC1_neq0 P)`.

5.2 Robustness of `filter1d`

We can finally prove the robustness of `filter1d`. The recursive definition based on the measure `#|0.-support C|` ensures that the function returns `None` if and only if the computation has failed and all C_i 's are set to zero. Otherwise, it returns the empirical mean upon termination.

The first step in proving robustness is to show the preservation of the invariant (\mathcal{I}) w.r.t. the update performed at Step 5:

```
Lemma invariant_update : let C' := update X PC0 in
  invariant P C S eps -> invariant P C' S eps.
```

This formalizes [11, Lemma 1.5, page 5]. The proof is a consequence of Lemma 2 (Sect. 4) used in conjunction with the following property of `update` (`update_removed_weight` in [1]): $\sum_{i \in E} p_i(1 - c'_i) = \sum_{i \in E} p_i(1 - c_i) + 1/\tau_{max} \sum_{i \in E} p_i c_i \tau_i$ where the c'_i 's are the weights after an update, i.e., $c'_i := c_i \left(1 - \frac{\tau_i}{\tau_{max}}\right)$.

Using the preservation of the invariant, we show that `filter1d` is robust with the following theorem and its corollary:

```
1 Hypothesis low_eps : eps <= eps_max.
2 Lemma filter1D_correct :
3   let v := `V_[X | S] in
4   if @filter1D U P X v v_ge0 is Some mu_hat
5   then `| `E_[X | S] - mu_hat | <= Num.sqrt (v * (2 * eps) / (2 - eps)) +
6     Num.sqrt (16 * v * (2 * eps) / (1 - eps))
7   else false.
8
9 Corollary filter1D_converges : @filter1D U P X `V_[X | S] v_ge0 != None.
```

The `@` mark (in lines 4 and 9) disables the inference of implicit arguments in COQ. The theorem shows that, if given the appropriate ε and variance: (i) the algorithm never terminates with an error; and (ii) when the algorithm terminates, the empirical mean is close enough to the true mean.

6 Conclusions

This paper describes a mechanized proof of a simple M-estimator in the proof assistant COQ. The result contributes to the state of the art by improving the pencil-and-paper presentation, by making explicit all details, by clarifying the termination argument, by fixing errors in the pencil-and-paper proofs, and by improving the error bounds. Our approach makes use of the `interval` tactic, which helped determine the correct bounds in the formulation of the main lemma (Sect. 4). The formal verification of the M-estimator can be seen as yet another result in a series of applications of formalized probability, among them the verification of stochastic algorithms [13] and machine learning algorithms [15].

In future work, we plan to generalize the present work on robust mean estimation to the multi-dimensional case (as in [6]) and to normed vector spaces.

References

- 1 Reynald Affeldt, Alessandro Bruni, Ieva Daukantas, and Takafumi Saikawa. Robust mean estimators. Available as part of the InfoTheo library [4], directory `robust`, 2024.
- 2 Reynald Affeldt, Jacques Garrigue, and Takafumi Saikawa. Reasoning with conditional probabilities and joint distributions in Coq. *Computer Software*, 37(3):79–95, 2020. doi: 10.11309/jssst.37.3_79.

- 3 Reynald Affeldt, Manabu Hagiwara, and Jonas Sénizergues. Formalization of Shannon’s theorems. *J. Autom. Reason.*, 53(1):63–103, 2014. doi:10.1007/S10817-013-9298-1.
- 4 Reynald Affeldt, Manabu Hagiwara, Jonas Sénizergues, Jacques Garrigue, Kazuhiko Sakaguchi, Taku Asai, Takafumi Saikawa, Naruomi Obata, and Alessandro Bruni. InfoTheo: A Coq formalization of information theory and linear error-correcting codes. <https://github.com/affeldt-aist/infotheo>, 2018. Last stable release: 0.7.2 (2024).
- 5 Ieva Daukantas, Alessandro Bruni, and Carsten Schürmann. Trimming data sets: a verified algorithm for robust mean estimation. In *23rd International Symposium on Principles and Practice of Declarative Programming (PPDP 2021), Tallinn, Estonia, September 6–8, 2021*, pages 17:1–17:9. ACM, 2021. doi:10.1145/3479394.3479412.
- 6 Ilias Diakonikolas, Gautam Kamath, Daniel Kane, Jerry Li, Ankur Moitra, and Alistair Stewart. Robust estimators in high-dimensions without the computational intractability. *SIAM J. Comput.*, 48(2):742–864, 2019. doi:10.1137/17M1126680.
- 7 Peter J. Huber. *Robust Estimation of a Location Parameter*, pages 492–518. Springer New York, New York, NY, 1992. doi:10.1007/978-1-4612-4380-9_35.
- 8 MathComp. The mathematical components library. Available at <https://github.com/math-comp/math-comp>, 2007. Last stable version: Version 2.2.0 (2024).
- 9 Guillaume Melquiond. <https://coqinterval.gitlabpages.inria.fr/>, 2008. Last stable version: 4.12.0 (2024).
- 10 Guillaume Melquiond. Proving bounds on real-valued functions with computations. In *4th International Joint Conference on Automated Reasoning (IJCAR 2008), Sydney, Australia, August 12–15, 2008*, volume 5195 of *Lecture Notes in Computer Science*, pages 2–17. Springer, 2008. doi:10.1007/978-3-540-71070-7_2.
- 11 Jacob Steinhardt. *Robust Learning: Information Theory and Algorithms*. PhD thesis, Stanford, 2018.
- 12 The Coq Development Team. *The Coq Proof Assistant Reference Manual*. Inria, 2024. Available at <https://coq.inria.fr>. Version 8.19.0.
- 13 Jean-Baptiste Tristan, Joseph Tassarotti, Koundinya Vajjha, Michael L. Wick, and Anindya Banerjee. Verification of ML systems via reparameterization, 2020. arXiv:2007.06776.
- 14 J.W. Tukey and Princeton University. Department of Statistics. *A Survey of Sampling from Contaminated Distributions*. STRG Technical report. Princeton University, 1959.
- 15 Koundinya Vajjha, Barry M. Trager, Avraham Shinnar, and Vasily Pestun. Formalization of a stochastic approximation theorem. In *13th International Conference on Interactive Theorem Proving (ITP 2022), August 7–10, 2022, Haifa, Israel*, volume 237 of *LIPICs*, pages 31:1–31:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPICs.ITP.2022.31.

Formalising Half of a Graduate Textbook on Number Theory

Manuel Eberl   

University of Innsbruck, Austria

Anthony Bordg   

Université Paris-Saclay, INRIA, CNRS, ENS Paris-Saclay, Laboratoire Méthodes Formelles, France

Lawrence C. Paulson   

University of Cambridge, UK

Wenda Li   

University of Edinburgh, UK

Abstract

Apostol's *Modular Functions and Dirichlet Series in Number Theory* [2] is a graduate text covering topics such as elliptic functions, modular functions, approximation theorems and general Dirichlet series. It relies on complex analysis, winding numbers, the Riemann ζ function and Laurent series. We have formalised several chapters and can comment on the sort of gaps found in pedagogical mathematics. Proofs are available from https://github.com/Wenda302/Number_Theory_ITP2024.

2012 ACM Subject Classification Theory of computation \rightarrow Logic and verification

Keywords and phrases Isabelle/HOL, number theory, complex analysis, formalisation of mathematics

Digital Object Identifier 10.4230/LIPIcs.ITP.2024.40

Category Short Paper

Supplementary Material

Software (Isabelle/HOL files): <https://doi.org/10.5281/zenodo.12586104> [5]

Funding ERC Advanced Grant ALEXANDRIA (Project 742178).

Acknowledgements We would like to thank Sander Dahmen and Kevin Buzzard for providing various advice concerning number theory and the reviewers for their suggestions.

1 Introduction

Number theory is an ideal testbed for techniques in the formalisation of mathematics: it is central to mathematics, as many Fields medals attest, and its analytic branch requires the deployment of complex analysis and approximation theory.

Apostol's popular textbook series is a good choice of source material. His *Modular Functions and Dirichlet Series* [2] follows on from his *Introduction to Analytic Number Theory* [1], most of which has already been formalised in Isabelle/HOL [4]. By formalising both volumes we create a good basis for formalising further work in analytic number theory, while at the same time investigating Apostol's actual text forensically.

Isabelle/HOL [7] is a popular proof assistant. Based on simple type theory, its advantages include best-in-class automation, a library of over four million lines of formal proofs, and a structured proof language offering a good degree of legibility. Users work within a highly sophisticated interactive development environment.

We report on ongoing work to formalise the book and build a foundation of modular forms in Isabelle/HOL. We explore the chapters that we formalised fully (1, 2, 3, 7) and the parts of Chapter 6 that were already completed, commenting on what was covered and



© Manuel Eberl, Anthony Bordg, Lawrence C. Paulson, and Wenda Li;
licensed under Creative Commons License CC-BY 4.0

15th International Conference on Interactive Theorem Proving (ITP 2024).

Editors: Yves Bertot, Temur Kutsia, and Michael Norrish; Article No. 40; pp. 40:1–40:7

Leibniz International Proceedings in Informatics



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

where we had issues with the text. Except for one technical lemma that we did not need, all theorems from these chapters have been formalised. In particular, all results mentioned in this paper have been formalised.

2 Prerequisites: holomorphicity, analyticity, meromorphicity

A complex function is called *holomorphic* (or *analytic*) on an open set $A \subseteq \mathbb{C}$ if its derivative exists at every point of A . In the Isabelle library, these notions are defined not only for open sets and here they do not coincide: `f holomorphic_on A` means that f is differentiable at every point of A . On the other hand, `f analytic_on A` means that f has a power series expansion at every point of A – or, equivalently, that f is holomorphic on some open superset of A . For non-open sets, the notion `analytic_on` turns out to be much more useful.

A weaker condition than holomorphicity is *meromorphicity* on a set A : the function is differentiable at every point of A except for some isolated points at which it has *poles* (i.e. it tends to infinity). It was not straightforward to extend this definition to non-open sets, and after some false starts we arrived at the following very simple definition: f is meromorphic on A if f has a Laurent series expansion at every point of A .

definition `meromorphic_on` :: "(complex \Rightarrow complex) \Rightarrow complex set \Rightarrow bool"
 (infixl "(meromorphic'_on)" 50) **where**
 "f meromorphic_on A \longleftrightarrow ($\forall z \in A. \exists F. (\lambda w. f (z + w))$ has_laurent_expansion F)"

3 Elliptic functions and complex lattices

Two complex numbers ω_1 and ω_2 such that $\omega_2/\omega_1 \notin \mathbb{R}$ generate a lattice $\Lambda = \mathbb{Z}\omega_1 + \mathbb{Z}\omega_2$ in the complex plane. If we identify all complex numbers that differ by an element of Λ we obtain a complex torus \mathbb{T}_Λ .

► **Definition 1** (elliptic functions). An elliptic function is a meromorphic function $\mathbb{T}_\Lambda \rightarrow \mathbb{C}$.

Apostol defines it as a meromorphic function $\mathbb{C} \rightarrow \mathbb{C}$ that is periodic in both ω_1 and ω_2 . This is also how we formalise it. The simplest non-trivial elliptic function is the following:

► **Definition 2** (Weierstraß \wp function). $\wp(\Lambda, z) = \frac{1}{z^2} + \sum_{\omega \in \Lambda \setminus \{0\}} \left(\frac{1}{(z-\omega)^2} - \frac{1}{\omega^2} \right)$.

From this, a collection of related numbers arises:

Eisenstein series: $G_n(\Lambda) = \sum_{\omega \in \Lambda \setminus \{0\}} \omega^{-n}$.

Weierstraß invariants: $g_2(\Lambda) = 60G_4(\Lambda)$ and $g_3(\Lambda) = 140G_6(\Lambda)$.

Modular discriminant: $\Delta(\Lambda) = g_2(\Lambda)^3 - 27g_3(\Lambda)^2$.

Klein's J invariant: $J(\Lambda) = g_2(\Lambda)^3/\Delta(\Lambda)$.

To illustrate the relevance of these numbers, note the following results:

► **Theorem 3** (Laurent series expansion of \wp at $z = 0$). $\wp(\Lambda, z) = \frac{1}{z^2} + \sum_{n \geq 1} (n+1)G_{n+2}(\Lambda)z^n$.

► **Theorem 4** (Differential equation for \wp). $[\wp'(\Lambda, z)]^2 = 4\wp^3(\Lambda, z) - g_2(\Lambda)\wp(\Lambda, z) - g_3(\Lambda)$.

► **Theorem 5** (Non-vanishing of Δ). $\Delta(z) \neq 0$ for all z .

It is convenient to rotate and scale the lattice such that $\omega_1 = 1$ and $\omega_2 = \tau$ (where $\text{Im}(\tau) > 0$) so that we can describe the lattice by a single complex parameter. We can thus also write $G_n(\tau)$, $\Delta(\tau)$ etc. and view G_n , Δ , etc. as functions $\mathcal{H} \rightarrow \mathbb{C}$, where $\mathcal{H} = \{z \mid \text{Im}(z) > 0\}$ is the complex upper half plane. Importantly, all functions mentioned in this section are meromorphic on \mathcal{H} .

The last important results in this section are the *Fourier expansions* of G_n , Δ , and J . For example, using the Riemann ζ function and writing σ_a for the divisor function $\sigma_a(n) = \sum_{d|n} d^a$ and $q = e^{2i\pi\tau}$, we have the following:

► **Theorem 6** (Fourier expansion of G_n). *For even n , $G_n(\tau)$ has the following Fourier expansion at $\tau = i\infty$: $G_n(\tau) = 2\left(\zeta(n) + \frac{(2i\pi)^n}{(n-1)!} \sum_{k \geq 1} \sigma_{n-1}(k) q^k\right)$.*

We also formalised similar Fourier expansions for Δ and J . The Fourier coefficients of these do not have such simple closed forms, but we derive useful recurrences for them. These expansions show that G_n , Δ , and J are “meromorphic at $i\infty$ ”, which will be important later.

4 Modular forms

4.1 The modular group

The Möbius transformations of the form $z \mapsto \frac{az+b}{cz+d}$ form a group under function composition. This is the projective linear group $\text{PSL}(2, \mathbb{Z})$, also known as the *modular group* Γ . This group is related to the functions G_n , Δ , J above because they satisfy simple functional equations under composition with elements from the modular group, namely if $h(z) = \frac{az+b}{cz+d}$ then $G_n(h(z)) = (cz+d)^n G_n(z)$ and $\Delta(h(z)) = (cz+d)^{12} \Delta(z)$ and $J(h(z)) = J(z)$.

In Isabelle, we represent the modular group as a type `modgrp`. This is a quotient type of the set of tuples (a, b, c, d) with $a, b, c, d \in \mathbb{Z}$ and $ad - bc = 1$ modulo a relation that identifies (a, b, c, d) and $(-a, -b, -c, -d)$. We show that this is a group, which we write multiplicatively.

Two special kinds of modular transformations are shifts $T_n(z) = z + n$ and “mirror-inversions” $S(z) = -1/z$. Notably, any modular transformation can be decomposed (non-uniquely) into a product of S and T_n . We formalise this fact as an induction rule:

```
lemma modgrp_induct_S_shift [case_names id S shift]:
  fixes P:: "modgrp => bool"
  assumes "P 1" and "\x. P x => P (S_modgrp * x)"
    and "\x n. P x => P (shift_modgrp n * x)"
  shows "P x"
```

4.2 Fundamental regions

Consider a subgroup G of the modular group. We now consider two points in the upper half-plane \mathcal{H} to be equivalent whenever there exists a transformation in G that maps one to the other:

► **Definition 7** (equivalence under a subgroup of the modular group). *Let G be a subgroup of the modular group `modgrp`, and τ and τ' be two points in the upper half-plane \mathcal{H} . We consider τ and τ' to be equivalent under G if $\tau' = f\tau$ for some f in G .*

We can designate a canonical representative for each equivalence class e.g. by picking a sub-region of \mathcal{H} that contains exactly one representative of each class. The interior of a region that satisfies this is called a *fundamental region*.

► **Definition 8** (Fundamental region). *An open subset R of \mathcal{H} is a fundamental region of G provided that:*

- *No two distinct points of R are equivalent under G .*
- *If $\tau \in \mathcal{H}$ then there is a point τ' in the closure of R such that τ' is equivalent to τ .*

Next we show that a particular region is indeed a fundamental region of the full modular group. We call this the *standard fundamental region* \mathcal{R}_Γ :

► **Theorem 9.** *The open set $\mathcal{R}_\Gamma = \{\tau \in H \mid |\tau| > 1, |\operatorname{Re}(\tau)| < \frac{1}{2}\}$ is a fundamental region for Γ .*

4.3 Removing removable singularities

One issue that arises in formalising complex analysis is that on paper, removable singularities are essentially ignored completely. For example, if we have the functions $f(z) = z$ and $g(z) = 1/z$ then a mathematician would write $f(z) \cdot g(z) = 1$. In a theorem prover like Isabelle/HOL, this does not work: at least not if f and g are functions of type $\mathbb{C} \rightarrow \mathbb{C}$.

Our solution is to introduce a special type to capture meromorphic complex functions modulo removable singularities. Since our main interest later on will be functions on the upper half plane $\mathcal{H} = \{z \mid \operatorname{Im}(z) > 0\}$, we additionally restrict the functions to that domain.

To be precise: our type `mero_uhp` consists of those functions $f : \mathbb{C} \rightarrow \mathbb{C}$ that are meromorphic on \mathcal{H} and return 0 at their poles and outside \mathcal{H} . This captures exactly the mathematical idea of meromorphic functions on \mathcal{H} .

Conversion of a “normal” complex function f to the `mero_uhp` type is done by restricting f to the appropriate domain and fixing removable singularities. The latter is done with the very useful function `remove_sings`:

```
definition remove_sings :: "(complex  $\Rightarrow$  complex)  $\Rightarrow$  complex  $\Rightarrow$  complex" where
  "remove_sings f z = (if  $\exists c. f -z \rightarrow c$  then Lim (at z) f else 0)"
```

This function takes a complex function (assumed to be meromorphic) and returns a version of that function with all removable singularities removed and all poles totalised to 0.

With this, we can now also define basic arithmetic on `mero_uhp` and prove that it is a field and a \mathbb{C} -vector space, which would not be possible for the normal function type.

This type `mero_uhp` now forms the basis for our formalisation of modular forms and modular functions.

4.4 Definition of modular forms

Next we will finally define modular forms and related concepts, namely as “sufficiently nice” functions that satisfy interesting equations under composition with modular transformations.¹

► **Definition 10.** *A weakly modular form of integer weight k w.r.t. a subgroup G of the modular group is a meromorphic function $f : \mathcal{H} \rightarrow \mathbb{C}$ that satisfies the functional equation $f(h(z)) = (cz + d)^k f(z)$ for any $h(z) = \frac{az+b}{cz+d}$ with $h \in G$.*

By adding more conditions, we can define the following concepts.

- if f is additionally meromorphic at the cusps, we call it a *meromorphic form*
- if f is even holomorphic (including at the cusps), we call it a *modular form*
- a meromorphic form of weight 0 is called a *modular function*

Here, “meromorphic at the cusps” means that $f(h(z))(cz + d)^{-k}$ has a meromorphic Fourier expansion $\sum_{n \geq n_0} a_n e^{2i\pi n z}$ at $z = i\infty$ for all $h \in \Gamma$ (not just in G). For “holomorphic at the cusps”, we additionally require $n_0 \geq 0$.

¹ For simplicity, some of our definitions in Isabelle currently only work when G is the full modular group, but this will be generalised soon.

In Section 3 we have already seen that G_n is a modular form of weight n for $n \geq 3$, Δ is a modular form of weight 12, and J is a modular function.

Apostol does not use the terms “weakly modular form” and “meromorphic form” at all, but we find that they make the formalisation more modular: they allow e.g. the valence formula (below) to be shown directly for meromorphic forms rather than deriving them separately for modular forms and modular functions. This is a typical case where the educational approach of Apostol’s textbook clashes with the needs of formalisation.

4.5 The valence formula

The central result in our formalisation so far is the valence formula for meromorphic forms. It relates the number of zeros of a modular form to the number of its poles:

► **Theorem 11.** *Let f be a non-zero meromorphic form of weight k on the full modular group Γ . Then the sum of the multiplicities of the zeros of f inside the closure of \mathcal{R}_Γ minus the sum of the multiplicities of its poles in the same set is exactly $k/12$.*

Several caveats apply here about how to count zeros and poles directly at the border of the region: any point on the border is weighted with $\frac{1}{2}$, except for the points $\pm\frac{1}{2} + \frac{\sqrt{3}}{2}i$, which are weighted with $\frac{1}{6}$. It should also be noted that $i\infty$ may also be a zero or pole and must be counted accordingly (with weight 1).

The proof of the valence formula was the most difficult to formalise so far. The basic idea is simple: we apply the argument principle and integrate along a contour that is essentially a finite version of the border of \mathcal{R}_Γ . Due to the symmetries of \mathcal{R}_Γ and f , most of the integral cancels, only $k/12$ remains plus the contribution of the potential zero or pole at $i\infty$.

The problem is that there may be zeros or poles directly on the border itself and we need to add little “wiggles” to avoid these and account for the error made by this. This is easy to justify on paper, but not in a theorem prover. We eventually solved this problem by using the “Wiggle Framework”, which the first author developed specifically for this proof (but with similar future applications in mind). It allows deforming integration contours and relating them to the original contour. We are currently planning to eventually replace this framework with a much simpler approach based on a generalised residue theorem that allows singularities on the integration path. [6]

For modular functions, the valence formula is particularly striking: it means that the number of zeros of a modular function $f(z)$ is exactly the same as the number of its poles. Moreover, since the number of zeroes in $f(z) - c$ is the same as that of $f(z)$, we can even say that f takes on all complex values equally often. In particular, J is a bijection between \mathcal{R}'_Γ and \mathbb{C} (where \mathcal{R}'_Γ denotes the union of \mathcal{R}_Γ and the left half of its closure).

Apostol uses this last fact to give a relatively simple proof of Picard’s little theorem (a non-constant entire function takes every value in \mathbb{C} with at most one exception). We formalised this as well, but it turned out to be not quite so simple: in particular, we had to first prove the stronger fact that J is a *covering* between \mathcal{H} and \mathbb{C} , which was a reasonably simple, but somewhat tedious and definitely non-trivial proof. It is surprising that Apostol does not mention this seemingly indispensable bit of work in his proof.

Another straightforward application of the valence formula is to determine the dimension of the vector space of modular forms of weight k , the formalisation of which is ongoing work.

5 Dedekind's η function

► **Definition 12** (The Euler function ϕ and Dedekind's η function). Define $\phi(q) = \prod_{k \geq 1} (1 - q^k)$ and $\eta(z) = e^{i\pi z/12} \phi(e^{2i\pi z})$. They are holomorphic for $|q| < 1$ and $z \in \mathcal{H}$, respectively.

Dedekind's η function is not a modular form in the sense that Apostol defines, but it does display interesting behaviour under the two generators $z \mapsto z + 1$ and $z \mapsto -1/z$ of the modular group:²

► **Theorem 13.** $\eta(z + 1) = e^{i\pi/12} \eta(z)$ and $\eta(-1/z) = \sqrt{-iz} \eta(z)$.

Using these two relations and our induction rule for the modular group, one can show the following more general equation:

► **Theorem 14.** If $h(z) = \frac{az+b}{cz+d}$ is an element of the modular group, then $\eta(h(z)) = \varepsilon_h \sqrt{cz+d} \eta(z)$ where ε_h is a 24th root of unity depending on h but not on z .

This ε_h has an explicit (albeit complicated) formula in terms of Dedekind sums which we shall not show here. It is noteworthy that our definition of ε_h and our version of the theorem differ somewhat from Apostol's, since ours work for any value of c while he requires $c > 0$.

Interestingly, Apostol proves Theorem 14 directly using Iseki's formula: a technical lemma whose proof is four pages of dense calculations and which is never used again. We chose *not* to formalise Iseki's formula and to instead follow a simpler approach outlined in the appendix of the second edition of the book: we first follow Apostol's proofs for Theorem 13 and then obtain Theorem 14 from it.

An interesting consequence of Theorem 14 is that η^{24} is a modular form of weight 12. Combining this with the valence formula, one obtains (relatively easily) a remarkable connection: $\Delta(z) = (2\pi)^{12} \eta(z)^{24}$.

6 Discussion and related work

The tradition of formalising textbooks dates back to Jutting's formalisation of Landau's *Foundations of Analysis* using AUTOMATH [8] in 1977. The challenge is about the volume of material but also the obligation to cover everything rather than to pick and choose. Although we did not have time to formalise the entire text, we did cover half of the eight chapters.

We built upon a huge library of prior material, including Laurent series, winding numbers, Dirichlet series, polynomial factorisation and Bernoulli numbers, all of which had to interoperate. We worked under the handicap that none of us is a number theorist. Perhaps for this reason, many of the Isabelle/HOL proofs are considerably longer than Apostol's. We invested some effort in making the formal proofs clear, through Isabelle's structured proof language, hoping to retain some of the pedagogical value of the original text. The four chapters (1, 2, 3, 7) respectively consist of 12K, 10K, 4K, and 3K lines of proof scripts including comments. Together with other supporting material, the project has already exceeded 53,000 lines.

Much number theory has been formalised in other proof assistants, chiefly Lean. To our knowledge, ours was the first treatment of elliptic functions and modular forms in a theorem prover, although we are aware of more recent unpublished work by Birkbeck [3] in Lean covering mostly the definition of modular forms and Eisenstein series. This work is now also part of Mathlib 4.

² Here, $\sqrt{\cdot}$ denotes the standard branch of the complex square root where $\operatorname{Re}(\sqrt{z}) \geq 0$ for all $z \in \mathbb{C}$ and $\operatorname{Im}(\sqrt{x}) > 0$ for any real $x < 0$.

7 Conclusions

The formalisation of a textbook remains challenging. Our impression was that Apostol’s proofs were clear overall, and wherever they were, the formalisation process was straightforward, regardless of the mathematical tools required. There were however some gaps, mistakes, and informal arguments that took time to overcome, but none that were serious. Graduate-level analytic number theory can be formalised in Isabelle/HOL without undue effort.

References

- 1 Tom M. Apostol. *Introduction to Analytic Number Theory*. Springer, 1976.
- 2 Tom M. Apostol. *Modular Functions and Dirichlet Series in Number Theory*. Springer, 1990.
- 3 Chris Birkbeck. ModularForms. GitHub repository. URL: <https://github.com/CBirkbeck/ModularForms>.
- 4 Manuel Eberl. Nine chapters of analytic number theory in Isabelle/HOL. In *10th International Conference on Interactive Theorem Proving (ITP 2019)*, volume 141 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 16:1–16:19, 2019.
- 5 Manuel Eberl, Anthony Bordg, Lawrence C. Paulson, and Wenda Li. Formalising half of a graduate textbook on number theory (formal proof development), June 2024. doi: 10.5281/zenodo.12586104.
- 6 Norbert Hungerbühler and Micha Wasem. Non-integer valued winding numbers and a generalized residue theorem. *Journal of Mathematics*, 2019:1–9, March 2019. doi: 10.1155/2019/6130464.
- 7 Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer, 2002.
- 8 L. S. van Benthem Jutting. *Checking Landau’s “Grundlagen” in the AUTOMATH System*. PhD thesis, Eindhoven University of Technology, 1977.

Graphical Rewriting for Diagrammatic Reasoning in Monoidal Categories in Lean4

Sam Ezeh   
Durham University, UK

Abstract

We present Untangle, a Lean4 extension for Visual Studio Code that displays string diagrams for morphisms inside monoidal categories, allowing users to rewrite expressions by clicking on natural transformations and morphisms in the string diagram. When the user manipulates the string diagram by clicking on natural transformations in the Graphical User Interface, it attempts to generate relevant tactics to apply which it then inserts into the editor, allowing the user to prove equalities visually by diagram rewriting.

2012 ACM Subject Classification Theory of computation → Interactive proof systems; Human-centered computing → Graphical user interfaces

Keywords and phrases Interactive theorem proving, Lean4, Graphical User Interface

Digital Object Identifier 10.4230/LIPIcs.ITP.2024.41

Category Short Paper

Supplementary Material *Software (Source Code)*: <https://github.com/dignissimus/Untangle> [7]

1 Introduction

String diagrams are a visual language for reasoning about morphisms in monoidal categories where rewriting rules for expressions using equalities becomes a series of visual transformations on a diagram that appear in a variety of mathematical contexts, both pure and applied. In pure settings string diagrams appear when reasoning about objects whose structure can be modelled by symmetric monoidal categories such as Hopf algebras and braided monoidal categories [1] and in applied settings, string diagrams appear as tools for reasoning about resource-sensitive systems in fields such as quantum mechanics and circuit theory [3]. String diagrams also simplify the exposition of ideas otherwise obscured by terse written notation. In this paper, we contribute a Lean4 extension that renders morphisms and natural transformations inside a monoidal category as string diagrams using the ProofWidgets [13] framework for building Graphical User Interfaces as part of ongoing work.

2 Background and Related Work

2.1 Lean

Lean [12] is a dependently typed programming language and proof assistant based on the Calculus of Inductive Constructions. In Lean, users construct mathematical statements by building types and prove these statements by exhibiting terms that inhabit these types. Lean also provides a “tactic” language that allows users to prove statements via a series of commands called tactics that rewrite the current statement that the user intends to prove by using existing lemmas and hypotheses. Lean exposes the statement that the user would like to prove, referred to as the “goal”, alongside the assumptions and hypotheses at hand which are referred to collectively as the “goal state” which can then be accessed and manipulated programmatically through the use of a rich, monadic meta-programming interface. This



© Sam Ezeh;
licensed under Creative Commons License CC-BY 4.0

15th International Conference on Interactive Theorem Proving (ITP 2024).

Editors: Yves Bertot, Temur Kutsia, and Michael Norrish; Article No. 41; pp. 41:1–41:8

Leibniz International Proceedings in Informatics



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

extension integrates with the Lean4 proof assistant as an extension that generates tactics in response to user interaction with the rendered string diagram that it presents in the Graphical User Interface.

2.2 Penrose

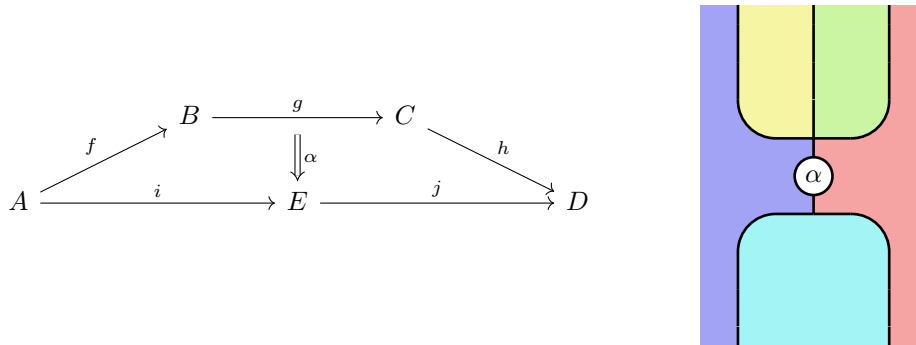
Penrose [14] is a tool for declaratively constructing mathematical diagrams that allows developers to construct a DSL (Domain Specific Language) that describes the types of mathematical objects that the developer would like to display and the relations between them and a stylesheet to describe how the system should render these mathematical objects. After specifying a DSL, the developer can programmatically construct diagrams containing instances of these mathematical objects by generating scripts written using this DSL. Penrose will then generate images depicting these mathematical objects in the manner described by the user-provided stylesheet. Penrose excels for our use case as it does not rely on a fixed library of visualisation tools and the DSL allows the developer to define visual representations in its constraint-based specification language. We use Penrose to visually render string diagrams. ProofWidgets [13] is a general-purpose framework for building Graphical User Interfaces in Lean, allowing developers to build visual “widgets” that interact with Lean. ProofWidgets provides an API that allows for effortless construction of interactive widgets and extensions for Lean4 by allowing the developer to create composable components and includes a built-in component that integrates the Penrose system for constructing mathematical diagrams. We build the extension using the ProofWidgets framework, allowing us to write user interface logic in JavaScript and relay information from the user interface to the Lean server and Visual Studio Code using RPCs (Remote Procedure Calls) and also write RPC handlers in Lean.

2.3 Graphical proof assistants

There are a handful of stand-alone visual proof assistants that allow the user to work by manipulating string diagrams, such as Globular [2], homotopy.io [5] and Quantomatic [9]. Graphical proof assistants represent terms as diagrams and allow users to manipulate diagrams according to rewrite rules that correspond to equalities between two terms. In these proof assistants, users interact with graphical proof assistants by clicking on diagram elements that represent mathematical objects in order to apply rewrite rules to them and the neighbouring elements. These proof assistants typically exist either as online web applications or downloadable executables as opposed to integrating with pre-existing proof assistants. In addition to these tools, there is an in-progress Pull Request to Lean’s Mathlib for displaying string diagrams authored by Yuma Mizuno [11], although it does not aim to rewrite terms in the goal.

2.4 String diagrams

Commutative diagrams are a graphical representation depicting morphisms inside a category where, typically, objects are represented as points or vertices, morphisms are represented as arrows between these points and 2-morphisms are presented as arrows between arrows. Dualising this representation, one arrives at string diagrams where objects are represented as faces, morphisms as lines and 2-morphisms as points or vertices.



■ **Figure 1** The corresponding commutative diagram and string diagram.

For example, let A, B, C, D and E be objects inside a category, let $f : A \rightarrow B, g : B \rightarrow C, h : C \rightarrow D, i : A \rightarrow E, j : E \rightarrow D$ be morphisms inside this category and let $\alpha : h \circ g \circ f \Rightarrow j \circ i$ be a 2-morphism between them. As a commutative diagram and as a string diagram, this would appear as in Figure 1 where every object is represented by a coloured planar face.

String diagrams allow for reasoning about morphism equalities with soundness guaranteed by coherence theorems [8] and provide a graphical representation for certain structures inherent to monoidal categories. For example, the associativity of morphism composition is inherent to the string diagram representation as the composition of morphisms in string diagrams is represented by positioning morphisms such that they are adjacent. More generally, string diagrams are for bicategories of which monoidal categories are a special case.

3 Proof of Concept

We implement “Untangle” as a proof of concept and use it to prove example statements about monads: monoid objects in the monoidal category of endofunctors and natural transformations between them. This work is accessible online at <https://github.com/dignissimus/Untangle>. When the current proof goal is an equality, untangle renders string diagrams for the morphisms on both sides of the equality as seen in Figure 2. The user may then prove the equality of the two morphisms by manipulating the diagram by clicking on morphisms on either side of the diagram. When the user has selected the section of the diagram that they would like to rewrite, the extension constructs a tactic to apply a relevant theorem to rewrite the goal statement in the sub-expressions that the diagram components represent. After the goal statement updates in the Lean info view, the Graphical User Interface re-renders the diagram so that it represents the updated goal statement.

3.1 Example workflow

If, for example, one were to prove the equality between $\mu_X \circ f \circ \mu_X$ and $\mu_X \circ T\mu_X \circ TTf$ for some monad T equipped with natural transformations $\mu : T^2 \rightarrow T$ and $\eta : \mathbf{1} \rightarrow T$ satisfying the monad laws and a morphism $f : X \rightarrow TX$ for some object X . We may begin by using the naturality of μ to argue that $\mu_X \circ f \circ \mu_X$ is equal to $\mu_X \circ \mu_{TX} \circ TTf$, we could then proceed to use the associativity law for the μ natural transformation to argue that this equals $\mu_X \circ T\mu_X \circ TTf$, completing the proof.

41:4 Graphical Rewriting for Diagrammatic Reasoning in Lean4

In Lean, this statement would appear as follows:

```
example [Category C] {T : Monad C} {X : C} {f : X → T.obj X}
  : T.μ.app X >> T.map f >> T.μ.app X
    = T.map (T.map f) >> T.map (T.μ.app _) >> T.μ.app _
  := by with_panel_widgets [Untangle] {
}
```

When the user places the cursor inside the braces inside Visual Studio Code, the extension will render string diagrams for both of the morphisms on each side of the equality that appear inside the Lean infoview as in Figure 2. Visually, the proof of the theorem consists of pulling up the point that represents the morphism f as a simple planar deformation then using the monad associativity law to swap the order of the μ natural transformations.

With the extension, clicking on the morphism f and the natural transformation μ generates the following tactic and enters it into the lean editor:

```
conv => {
  enter [1];
  slice 1 2;
  rw [
    ← (Monad.μ T).naturality (f),
    CategoryTheory.Functor.comp_map
  ];
};
try simp only [CategoryTheory.Category.assoc];
```

This tactic does the following:

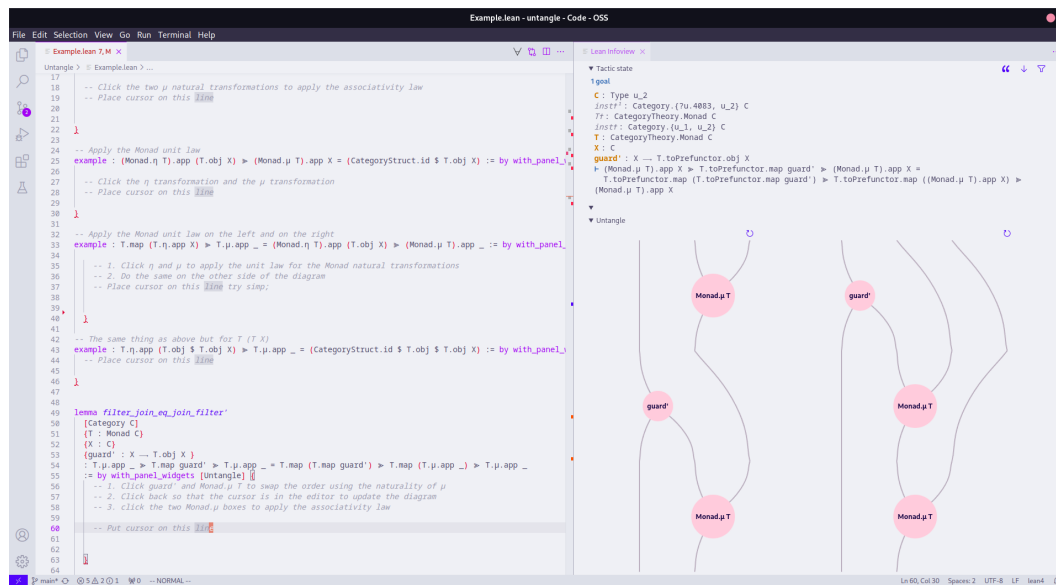
- Selects the left-hand side of the expression
- Uses the indices 1 and 2 to select a “slice” containing the subexpression to re-write
- Applies the naturality condition for the μ natural transformation to rewrite the goal statement
- Rewrites $(T \circ T)f$ as TTf
- Attempts to simplify the expression using the associativity of morphism composition.

This updates the goal state and the extension renders a new diagram that has the order of f and μ swapped.

After this, we can click on the two μ morphisms to apply the associativity law which inserts the following tactic into the editor:

```
conv => {
  enter [1];
  slice 2 3;
  rw [← Monad.assoc];
};
try simp only [CategoryTheory.Category.assoc];
```

The equality has now been proved and the Lean type-checker accepts the statement and proof.



■ **Figure 2** Using the extension to prove morphism equality.

4 Implementation

4.1 Data structures

In a similar manner to Comfort et al. [4], we represent diagrams as a series of diagram components without association information where each diagram component is represented as the number of inputs into the natural transformation, its number of outputs and its location or “offset” in its level in the string diagram referring to the number of wires or strings that exist to the left of the component at its level. See Delpuch and Vicary [6] for more information about this data structure.

4.2 Parsing morphism expressions

We parse morphisms, natural transformation components and objects from the goal statement into an internal representation that closely resembles the syntax tree for the Lean expressions that represent them by using a basic recursive-descent parser. While parsing morphisms, we use type information from Lean to tag expressions with semantic labels such as “functor”, “morphism” or “natural transformation component”. We use this semantic information to infer which tactics to apply when the user interacts with the diagram in the user interface.

4.3 Displaying diagrams

To display string diagrams to the user, we parse expressions into diagram components and use our representation of diagram components to construct declarative statements in the extension’s Penrose DSL which Penrose uses to render a diagram according to the specification in the extension’s stylesheet. We then construct a Penrose component using the ProofWidget framework and include it in the extension’s section of the info view.

4.4 Handling user input

We handle user input by writing JavaScript to listen to click events in Visual Studio Code. After the user has interacted with the diagram, we send an RPC that contains information describing the interaction. This includes the location of the the elements that the user clicked on in the diagram and information about the state of the editor such as the position of the mouse cursor.

4.5 Pattern matching and tactic selection

We implement a simple rule-based system for deciding which tactic to select by using a combination of syntactic information from Lean’s representation of the expression and the inferred semantic tags. For example, if the user selects two μ natural transformations we infer that the user would like to transform the diagram by swapping their order and rewrite the expression using the μ associativity law: $\mu_X \circ T\mu_X = \mu_X \circ \mu_{TX}$. We identify the direction in which we need to perform the rewrite by looking at the syntactic structure of the expression. For the example of the μ associativity law, we determine the direction of the rewrite by comparing the functor lifts of each of the two instances of the μ natural transformation in the diagram.

4.6 Rewriting up to associativity

A graphical proof interface with Lean faces a unique challenge with rewriting and associativity. In string diagrams, two expressions that differ only by association are represented identically however, in Lean, two expressions that differ only by their associations are not necessarily equivalent and we must associate theorems in the correct way before we may apply them to expressions. For example, if the context contains a lemma that states $f \circ g = f'$ we may reduce $(f \circ g) \circ h$ to $f' \circ h$ but we may not immediately reduce $f \circ (g \circ h)$ without re-associating the expression. This poses problems as to translate the user’s theorem into a tactic to rewrite the goal statement as we must get the association right in terms of both the hypotheses and the resulting goal state. We tackle this by making use of Lean’s “conv” tactic and in particular, its “slice” command. When parsing the composition of morphisms in the equalities, we calculate the “location” of each morphism in the expression. We then use this as input to Lean’s slice command which allows us to specify indices to select subexpressions in the goal statement and re-associate them into a normal form by repeatedly applying the `Category.assoc` lemma which asserts the equality of $f \circ (g \circ h)$ and $(f \circ g) \circ h$

4.7 Extensibility

From a Software Engineering perspective, we achieve extensibility in the extension by defining an abstract `GraphicalLanguage` structure which developers can implement. A developer defines graphical languages for new structures by implementing a function that parses Lean expressions and produces a representation of the expression’s structure, which the extension uses to render the diagram. Additionally, the developer defines a function to handle click events, producing a list of tactics to insert into the editor, and implementing functions that determine the cosmetic styling of elements in the diagram.

4.8 Entering tactics into the editor

After the RPC handler has received the message describing the user’s interaction with the diagram and has constructed the tactic that is to be entered into the user interface, we enter tactics into the Visual Studio Code editor by embedding the generated tactic and information about where in the document we should insert it in the response to the RPC which we then handle in the JavaScript code. The JavaScript then uses an “EditorContext” to relay this information to Visual Studio code.

5 Conclusion and Future Work

In conclusion, we have created a Lean4 extension capable of rendering morphisms inside a monoidal category as string diagrams to assist users in proving equalities between morphisms in monoidal categories. Untangle is still in the early stages of development and, in the future, we seek to implement rewrite rules for other monoidal structures such as comonoids, bimonoids and Hopf monoids. In addition to supporting more monoidal structures we want to achieve the following:

- At present, the user interface is simple: rewrite rules are applied by clicking on two morphisms and natural transformations in the diagram. This simplicity limits the type of rewrite rules that the extension can support while remaining intuitive. In the future, we want to extend the user interface by implementing a context menu for a more explicit interface over what tactic the extension selects.
- There are simple graphical rules for checking whether certain rewrite rules or transformations can be applied to a diagram and while the Lean4 type-checker will refuse erroneous applications of rewrite rules and output an error message, using these simple diagrammatic checks to alert the user about incorrect rewrites within the graphical interface would provide the user with a better experience.
- While existing Lean tactics continue to work as normal with the assumptions and hypotheses in scope, the extension doesn’t provide a way to apply these lemmas graphically. Allowing the user to graphically apply lemmas from the goal state to the diagram via the user interface would be a great addition to the user experience.
- Currently, all rewrite rules are pre-declared for mathematical objects that exhibit monoidal structure. For example, the extension can render diagrams for classes of expressions in the language of HopfAlgebra and Monads and we have written rewrite rules for the monoidal structure of Monads and this would be repeated for new mathematical objects with monoidal structure. While this process is not tedious, it would be much better if there were a more general way of building rewrite rules for mathematical objects with monoidal structures.
- The extension lacks parsing for certain types of expressions in the statement. While we currently parse TTf and $Tf \circ Tg$ into the internal representation of the syntax tree, we do not currently parse $(T \circ T)f$ or $T(f \circ g)$. In the future, we want to parse a wider variety of expressions and provide more comprehensive coverage over the types of expressions the user can work with diagrammatically.

References

- 1 John Baez and Mike Stay. *Physics, topology, logic and computation: a Rosetta Stone*. Springer, 2011.
- 2 Krzysztof Bar, Aleks Kissinger, and Jamie Vicary. Globular: an online proof assistant for higher-dimensional rewriting. *Logical Methods in Computer Science*, 14, 2018.

- 3 Filippo Bonchi, Fabio Gadducci, Aleks Kissinger, Pawel Sobocinski, and Fabio Zanasi. String diagram rewrite theory ii: Rewriting with symmetric monoidal structure. *Mathematical Structures in Computer Science*, 32(4):511–541, 2022.
- 4 Cole Comfort, Antonin Delpuch, and Jules Hedges. Sheet diagrams for bimonoidal categories. *arXiv preprint arXiv:2010.13361*, 2020.
- 5 Nathan Corbyn, Lukas Heidemann, Nick Hu, Chiara Sarti, Calin Tataru, and Jamie Vicary. homotopy.io: a proof assistant for finitely-presented globular n -categories. *arXiv preprint arXiv:2402.13179*, 2024.
- 6 Antonin Delpuch and Jamie Vicary. Normalization for planar string diagrams and a quadratic equivalence algorithm. *Logical Methods in Computer Science*, 18, 2022.
- 7 Sam Ezeh. Dignissimus/untangle: Graphical rewriting for diagrammatic reasoning in monoidal categories in lean4, 2024. Software (visited on 2024-08-19). URL: <https://github.com/dignissimus/Untangle>.
- 8 André Joyal and Ross Street. The geometry of tensor calculus, i. *Advances in Mathematics*, 88(1):55–112, 1991. doi:10.1016/0001-8708(91)90003-P.
- 9 Aleks Kissinger and Vladimir Zamdzhiev. Quantomatic: A proof assistant for diagrammatic reasoning. In *Automated Deduction-CADE-25: 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings 25*, pages 326–336. Springer, 2015.
- 10 Daniel Marsden. Category theory using string diagrams. *arXiv preprint arXiv:1401.7220*, 2014.
- 11 Yuma Mizuno. Feat: String diagram widget by yuma-mizuno (pull request #10581) leanprover-community/mathlib4, February 2024. URL: <https://github.com/leanprover-community/mathlib4/pull/10581>.
- 12 Leonardo de Moura and Sebastian Ullrich. The lean 4 theorem prover and programming language. In André Platzter and Geoff Sutcliffe, editors, *Automated Deduction – CADE 28*, pages 625–635, Cham, 2021. Springer International Publishing.
- 13 Wojciech Nawrocki, Edward W. Ayers, and Gabriel Ebner. An Extensible User Interface for Lean 4. In Adam Naumowicz and René Thiemann, editors, *14th International Conference on Interactive Theorem Proving (ITP 2023)*, volume 268 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 24:1–24:20, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ITP.2023.24.
- 14 Katherine Ye, Wode Ni, Max Krieger, Dor Ma’ayan, Jenna Wise, Jonathan Aldrich, Jonathan Sunshine, and Keenan Crane. Penrose: From mathematical notation to beautiful diagrams. *ACM Trans. Graph.*, 39(4), 2020.